# 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 focused on the effect of the sortedness of Data on the performance of Data Compression Algorithms. Our main aim with this project was to understand how the performance of various Data Compression Algorithms changes with varying levels of sortedness. We implemented five different data compression algorithms: Run Length Encoding, Huffman Encoding, Delta Encoding, LZ77 Encoding, and Snappy Algorithm. We applied these algorithms to different combinations of Data having different levels of sortedness and percentages of unique values. We then compared their compression times and compression ratios. The results of our experiments indicate that Snappy Algorithm and Delta Encoding are the two best encoding algorithms for sorted, near sorted, and unsorted Data. +++++

## 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. 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. The existing research on this matter only focuses on completely sorted or unsorted Data. Therefore, we aimed to explore the performance of Data Compression Algorithms on Data with varying degrees of sortedness. The result of our research provides valuable insights and can help optimize Data Compression Algorithms for real-world scenarios.

### 2.2 Problem Statement

Our aim for this project was 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

The work we have done for this paper contributes greatly to the field of Data Compression. Initially, we generated a few simple workloads with varying levels of sortedness which ranged from unsorted, near sorted to completely sorted Data. We then explored the effects of these different levels of sortedness on the performance of five commonly used Data Compression Algorithms. We specifically looked at the compression ratio and the compression time of Run Length Encoding, Huffman Encoding, Delta Encoding, LZ77 Encoding, and Snappy Algorithm.

One of the main motives behind this project was to expand on the previous research that has been done on this topic by exploring the performance of these algorithms on near-sorted Data. This is one of the important contributions of this paper as real-world Data is rarely completely sorted or unsorted. The insights we derived from our research indicate that +++++

Finally, after understanding the performance of each of these algorithms for different levels of sortedness and with the different frequency distributions of the Data, we provide recommendations for the optimal use of Data Compression algorithms based on the level of sortedness of the Data. Our recommendations can be used to optimize the performance and efficiency of data compression algorithms for real-world scenarios.

All in all, this work provides valuable insights into the performance of data compression algorithms on varying degrees of sortedness and can be used to inform the development and application of these algorithms in practice.

# 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.

In this project, we focused on Data Sortedness and its impact on the performance of different Data Compression Algorithms. We implemented several algorithms for this purpose, including:

- Run Length Encoding: It is a simple yet efficient compression algorithm that is mostly used for compressing images, videos and data with long runs of repeated values. The algorithm basically replaces consecutive identical values with a count and the value.
- Huffman Encoding: It is a very popular compression algorithm which uses variable length codes to represent data. It assigns shorter codes to characters that occur frequently in the data, so that the compressed output takes less space than original data. It basically works by building a tree of characters based on their frequency of occurrence and then assign codes to each leaf.
- LZ77 Encoding: It is a losstless compression algorithm, it uses the approach of sliding window to find the identical patterns in the data. It encodes each pattern as the pattern, the length of the pattern and the starting index of the pattern.
- Delta Encoding: It works by storing the differences between two consecutive values in the data. It works greatly for data that has a predictive pattern or for the data that has a small range of values.
- Snappy Algorithm: It is a fast and lossless compression algorithm developed by Google. It gives more importance to speed rather than compression ratio. The algorithm works by using a combination of variable-length codes and a dictionary of frequently occurring patterns to compress the input data.

These 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: if we have time series data, then the order of data could be chronological.

If our data is sorted in some order, Data Compression Algorithms can exploit the repetitive patterns and the redundancy to efficiently compress the data. However, most research studying sortedness on compression performance assumes that the data is either completely sorted or unsorted, which is rarely the case in practice. Data is usually partially sorted, and the degree of sortedness depends on the application and the data source. Hence, we explored the performance of these popular Data Compression Algorithms with varying degrees of sortedness in this project.

# 4 Architecture

Our research focused on exploring the impact of varying levels of sortedness on the performance of different data compression algorithms. To this end, we implemented and benchmarked five data compression algorithms: Run Length Encoding, Delta Encoding, Huffman Encoding, LZ77 Encoding, and Snappy Algorithm.

The first step of our project was data preparation. We developed a workload generator in Python that generates different-sized workloads with varying K-L values. This allowed us to create datasets with different levels of sortedness, which we could use to evaluate the performance of the compression algorithms.

K-L values are a measure of sortedness in a Dataset. K represents the number of out of order elements in the Data, and L represents the maximum displacement of the out of order entry.

In the next step, we implemented the five data compression algorithms and benchmarked their performance. We evaluated the time taken for encoding and the compression ratio achieved by each algorithm. By comparing the results, we were able to identify the algorithms that performed best for datasets with varying degrees of sortedness.

## 4.1 Run Length Encoding

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.

**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
```

## 4.2 Huffman Encoding

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.

**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
```

## 4.3 Delta Encoding

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.

**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
```

## 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

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.

**Algorithm 5** Workload Generator

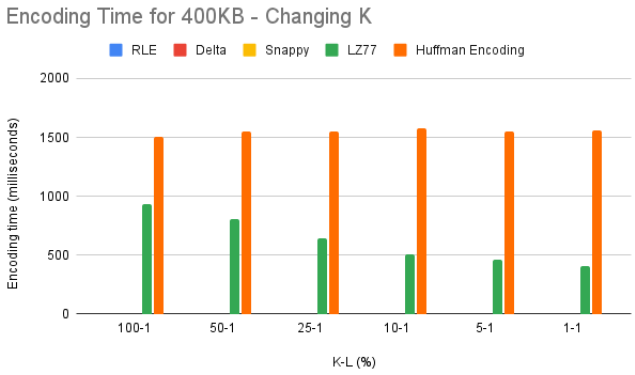
```
function WORKLOADGENERATOR(n, k, l)
    unique ← ⌊n * 0.1⌋
    duplicate ← ⌊n * 0.9⌋
    maxfreq ← ⌊duplicate/unique⌋
    data ← []
    for i ← 0 to unique − 1 do
        data.append(i)
    end for
    for i ← 0 to (n − unique) + 1 do
        num ← random integer between 0 and unique − 1
        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
```

# 5 Results

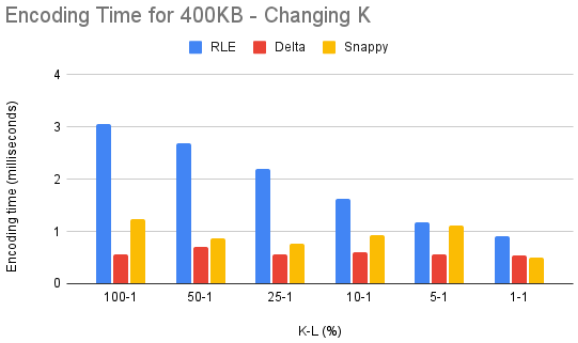
We conducted a series of experiments on the five data compression algorithms, utilizing various combinations of K-L values, workload sizes of 40KB, 100KB, 400KB, 1MB, 4MB, 10MB and frequency distributions of 10%, 25%, 50%, 75% and 100% of unique values in the workload. The ensuing section of this report will provide a more detailed analysis of these findings.

## 5.1 Encoding Time vs Various K-L values

### 5.1.1 Changing K for 10% Unique Values in the Workload



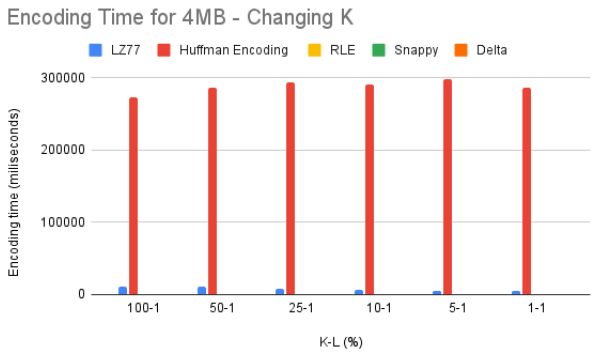

(a) All Algorithms



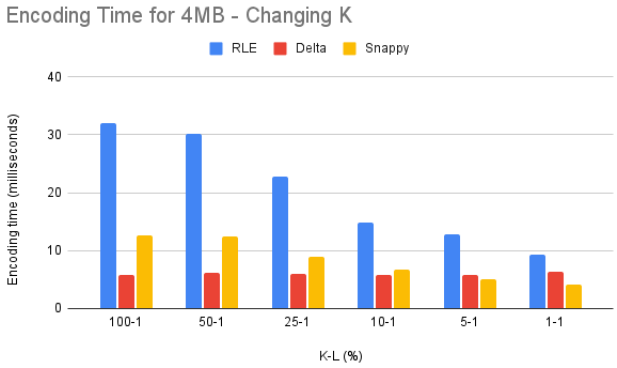

(b) Delta, RLE and Snappy Algorithm

Figure 1: Encoding Time for 400KB - Changing K

From the above two graphs, we can see the trends for the Encoding time of each Compression Algorithm when we changed the percentage of out-of-order queries. We can see that Huffman Encoding has a constant bar for all values, this happens because the frequency distribution of all the unique values in the workload is almost uniform, and hence encoding time is also similar. LZ77 and Run Length Encoding shows a decreasing trend where the encoding time decreases as the data becomes more and more sorted. We can also see that encoding times for Delta Encoding and Snappy Algorithm are similar throughout all the workloads because their encoding time is independent of the degree of sortedness of the workload.



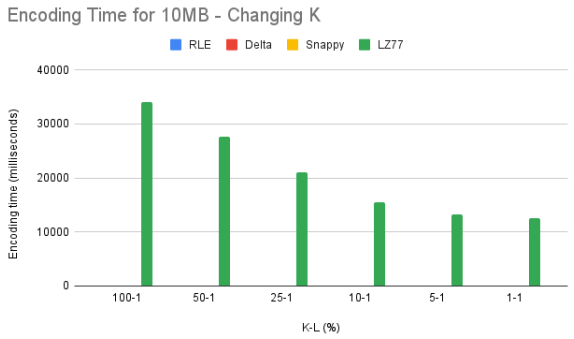

(a) All Algorithms



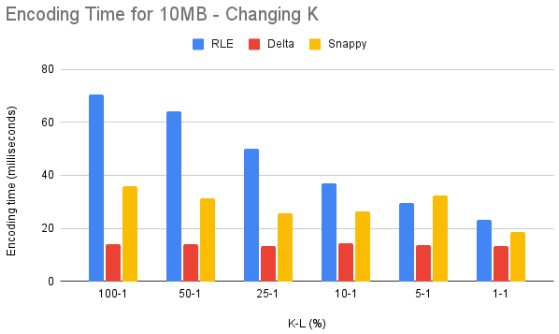

(b) Delta, RLE and Snappy Algorithm

Figure 2: Encoding Time for 4MB - Changing K

The graphs follow the same trend as what we saw for 400KB, where Huffman Encoding has almost constant Encoding time. LZ77 and RLE show decreasing trends and the encoding time for Delta Encoding and Snappy Algorithm don't seem to be dependent on the degree of sortedness of the Data.




(a) All Algorithms

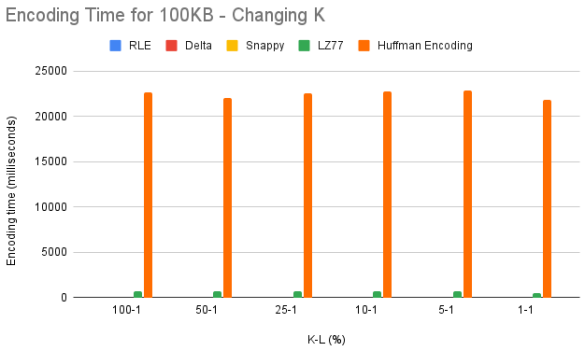



(b) Delta, RLE and Snappy Algorithm

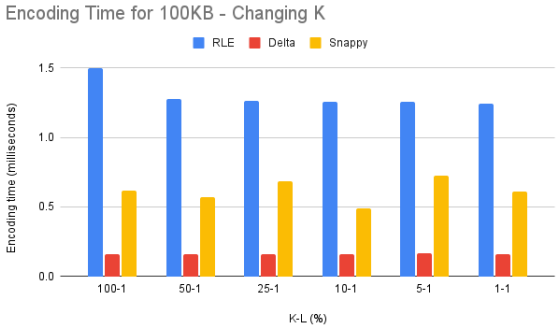
Figure 3: Encoding Time for 10MB - Changing K

After looking at the above two graphs for a 10MB workload, we can conclude that the Encoding time shows the same trend regardless of the size of the Workload. Although we were not able to record the time for Huffman Encoding as it was taking more than 10 minutes to run for each workload, from our previous experience we can hypothesize that it would take almost constant time as it did for previous workloads.

### 5.1.2 Changing K for 100% Unique Values in the Workload



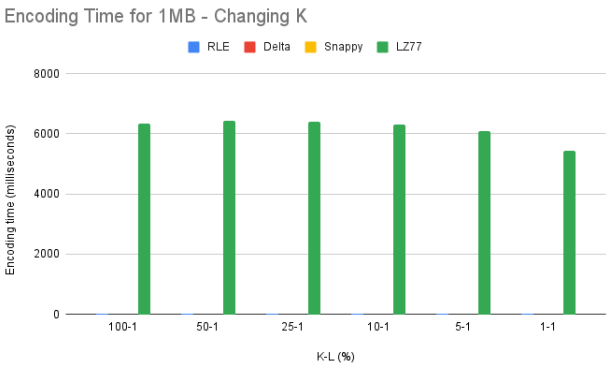

(a) All Algorithms



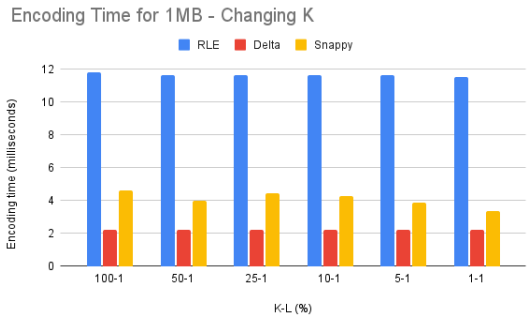

(b) Delta, RLE and Snappy Algorithm

Figure 4: Encoding Time for 100KB - Changing K

From the above two graphs, we can see the trends for the Encoding time of each Compression Algorithm when we changed the percentage of out-of-order queries. We can see that all the algorithms have almost similar encoding times for each degree of sortedness. This is happening because there is no pattern or duplicates in the data which is leading to higher and similar times for all degrees of sortedness. Also, for RLE the encoding time is higher for the 100-1 workload, which could be because of experimental errors or the cache behavior.



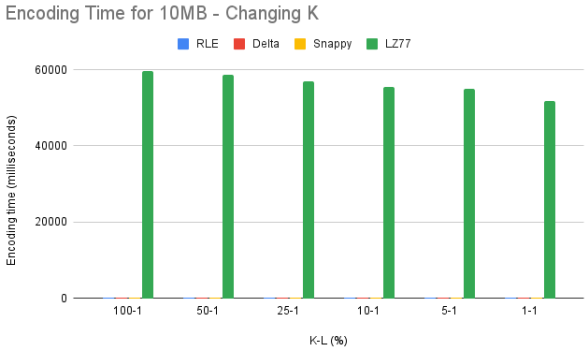

(a) All Algorithms



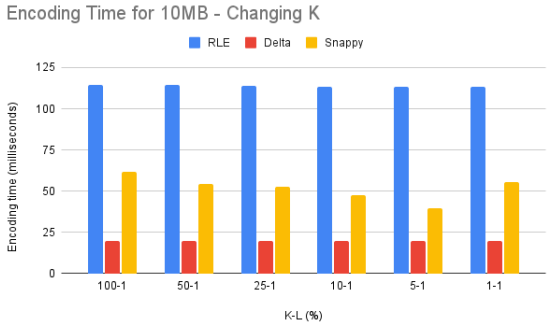

(b) Delta, RLE and Snappy Algorithm

Figure 5: Encoding Time for 1MB - Changing K

The graphs above follow the exact same trend that we saw for the 100KB workload. For this experiment, we couldn't record the encoding time for Huffman Encoding as it was taking more than 10 minutes for each workload, but we can speculate that Huffman Encoding will also have a similar trend as it had for 100KB workload.




(a) All Algorithms

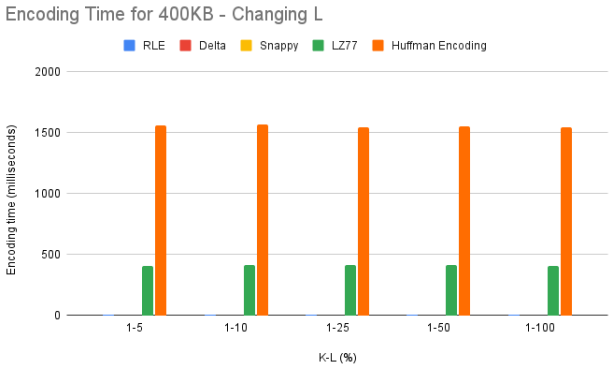



(b) Delta, RLE and Snappy Algorithm

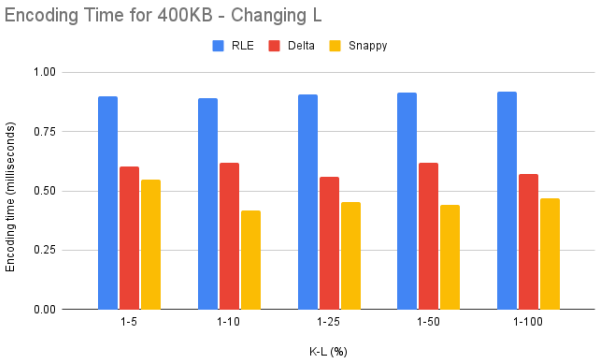
Figure 6: Encoding Time for 10MB - Changing K

After looking at the above two graphs for a 10MB workload, we can conclude that the Encoding time shows the same trend regardless of the size of the Workload. Although we were not able to record the time for Huffman Encoding as it was taking more than 10 minutes to run for each workload, from our previous experience, we can hypothesize that it would take almost constant time as it did for previous workloads.

### 5.1.3 Changing L for 10% Unique Values in the Workload



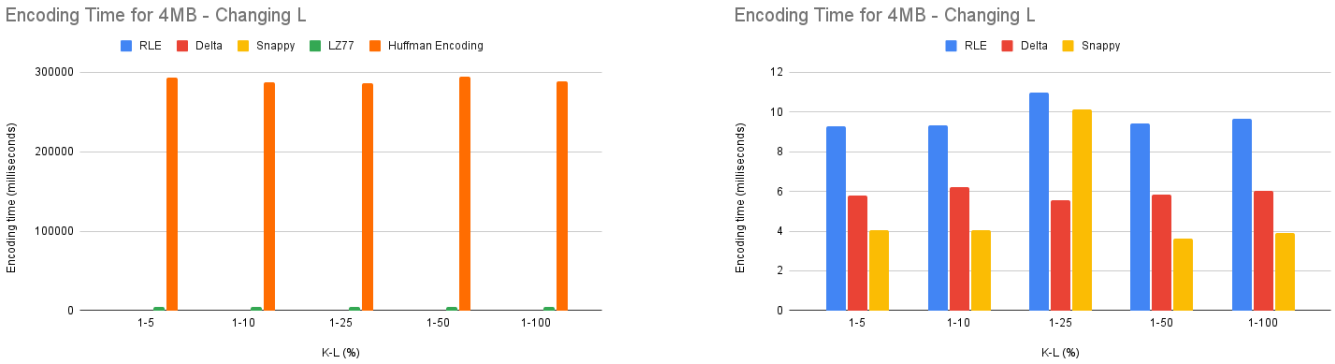

(a) All Algorithms




(b) Delta, RLE and Snappy Algorithm

Figure 7: Encoding Time for 400KB - Changing L

For workload of size 400 KB, the encoding time increases with increase in L because larger values of L increase the distance between the out-of-order elements, making it more difficult for compression algorithms to identify and exploit the patterns in the data. It can be seen that the increase in LZ77 and RLE algorithms is low but for delta, snappy, and Huffman we don't see a trend as their encoding time is independent of degree of sortedness.

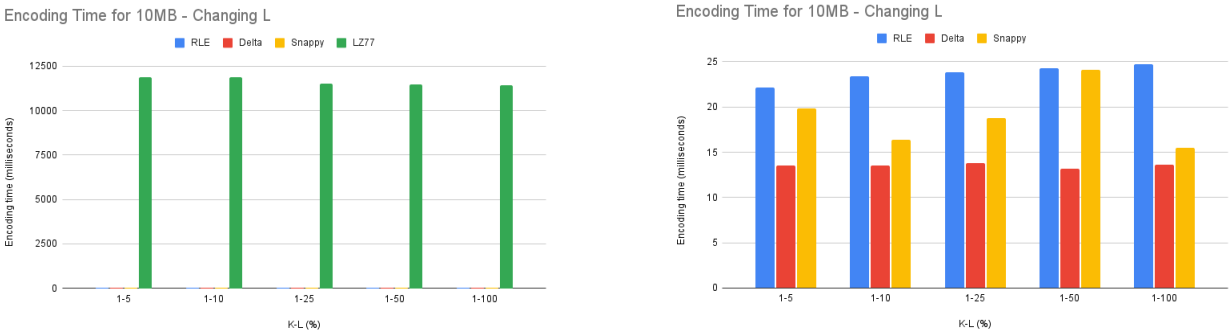



(a) All Algorithms

(b) Delta, RLE and Snappy Algorithm

Figure 8: Encoding Time for 4MB - Changing L

For workload of size 4MB, the trend is similar to that of 400KB. The encoding time increases slowly with increase in L for RLE and LZ77 and we don't see change in encoding time for delta, snappy, and Huffman as their encoding time is independent of degree of sortedness.



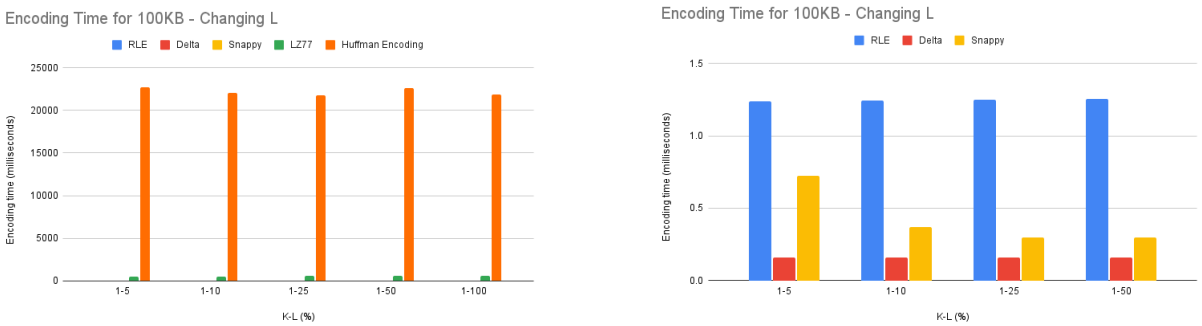

(a) All Algorithms

(b) Delta, RLE and Snappy Algorithm

Figure 9: Encoding Time for 10MB - Changing L

We see a similar trend for encoding time to that of 400KB and 4MB. The encoding time increases slowly with increase in L for RLE and LZ77 and there is no change in encoding time for delta and snappy. The Huffman Algorithm took more than 10 minutes to execute so we don't have results for that but we can speculate that Huffman will follow the same trend.

### 5.1.4 Changing L for 100% Unique Values in the Workload

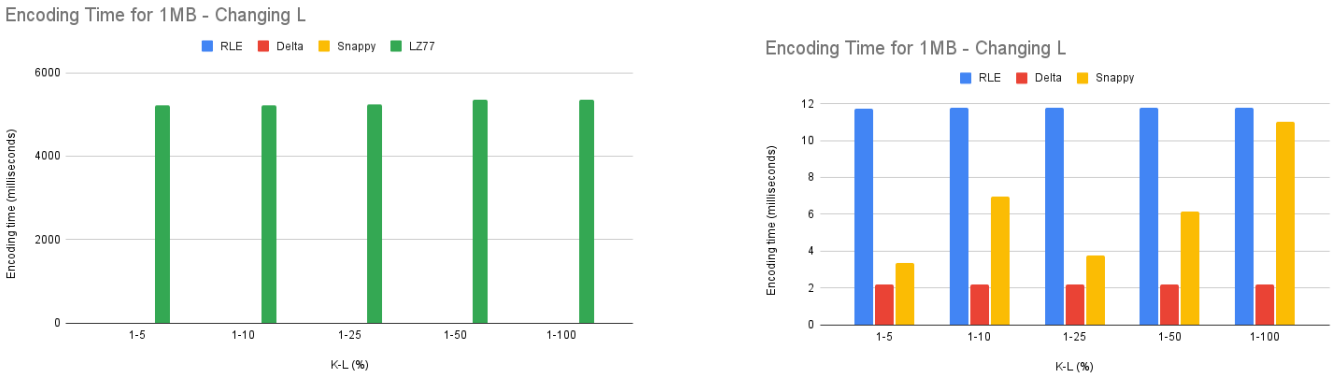



(a) All Algorithms

(b) Delta, RLE and Snappy Algorithm

Figure 10: Encoding Time for 100KB - Changing L

For a workload of size 100 KB, when L increases, the encoding time also increases due to the larger distance between out-of-order elements, which makes it more difficult for compression algorithms to identify patterns. While there is a slight increase in encoding time for RLE and LZ77, delta, snappy, and Huffman algorithms do not show any clear trend with respect to the degree of sortedness.

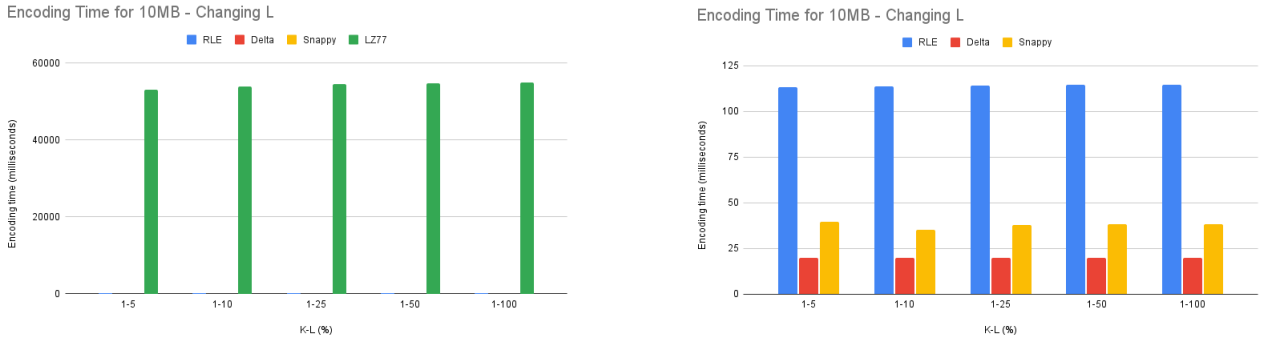



(a) All Algorithms

(b) Delta, RLE and Snappy Algorithm

Figure 11: Encoding Time for 1MB - Changing L

For a workload of size 1 MB, the trend is similar to that of 100 KB, with a slow increase in encoding time for RLE and LZ77 as L increases. There is no change in encoding time for delta, snappy, and Huffman as their encoding time is not affected by the degree of sortedness.



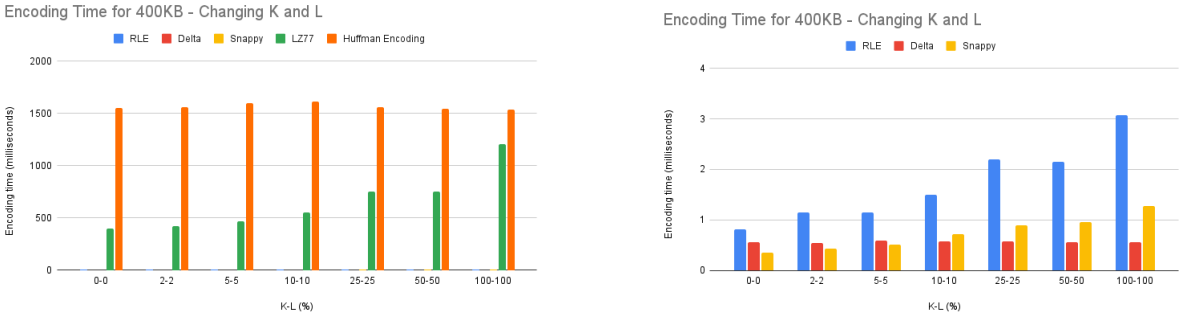

(a) All Algorithms

(b) Delta, RLE and Snappy Algorithm

Figure 12: Encoding Time for 10MB - Changing L

For a workload of size 10 MB, the trend is also similar to that of 100 KB and 1 MB. The encoding time increases slowly with increasing L for RLE and LZ77, while there is no change in encoding time for delta and snappy. The results for Huffman are not available due to the long execution time, but it is likely to follow the same trend as the other algorithms.

### 5.1.5 Changing L and K for 10% Unique Values in the Workload

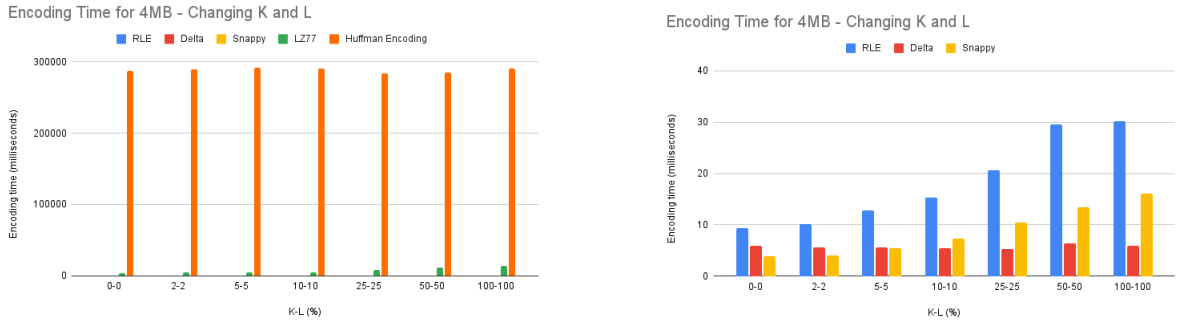



(a) All Algorithms

(b) Delta, RLE and Snappy Algorithm

Figure 13: Encoding Time for 400KB - Changing K and L

For workload size 400KB, as the value of K and L increases, the number of out-of-order elements in the workload and their maximum displacement also increase. This means that there are more complex patterns in the data that need to be analyzed and encoded. As a result, the compression algorithms take longer to encode the data, which leads to an increase in the encoding time drastically. For example, in the case of LZ77, as the value of K and L increases, the size of the sliding window also increases, which leads to more complex matching operations and longer encoding times. We can see that with increase in K and L values the encoding time for RLE and LZ77 increase drastically, while the encoding time for delta, snappy, and Huffman does not show any change as their encoding time is independent of degree of sortedness.

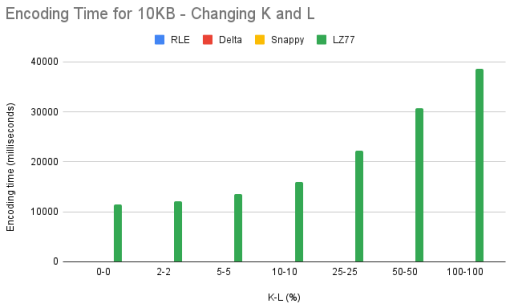



(a) All Algorithms

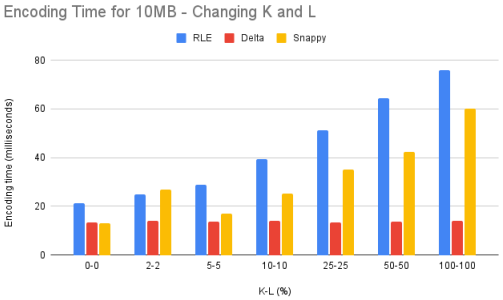
(b) Delta, RLE and Snappy Algorithm

Figure 14: Encoding Time for 4MB - Changing K and L

For workload of size 4MB, the trend is similar to that of 400KB. We see that with increase in K and L values the encoding time for RLE and LZ77 increase drastically, while the encoding time for delta, snappy, and Huffman does not show any change




(a) All Algorithms

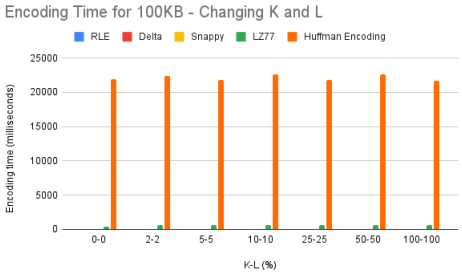



(b) Delta, RLE and Snappy Algorithm

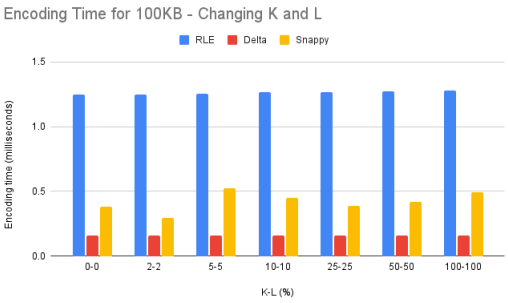
Figure 15: Encoding Time for 10MB - Changing K and L

For workload of size 10MB, the trend is similar to that of 400KB and 4MB. We see that with increase in K and L values the encoding time for RLE and LZ77 increase drastically, while the encoding time for delta, snappy, and Huffman does not show any change. The Huffman Algorithm took more than 10 minutes to execute so we don't have results for that but we can speculate that Huffman will follow the same trend.

### 5.1.6 Changing L and K for 100% Unique Values in the Workload



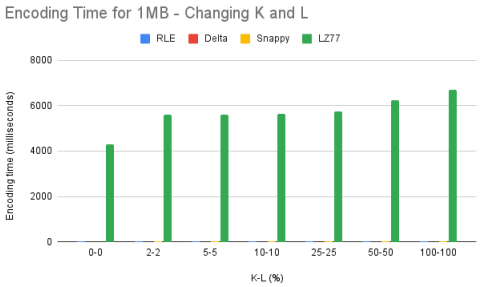

(a) All Algorithms



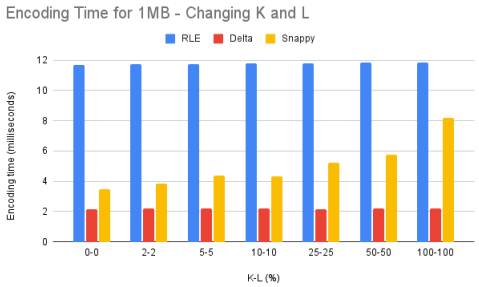

(b) Delta, RLE and Snappy Algorithm

Figure 16: Encoding Time for 100KB - Changing K and L

For a workload size of 100KB, when K and L values increase, the encoding time also increases because more complex patterns need to be analyzed and encoded. This results in a longer encoding time for compression algorithms like LZ77 and RLE, which have to perform more complex matching operations. On the other hand, the encoding time for delta, snappy, and Huffman algorithms remains constant regardless of the degree of sortedness.



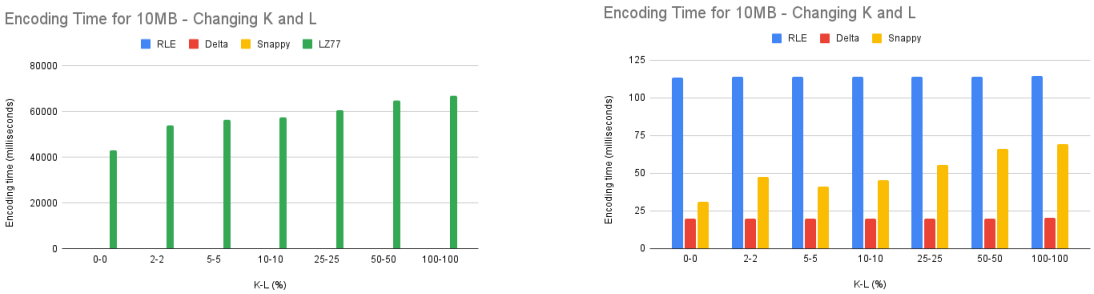

(a) All Algorithms




(b) Delta, RLE and Snappy Algorithm

Figure 17: Encoding Time for 1MB - Changing K and L

For a workload size of 1MB, the trend is similar to that of 100KB. The encoding time increases significantly with an increase in K and L values for LZ77 and RLE, while the encoding time for delta, snappy, and Huffman algorithms does not show any change.




(a) All Algorithms

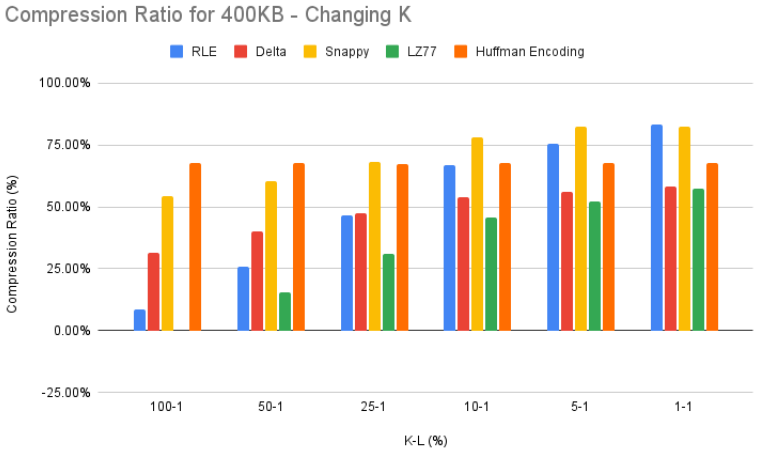
(b) Delta, RLE and Snappy Algorithm

Figure 18: Encoding Time for 10MB - Changing K and L

For a workload size of 10MB, we see the same trend as that of 100KB and 1MB. The encoding time increases drastically with an increase in K and L values for LZ77 and RLE, while the encoding time for delta, snappy, and Huffman algorithms does not show any change. We do not have results for Huffman Algorithm because it took more than 10 minutes to execute, but we can assume that it follows the same trend.

## 5.2 Compression Ratio vs Various K-L values

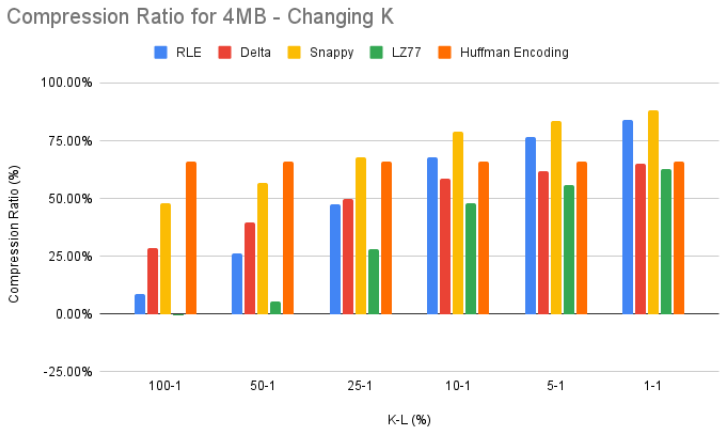
### 5.2.1 Changing K for 10% Unique Values in the Workload




(a) All Algorithms

Figure 19: Compression Ratio for 400KB - Changing K

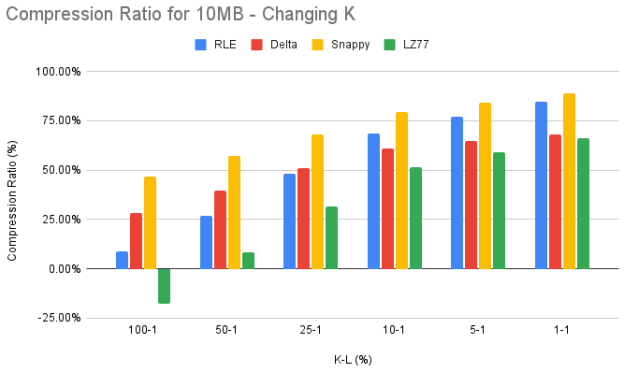
For workload of size 400 KB, as the value of K increases, the Compression Ratio generally decreases for all encoding methods. This is because increasing K means that there are more out-of-order elements in the workload, which makes it more difficult to compress. We can see that for algorithms RLE,delta, snappy and LZ77 the compression ratio increases with decrease in K value, but for Huffman we don't see any trend as their compression ratio is independent of degree of sortedness. However we see a negative compression ratio for LZ77 as when it does not find a pattern it still stores the data related to the pattern.




(a) All Algorithms

Figure 20: Compression Ratio for 4MB - Changing K

For workload of size 4MB, we see a similar trend to that of 400KB. For algorithms RLE,delta, snappy and LZ77 the compression ratio increases with decrease in K value, but for Huffman we don't see any change in its compression ratio.

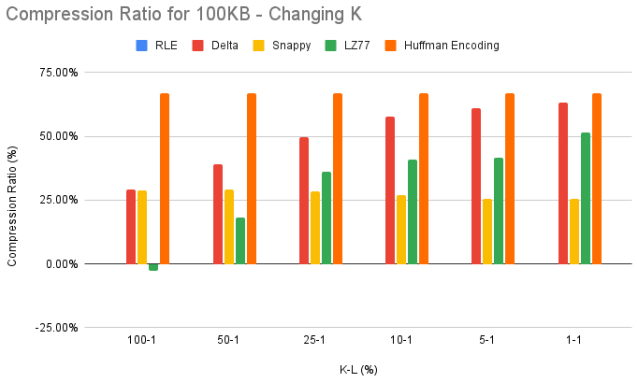



(a) All Algorithms

Figure 21: Compression Ratio for 10MB - Changing K

For workload of size 10MB, the trend is similar to that of 400KB and 4MB. We see that for RLE, delta, snappy and LZ77 the compression ratio increases with decrease in K value, but for Huffman we don't see any change in its compression ratio. The Huffman Algorithm took more than 10 minutes to execute so we don't have results for that but we can speculate that Huffman will follow the same trend.

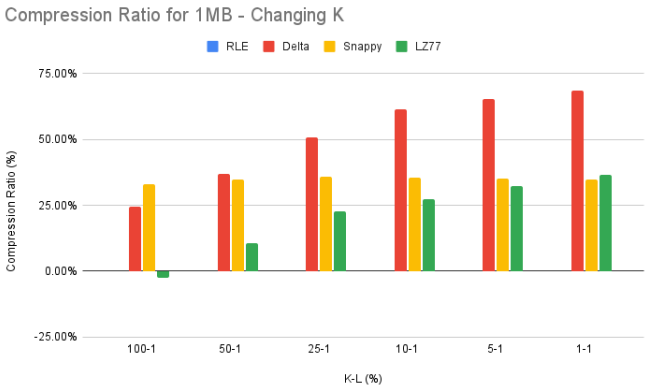
### 5.2.2 Changing K for 100% Unique Values in the Workload




(a) All Algorithms

Figure 22: Compression Ratio for 100KB - Changing K

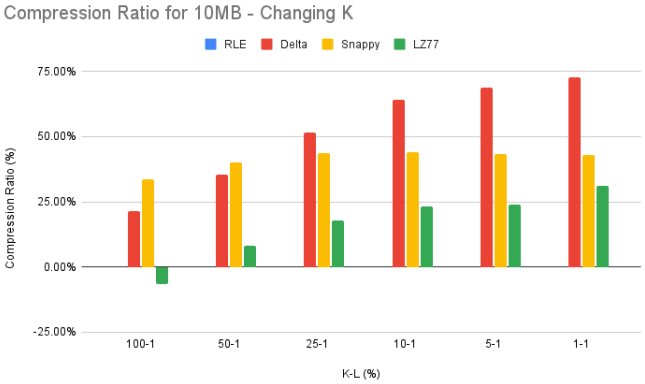
For the workload of size 100 KB, when the value of K increases, the number of out-of-order elements also increases, which makes it harder to compress the data. As a result, the Compression Ratio decreases for all encoding methods. However, for RLE, delta, snappy, and LZ77 algorithms, we see that the compression ratio increases as we decrease the value of K. For LZ77, we also observe negative compression ratios as it stores the data related to patterns even when it doesn't find a match. On the other hand, we don't see any trend for Huffman as its compression ratio is independent of the degree of sortedness.




(a) All Algorithms

Figure 23: Compression Ratio for 1MB - Changing K

For workload of size 1MB, we see a similar trend to that of 100KB. The Compression Ratio generally decreases for all encoding methods as the value of K increases, but for RLE, delta, snappy, and LZ77 algorithms, the compression ratio increases with a decrease in K value. However, we don't see any change in the compression ratio for Huffman.

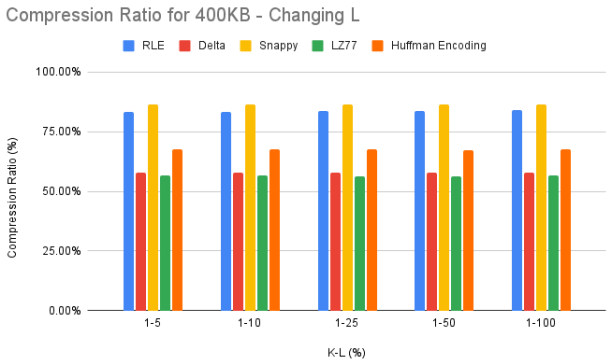



(a) All Algorithms

Figure 24: Compression Ratio for 10MB - Changing K

For workload of size 10MB, the trend is similar to that of 100KB and 1MB. For RLE, delta, snappy, and LZ77 algorithms, we observe that the compression ratio increases with a decrease in K value, but for Huffman, there is no change in the compression ratio. We don't have results for Huffman as its execution time was more than 10 minutes, but we can assume that it will follow the same trend.

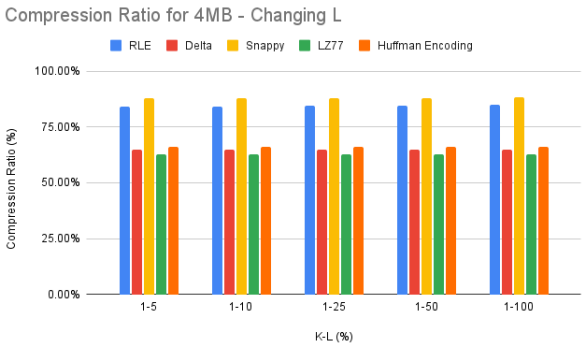
### 5.2.3 Changing L for 10% Unique Values in the Workload




(a) All Algorithms

Figure 25: Compression Ratio for 400KB - Changing L

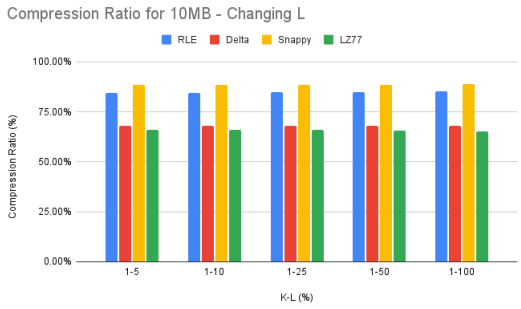
For workload of size 400 KB, as the value of L increases, the Compression Ratio generally increases for all encoding methods but by very small value. This is because increasing L means that we are increasing maximum displacement in the workload. We can see that for algorithms RLE,delta, snappy and LZ77 the compression ratio increases with increase in L value, but for Huffman we don't see any trend as their compression ratio is independent of degree of sortedness.




(a) All Algorithms

Figure 26: Compression Ratio for 4MB - Changing L

For workload of size 4MB, we see a similar trend to that of 400KB. For algorithms RLE,delta, snappy and LZ77 the compression ratio increases with increase in L value, but for Huffman we don't see any change in its compression ratio.

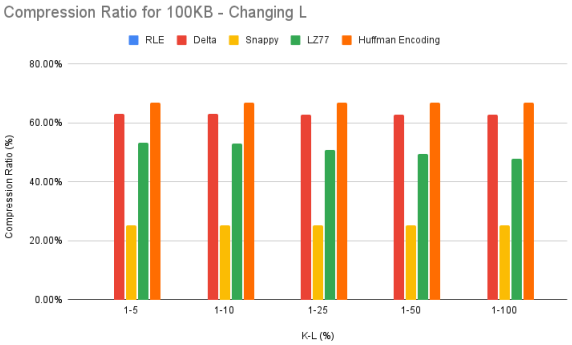



(a) All Algorithms

Figure 27: Compression Ratio for 10MB - Changing L

For workload of size 10MB, the trend is similar to that of 400KB and 4MB. We see that for RLE, delta, snappy and LZ77 the compression ratio increases with increase in L value, but for Huffman we don't see any change in its compression ratio. The Huffman Algorithm took more than 10 minutes to execute so we don't have results for that but we can speculate that Huffman will follow the same trend.

### 5.2.4 Changing L for 100% Unique Values in the Workload

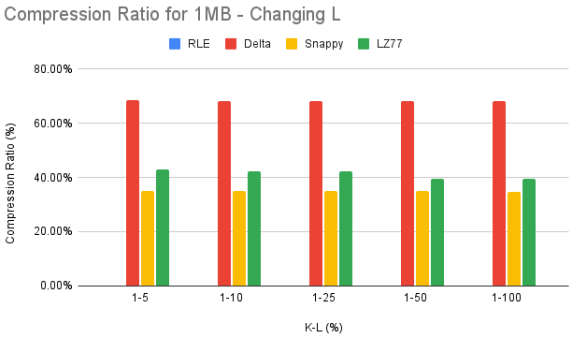



(a) All Algorithms

Figure 28: Compression Ratio for 100KB - Changing L

For a workload size of 100 KB, increasing the value of L leads to a slight increase in the compression ratio for all encoding methods. This is because as L increases, the maximum displacement in the workload increases, allowing for

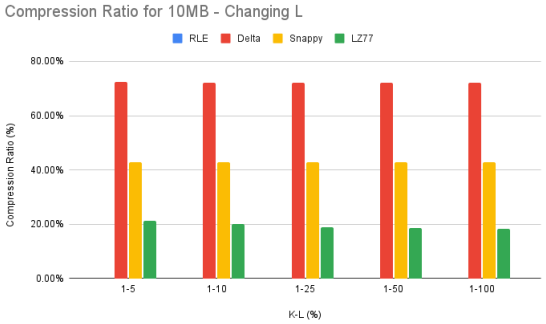
more opportunities for compression. However, the increase in compression ratio is very small. The compression ratios of RLE, delta, snappy and LZ77 algorithms generally increase with an increase in L value, while the compression ratio of Huffman remains independent of the degree of sortedness.




(a) All Algorithms

Figure 29: Compression Ratio for 1MB - Changing L

For a workload size of 1MB, the trend is similar to that of 100KB. The compression ratios of RLE, delta, snappy and LZ77 algorithms generally increase with an increase in L value, but the compression ratio of Huffman does not show any change.

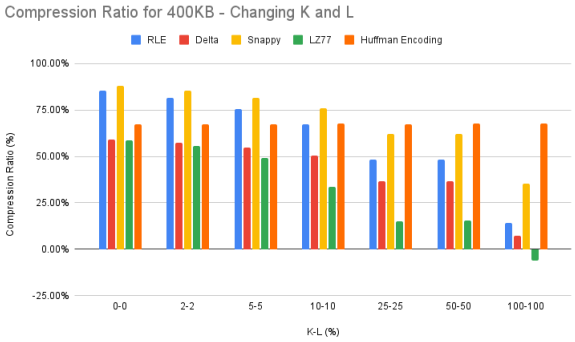



(a) All Algorithms

Figure 30: Compression Ratio for 10MB - Changing L

For a workload size of 10MB, the trend is similar to that of 100KB and 1MB. The compression ratios of RLE, delta, snappy and LZ77 algorithms generally increase with an increase in L value, but the compression ratio of Huffman does not show any change. However, we don't have the results for the Huffman algorithm due to its longer execution time, but we can speculate that it will follow the same trend.

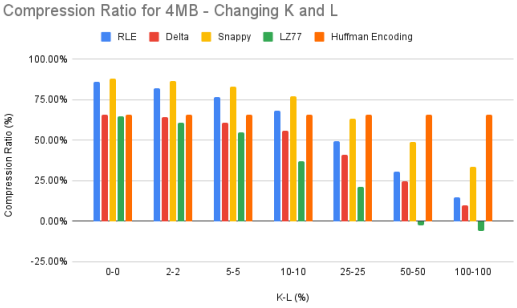
### 5.2.5 Changing L and K for 10% Unique Values in the Workload




(a) All Algorithms

Figure 31: Compression Ratio for 400KB - Changing K and L

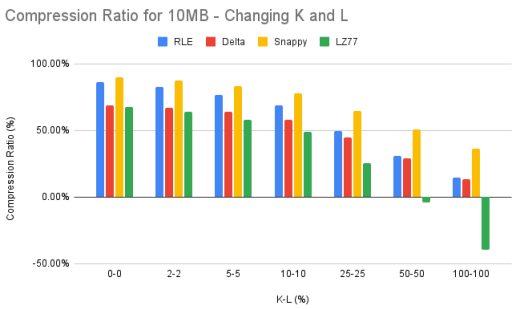
For workload of size 400 KB, as the value of K and L increases, the data in the workload becomes more out of order, and the amount of redundancy between adjacent elements decreases. This makes it more difficult for compression algorithms to identify and exploit similarities between adjacent elements in the data. Therefore, the compression ratio generally decreases as K and L increase. We see that for algorithms RLE,delta, snappy and LZ77 the compression ratio decreases drastically with increase in K value, but for Huffman we don't see any trend as their compression ratio is independent of degree of sortedness. However we see a negative compression ratio for LZ77 as when it does not find a pattern it still stores the data related to the pattern.




(a) All Algorithms

Figure 32: Compression Ratio for 4MB - Changing K and L

For workload of size 4MB, we see a similar trend to that of 400KB. For algorithms RLE,delta, snappy and LZ77 the compression ratio decreases drastically with increase in K value, but for Huffman we don't see any change in its compression ratio.

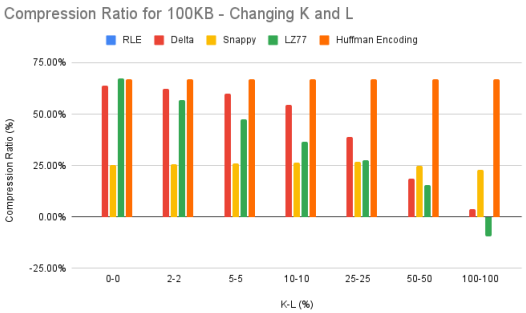



(a) All Algorithms

Figure 33: Compression Ratio for 10MB - Changing K and L

For workload of size 10MB, the trend is similar to that of 400KB and 4MB. We see that for RLE, delta, snappy and LZ77 the compression ratio decreases drastically with increase in K value, but for Huffman we don't see any change in its compression ratio. The Huffman Algorithm took more than 10 minutes to execute so we don't have results for that but we can speculate that Huffman will follow the same trend.

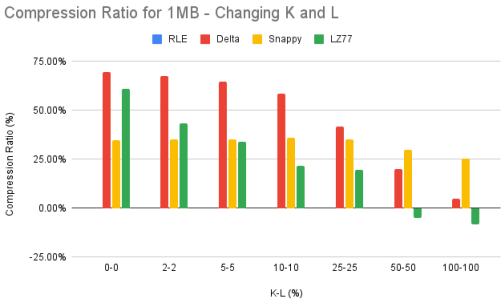
### 5.2.6 Changing L and K for 100% Unique Values in the Workload




(a) All Algorithms

Figure 34: Compression Ratio for 100KB - Changing K and L

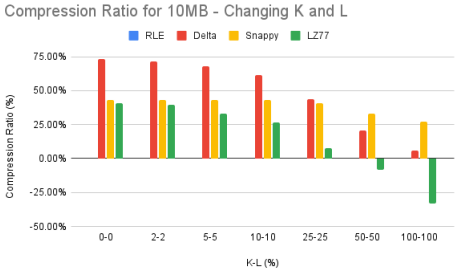
For a workload of 100 KB, when the values of K and L increase, the data becomes more disordered and the redundancy between adjacent elements decreases. As a result, compression algorithms have a harder time identifying and exploiting similarities between adjacent elements, leading to a general decrease in compression ratio. Among the algorithms tested, RLE, delta, snappy, and LZ77 show a significant decrease in compression ratio with increasing K value, while Huffman's compression ratio remains independent of the degree of sortedness. LZ77, however, shows a negative compression ratio when it fails to find a pattern and stores the data related to the pattern.




(a) All Algorithms

Figure 35: Compression Ratio for 1MB - Changing K and L

For a workload of 1 MB, we observe a similar trend to that of 100 KB. RLE, delta, snappy, and LZ77 algorithms experience a significant decrease in compression ratio with increasing K value, while Huffman's compression ratio remains unchanged.



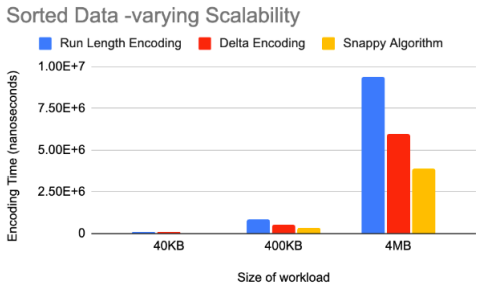

(a) All Algorithms

Figure 36: Compression Ratio for 10MB - Changing K and L

For a workload of 10 MB, the trend is the same as for 100 KB and 1 MB. RLE, delta, snappy, and LZ77 algorithms show a significant decrease in compression ratio with increasing K value, while Huffman's compression ratio remains unchanged. The Huffman Algorithm took more than 10 minutes to execute, so there are no results for it, but it is possible that it will follow the same trend.

## 5.3 Scalability of Algorithms

### 5.3.1 For Sorted Data

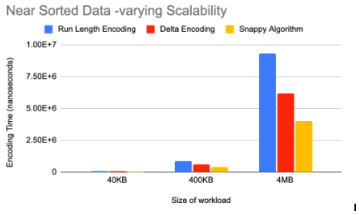



(a) All Algorithms

Figure 37: Encoding Time for Sorted Data

For sorted data (0-0) - We can see from the above graph as we increase the workload size the encoding time increases drastically for all the algorithms specially for RLE. As we increase the workload size, the amount of data that needs to be processed by the encoding algorithms also increases. This means that the encoding algorithms need to perform more operations on larger amounts of data, which takes more time.

### 5.3.2 For Near-Sorted Data

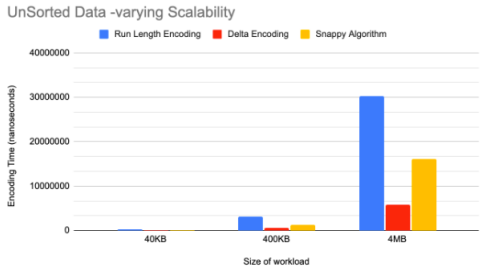



(a) All Algorithms

Figure 38: Encoding Time for Near Sorted Data

Similarly for near sorted data (1-10) we can see from the above graph as we increase the workload size the encoding time increases drastically for all the algorithms specially for RLE.

### 5.3.3 For Unsorted Data



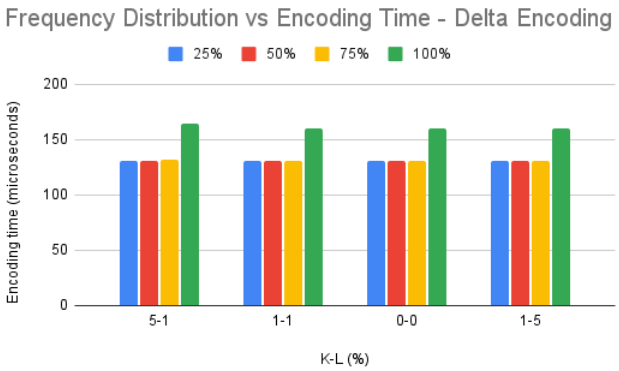

(a) All Algorithms

Figure 39: Encoding Time for Unsorted Data

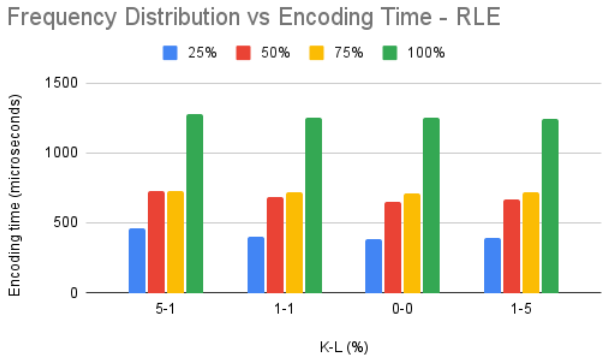
For unsorted data (100-100) - We can see from the above graph as we increase the workload size the encoding time increases drastically for all the algorithms specially for RLE.

## 5.4 Percentage of Unique Values

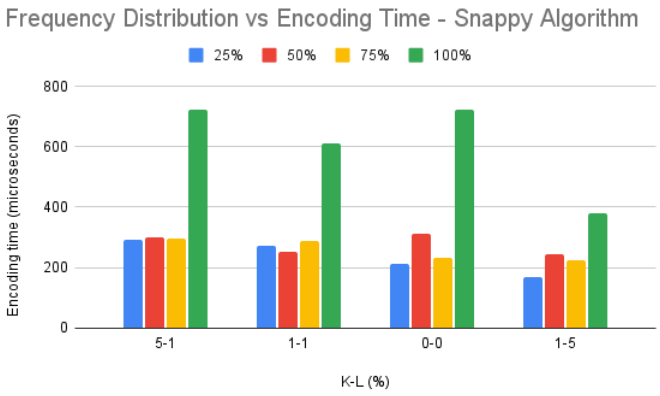
### 5.4.1 Percentage of Unique Values vs Encoding Time




(a) Delta Encoding




(b) Run Length Encoding




(c) Snappy Algorithm

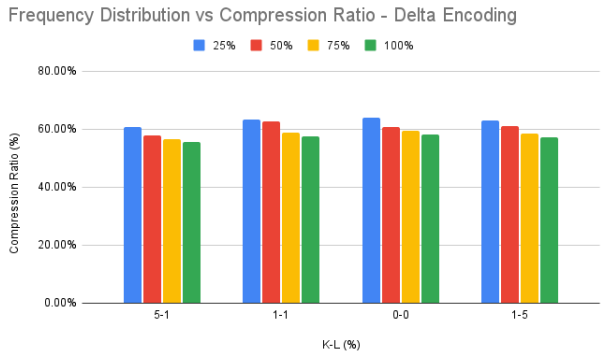
Figure 40: Percentage of Unique Values vs Encoding Time

From the above graph, we can see that for Delta Encoding the Encoding time is almost the same for all the frequency distributions. Although, we can see a spike for 100% unique data which could be because the encoded values are large and therefore take longer to compute.

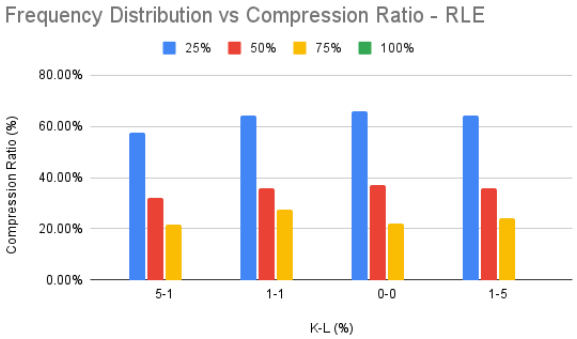
For Run Length Encoding, we can see that the encoding time is increasing almost exponentially as the percentage of unique values is increasing for all types of sorted and near-sorted workloads. This is happening because as the number of unique values are increasing the possibility of having consecutive duplicates decreases and hence it takes more time to encode the data.

In Snappy algorithm, we can see that it somewhat follows the trend of Delta Encoding where the encoding time is similar for 25%, 50% and 75% unique values and increases drastically for 100% unique values, which could be happening because of the additional computational time for encoding larger values.

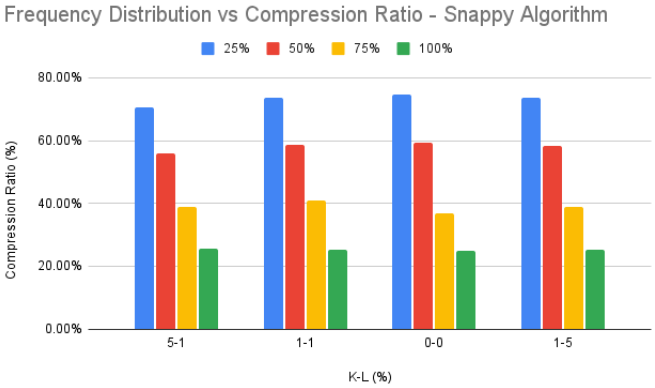
### 5.4.2 Percentage of Unique Values vs Compression Ratio




(a) Delta Encoding




(b) Run Length Encoding




(c) Snappy Algorithm

Figure 41: Percentage of Unique Values vs Compression Ratio

The first graph above for Delta Encoding shows that the compression ratio decreases as the percentage of unique values increases for all combinations of K-L values. This is happening because the repeating values lead to smaller differences between consecutive data points which lead to a higher compression ratio.

The second graph shows the compression ratio for Run Length Encoding and we can see that it follows a similar trend as delta encoding but here the compression ratio is decreasing more drastically which is expected as Run Length Encoding is based on finding the count of consecutive duplicate values. We can also note that the compression ratio for 100% unique values is zero as there are no duplicate values and nothing to compress.

The third graph shows the compression ratio for Snappy Algorithm. We can see that this graph also follows the same trend as Run Length Encoding and Delta Encoding.

# 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 some other algorithms. 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*