CS561 : Data Systems Architectures - Benchmark Compression With Near Sortedness

Harshitha Tumkur Kailasa Murthy
Boston University

Vishwas Bhaktavatsala
Boston University

ABSTRACT

This research is based on bench-marking of various Compression Algorithms. It provides a study on how different Compression Algorithms behaves with varied sortedness. Compression is a crucial process for storing and transmitting digital files efficiently. It is the process of reducing the size of a file by removing redundant or unwanted data. Compression can be lossless or lossy, depending on whether or not any data is lost during the compression process. Lossless compression maintains the quality of the original file, while lossy compression sacrifices some quality to further reduce the file size. The technique of data compression can save storage capacity, speed up file transfer, and decrease costs for storage hardware and network bandwidth. The Compression techniques enables sending a data object or file quickly over a network or the Internet and in optimizing physical storage resources. In this paper, we present Benchmark on Data Compression with near Sortedness by experimenting different compression algorithms with various sorting patterns to see the performance.

1 INTRODUCTION

In today's digital era everyone are connected by using handheld devices such as mobile phones and tablets. Additionally, this has increased data sharing, which necessitates effective disk storage management. Furthermore, compact file sizes are needed for quick and efficient data transfer over the internet. Data can be compressed using lossy and lossless compression algorithms provided by compression file formats. These facilitate efficient data transfer over the internet and lower disk storage use.

1.1 Motivation

Each day peta bytes of data is been collected in the world. We do not have right storage infrastructure to store these as it is. This is where we feel Data Compression plays a vital role to compress these crucial data and store it. Modern applications, in a variety of domains, have become data-intensive. In fact, the Vs [1] broadly characterize the big data as follows: volume, velocity, variety, and veracity. Each one of those Vs dictates a growing need to manage, if not control, the explosion of data growth. Applications use data compression techniques in a range of workloads in order to reduce the data footprint. As a motivating illustration, NASA employs sentiment analysis [3] to spot human trafficking through online information. A web crawler collects various textual (like xml, json, and csv) and visual (like images and videos) data from the web and feeds the information to the primary sentiment model, which then generates a set of intermediate data outlining the relationships between relevancy and sentiment. The output analyzer finally combines the intermediate data to create the end result. There may be multiple compression priority in such complex operations. Compression speed, decompression speed, and compression ratio are the three basic measures used to describe data compression. In each category, the performance of each compression algorithm varies.

1.2 Problem Statement

Data Compression algorithms have varying performance based on the sortedness of the data. Finding and analyzing the available algorithms for data compression can help choose the right algorithm when required. The main factor which seems to be affecting these algorithms is the initial sortedness of the input stream of data.

1.3 Contributions

We looked at various papers, few of which we have used as references for this report. These papers on data compression helped us understand the whole concept along with important aspects which must be considered for a data compression algorithm. We tried to reference various algorithms for different kinds of inputs.

Currently, our concentration was on Run Length Encoding and how it would work with integer workloads. The research helped us with the implementation for this algorithm and helped us with our initial design and solutions.

1.4 Types of Compression

Two fundamental forms of data compression are now used in various contexts. One of these, known as lossy data compression, is frequently used to reduce the size of image data files for transmission or archiving. The other is lossless data compression, which is frequently employed to transport or archive text or binary files that must maintain the integrity of their information at all times.

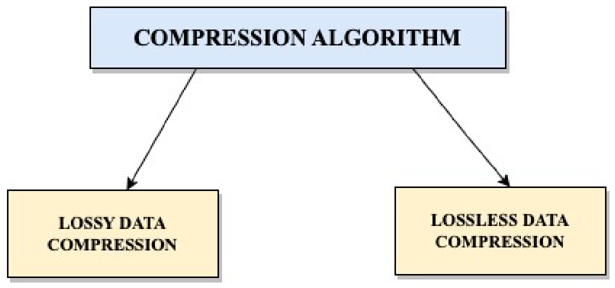

Figure 1: Example of Run Length Encoding Compression

2 BACKGROUND

The aim of data compression is to reduce redundancy in stored or communicated data, thus increasing effective data density. Data compression has important application in the areas of file storage and distributed systems. This paper surveys a variety of data compression [2] Algorithm spanning almost 40 years of research, from the work of Shannon, Fano, and Huffman in the late 1940s to a technique developed in 1986. The aim of data compression is to reduce redundancy in stored or communicated data, increasing effective data density. Data compression has important application in the areas of file storage and distributed systems. Concepts from information theory as they relate to the goals and evaluation of data compression methods are discussed briefly. A framework for evaluation and comparison of methods is constructed and applied to the algorithms presented. Comparisons of both theoretical and empirical natures are reported, and possibilities for future research are suggested.

2.1 Run-Length Encoding

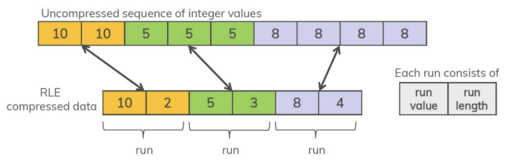

Figure 2: Example of Run Length Encoding Compression

Fig. 1 provides an illustration of Run-Length Encoding (RLE), a popular lightweight compression technique. [4] In the compressed output format, a run value and a run length serve as a lossless representation of continuous runs of instances of the same integer value in the input data. It can be seen that a compression at run level is attained starting at run length 3, and the compression rate increases as run length increases. So long as the average run duration across all runs is larger than 2, RLE will result in a data size reduction. In this example, there is a variance of 1 and an average run length of 3. Hence, the possible compression rate is mostly influenced by these data qualities. The implementation of these CD instructions to accelerate the Run-Length Encoding compression of sequences of integers with short run lengths was suggested in this study.

Algorithm 1: Run-length Encoding

```
Input  : An integer array $arr$
Output : A vector of pairs representing the run-length
         encoding of $arr$
$start\_time \leftarrow$ current time
$n \leftarrow$ size of $arr$
$res \leftarrow$ empty vector of pairs
for $i \leftarrow 0$ to $n - 1$ do
    $count \leftarrow 1$
    while $i < n - 1$ and $arr[i] == arr[i + 1]$ do
        $count \leftarrow count + 1$
        $i \leftarrow i + 1$
    end
    Add the pair $(arr[i], count)$ to $res$
end
return $res$
```

2.2 ZStandard Algorithm

Zstandard is a fast and Lossless Compression. High compression ratios are offered by this fast compression algorithm. It also provides dictionary compression, a specific mode for small data. It is a brand-new compression algorithm and implementation made to work with cutting-edge hardware and compress data smaller and more efficiently. Zstandard combines recent compression breakthroughs, like Finite State Entropy, with a performance-first design — and then optimizes the implementation for the unique properties of modern CPUs. several fast compression algorithms were tested and compared on a server running Arch Linux, as per the experiments there were drastic compromises. However Zstandard showed substantial improvement in Compression Speed and Decompression speed while maintaining a high Compression ratio. The trade-offs made by previous compression algorithms are thereby improved, and it has a broad variety of applications and a very high decompression speed.

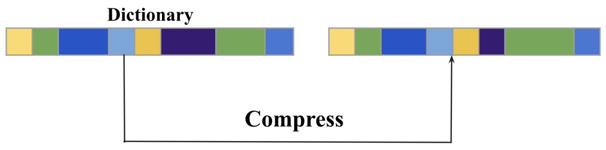

Figure 3: Example of Run Length Encoding Compression

2.3 Delta Algorithm

Delta-Encoding also known as Delta Compression works by reducing the amount of information in the data file or objects by saving only the difference(or delta) between the object and one or more reference objects.

Fig 4 shows the representation of delta encoding compression. The original data has a set of random numbers starting from 56030 to 56007. Once the data goes through delta compression, we can see in the representation that it has save the first number as a starting block and difference from that number till the end in the rest of the blocks.

Delta encoding performs best when data has small or constant variation; for an unsorted data set, there may be little to no compression possible with this method.

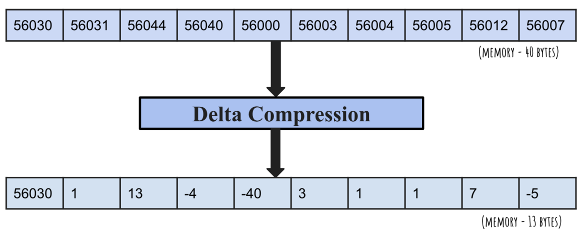

Figure 4: Example of Delta Encoding Compression

Algorithm 2: Delta Encoding

```
Result: Data with delta encoding
$start\_time \leftarrow$ current time
for $i$ in range $(|data| - 1), 0, -1$ do
    $data[i] \leftarrow data[i] - data[i-1]$
end
```

2.4 Snappy Algorithm

Snappy, previously known as Zippy is Google's fast compression and decompression library developed in C++. This algorithm is based on the concepts and ideas from LZ77 Algorithm. It is designed to prioritize speed and reasonable compression rather than maximum compression or compatibility with other compression libraries. Its usage is prevalent in various Google projects, including Bigtable, MapReduce, and compressing data for internal RPC systems. Moreover, it has gained traction in open-source projects such as Cassandra, Couchbase, Hadoop, LevelDB, MongoDB, RocksDB, Lucene, Spark, InfluxDB, and MariaDB ColumnStore. Additionally, decompression is thoroughly tested to identify errors in the compressed stream. Snappy strives for portability and does not use inline assembler, except for specific optimizations.

Snappy compression utilizes a sliding-window compression algorithm that incorporates a dictionary within the data itself. It achieves this by referencing repeated data sequences instead of storing them separately from the compressed data stream.[7]

The Snappy compression algorithm processes input data by breaking it into chunks and analyzing each chunk independently. The algorithm searches for repeated byte sequences in each chunk and encodes them as back-references to their previous occurrence in a sliding window. If no matches are found, the algorithm writes the literal data to the output stream.

To enhance compression speed, Snappy employs a simple entropy coding scheme that eliminates the need for a separate compression dictionary. Instead, the algorithm keeps a small buffer of recently compressed data to use as a local dictionary for future compression.

During decompression, the algorithm reads in the compressed data stream and reverses the process by reconstructing the original data from back-referenced repeated byte sequences. Snappy also includes a verification step that compares the decompressed data to a checksum stored in the compressed data stream to ensure its integrity. Overall, Snappy's design prioritizes compression and decompression speed over maximizing compression ratios.

2.5 Lempel Ziv Algorithm

LZ77 is a lossless data compression algorithm that works by identifying repeated sequences of data in the input stream and replacing them with back-references to their previous occurrences. The algorithm consists of two main steps:

Step 1 Sliding window search: The LZ77 algorithm maintains a sliding window over the input data stream, which contains the most recently seen data. It then searches for repeating sequences of data within this window. This search is typically performed using a hashing or suffix tree-based algorithm to efficiently identify repeating patterns.

Step 2 Back-reference encoding: When a repeating sequence of data is found, the LZ77 algorithm encodes it as a pair of values: a distance and a length. The distance value specifies the offset between the current position in the input stream and the previous occurrence of the repeated data, while the length value specifies the length of the repeated data. These values are then encoded using a variable-length code, such as Huffman coding, to represent them more efficiently in the compressed output stream.

Algorithm 3: LZ77 Compression

```
Input  : An integer array $input$
Output : A vector of triples representing the LZ77
         compression of $input$
$inputLength \leftarrow$ length of $input$; $output \leftarrow$ empty vector of
  triples; $pos \leftarrow 0$;
while $pos < inputLength$ do
    $maxMatchOffset \leftarrow -1$; $maxMatchLength \leftarrow -1$; for
     $i \leftarrow 1$ to $pos$ do
        $matchLength \leftarrow 0$; while
         $pos + matchLength < inputLength$ and
         $input[pos + matchLength] =$
         $input[pos - i + matchLength]$ do
            $matchLength \leftarrow matchLength + 1$;
        end
        if $matchLength > maxMatchLength$ then
            $maxMatchLength \leftarrow matchLength$;
             $maxMatchOffset \leftarrow i$;
        end
    end
    if $maxMatchLength > 0$ then
        $token.distance \leftarrow maxMatchOffset$;
         $token.length \leftarrow maxMatchLength$;
         $token.next_char \leftarrow input[pos + maxMatchLength]$;
         Add the triple
         $(maxMatchOffset, maxMatchLength, input[pos+$
         $maxMatchLength])$ to $output$;
         $pos \leftarrow pos + maxMatchLength + 1$;
    end
    else
        $token.distance \leftarrow 0$; $token.length \leftarrow 0$;
         $token.next_char \leftarrow input[pos]$; Add the triple
         $(0, 0, input[pos])$ to $output$; $pos \leftarrow pos + 1$;
    end
end
return $output$;
```

Figure 5 shows us an example of how LZ77 Algorithm Compresses an integer workload which has repeating characters. Here in this example, using the sliding window technique, with window size of 8 and a maximum match length of 4, The sliding window would look like the following '15 16 23 | 15 16 23 42'. To compress this sequence using LZ77, it would start by looking for repeating sequences of integers within the sliding window. In this case, the sequence "4 8 15 16 23 42" repeats twice. The first six integers are

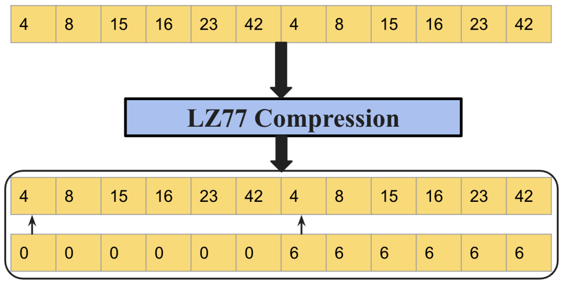

Figure 5: Example of LZ77 Encoding Compression

not encoded as back-references because they do not match any previous sequence within the sliding window.

3 ARCHITECTURE

This paper will also concentrate on bench-marking these compression algorithms with varying data workloads. The variations in the input data workloads would be depending on the level of data sortedness in the workloads. This will be achieved using a workload generator which will be able to choose a sortedness metric and generate a workload. We will be creating an API framework which can take different varying workloads and choose a data compression algorithm and run the benchmark and provide the metrics

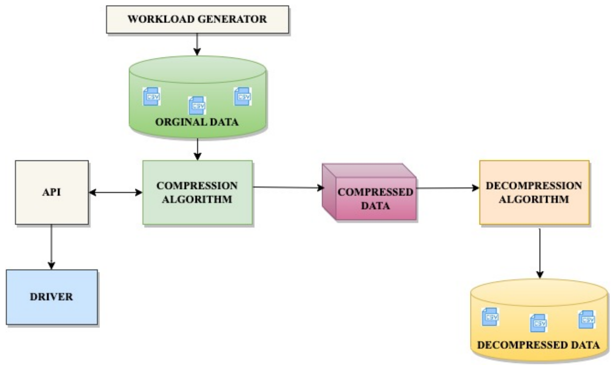

Figure 6: Initial Architectural Design

3.1 Solution Design

The initial plan is to create a framework which can house the workload generator as well as all the APIs for Data Compression. Workload Generator, which would be provided by the mentor has the ability to generate workloads. Currently, we have tried to create a very simple workload generator which would generate basic workloads based on input parameters which would be size and sortedness metric. In the next steps of the algorithm, we would be incorporating the official workload generator with our API Framework.

Our complete framework contains the required structure and methods to generate the workload based on specific parameters. The size of workload can differ based on the requirement from hundred thousands to tens of million values. Once the workload is generated and added to a file, the path of the file would be passed to an API. The API, which contains the main bench-marking code would take the workload with the required parameters for the algorithms. The required implementation would be called upon with the workload and the timer would be started. Once our implementation returns a result, our code would calculate the elapsed time and log them.

3.2 Workload Generator

For the experiments to benchmark the different algorithms we used various workloads which had different features. These workloads were designed to follow a certain distributions, frequencies, varying sortedness. To generate these workloads we created a python script which would create these workloads with a constant size of 10 million values which amounts up-to a 40Mb data file. This Workload generator uses these constants to create a different types of workloads.

Each workload has *n* unique numbers which are split into sorted runs of random frequencies. In our case the entries are in between 1 and 25. For the first three workloads we use, the frequencies follow a different distributions. We have created a workload which had its frequencies which are in the uniform distribution format, one which had a normal distribution and finally one with a U Quadratic Distribution. We can see the graphs for these distributions in Figure 7. We have also generated workloads which has random frequencies for all the sorted runs along with workloads which vary the sortedness by performing local and global disorder of the sorted runs. Additionally, to see the performance of these algorithm with varying workload size, we have generated four workloads which vary in size from 400 Kb to 400 Mb.

To generate all these mentioned workloads, we created a python script which would generate these workload dynamically based on the input parameters. For the sake of this research and experiments, we have set a size of 25 unique values which range over 10 million units of data. This script is designed to generate 25 frequency blocks which follows the distributions which we have mentioned previously.

4 COMPARATIVE ANALYSIS

Running Time/Latency The running time of an algorithm refers to the length of time it takes for it to run as a function. An algorithm's running time for a given input depends on the number of operations executed. An algorithm with more operations will have a lengthier running time.[9]

$$\text{Running Time or Latency} = \text{End Timestamp - Start Timestamp} \tag{1}$$

Compression ratio: the ratio of the original size (numerator) to the compressed size (denominator), expressed in unitless data as a value of at least 1.0.

$$\text{Compression Ratio} = \frac{\text{Size after compression}}{\text{Size before compression}} \text{ x } 100 \tag{2}$$

Accuracy: the ratio of the number of characters which are matching in the decompressed file and the original data file to the total number of characters in the original file.

$$\text{Accuracy} = \frac{\text{Matching characters}}{\text{Total characters}} \text{ x } 100 \tag{3}$$

where "Matching characters" refers to the number of characters in the decompressed file that match those in the original file, and "Total characters" refers to the total number of characters in the original file.

5 EVALUATIONS AND RESULTS

To run the benchmark tests on the algorithms for further analysis of their performance against various workloads, We used the previously explained workload generator. We ran multiple experiments by keeping three parameters into consideration Frequency of data, Latency and File sizes. We have created workloads which

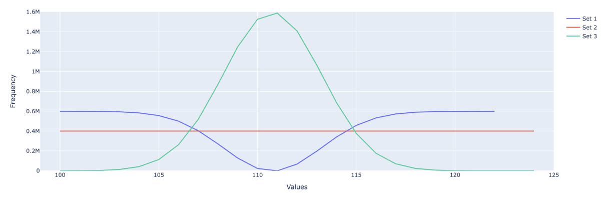

Figure 7: Distribution Graph - Normal, Uniform and U-Quadratic

Figure 7 shows the different distribution for the sorted runs in the workloads which we have created. In Fig 7, the red line shows the Uniform Distribution where the size of each sorted run blocks are the same. The green line depicts the Normal Distribution where we can see that the size of the sorted runs starts with a minimal value, reaching to the peak of the distribution and then falling back to the minimum. The blue line illustrates the distribution which is the exact opposite of the Normal distribution.

The three types of workloads which followed different distributions, had a very similar running time with all the algorithms in focus. Figure 8 clearly shows the difference in the execution times for the different algorithms. ZStandard and LZ77 is observed to take approximately 500 and 300 ms respectively which is comparatively higher when compared to the other algorithms. Run Length Encoding, Delta Compression and Snappy have the best execution times overall. When compared individually, Uniformly distributed workloads seems to take the most execution times when compared to the rest. This can be attributed to how the frequencies in this kind of distribution, since all the items have equal frequencies it takes more time to be compressed. While the other types of distributions have very few items which have high frequencies and this is reflecting in the running times which is lower than the previous distributions.

For further analysis on the performance of algorithms, we ran experiments on workloads which had sorted runs with random lengths i.e frequencies. Additionally, we performed local and global disorder on the original workload. This can help us understand how much the order of the items affect the execution times for all

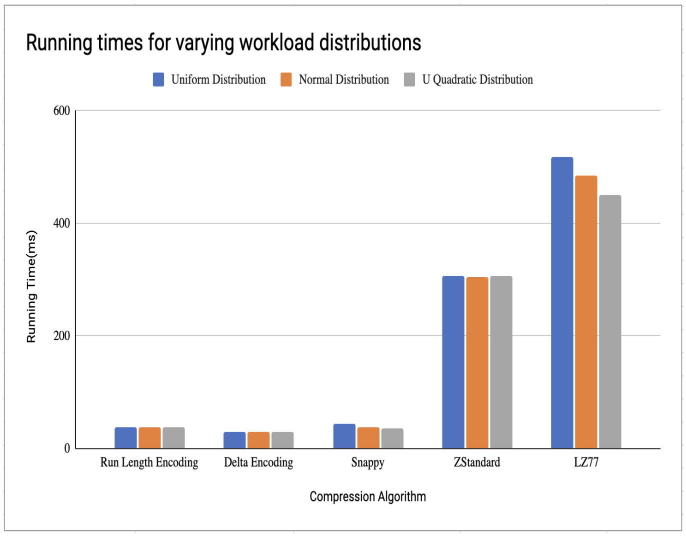

Figure 8: Running time uniform, normal and U Quadratic Distribution

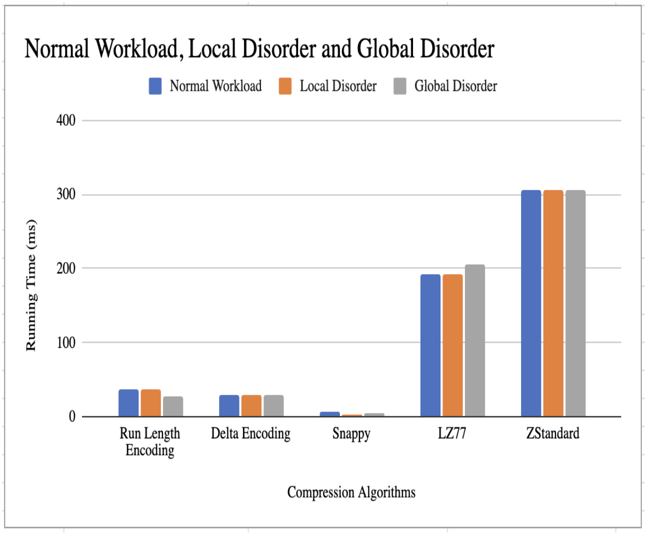

Figure 9: Running time Local and Global Disorder Workloads

the algorithms. Figure 9 shows the executions times where all the different algorithms perform very similarly for the different workloads. Since LZ77 and ZStandard are performing very poorly when compared to other algorithms, the bars are not visible clearly. When only the rest of the algorithms are analysed separately, snappy is performing the best which is almost taking less than 10 ms to compress the workloads which is highly efficient when compared to RLE and Delta Compression. This can be attributed the various combination of algorithms which are used to develop snappy and make it highly efficient. Figure 10 shows this clearly.

The next step was to check how the algorithms perform on workloads with various sizes. We have created workloads from varying from 400 Kb to 400 Mb with 10x increments in the sizes. Figure

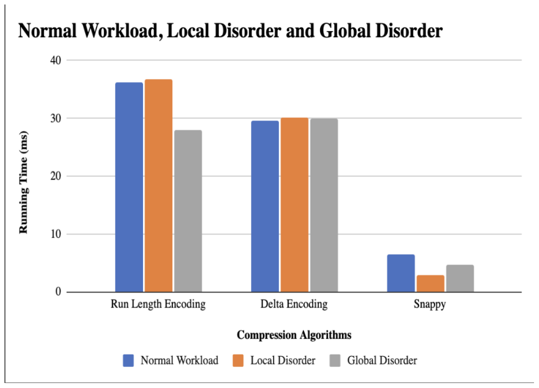

Figure 10: Running time Local and Global Disorder Workloads - RLE, Delta and Snappy

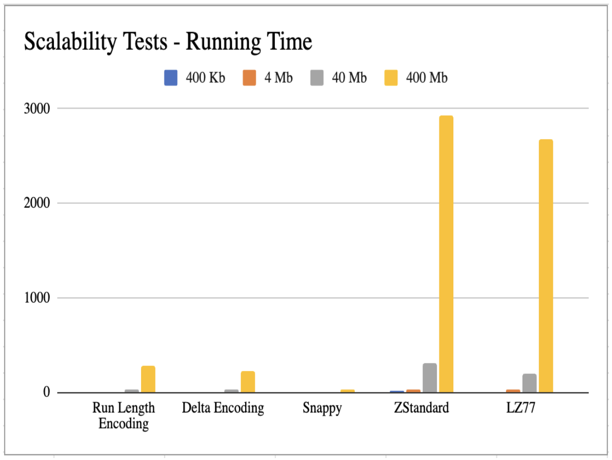

Figure 11: Scalability Workloads running times

11 shows the graph for the execution times for all the algorithms. All the algorithms have running times which are directly proportional to the size of the workloads. Run Length Encoding, Delta and Snappy are very efficient and have very less running times. Snappy still is proving to be the best algorithm.

Near Sorted Workloads All the experiments which are executed until this point has sorted runs with redundant data. These algorithms are fundamentally designed to work well with such redundant data, they can compress these workloads with ease. In the experiments until this point, we have observed that the compression ratio is quite high. One of the main objectives of this research is to understand the performance of these algorithms on near sorted data. To achieve this we referred to the concepts and workloads generator discussed by the authors in the "BoDS: A Benchmark on Data Sortedness" [10]. In this paper, the authors clearly describes the different parameters like K and L which defines the varying sortedness in the workloads. The data generator takes as input user-specified values for the K and L parameters of the sortedness metric as a fraction of the total number of entries (N), as well as the displacement distribution (on L)[10].

To assess the performance of the algorithms, we conducted a benchmark using different combinations of (K, L)-near-sorted sequences: (100, 1), (50, 1), (25, 1), (10, 1), (5, 1), (1, 1), (0, 0), (1, 5), (1, 10), (1, 25), (1, 50), (1, 100), and (100, 100). This approach allowed us to compare the systems across a range of near-sortedness levels, as well as the extremes of fully-sorted and unsorted data[10]. It must be noted that, LZ77 performs really poorly with near sorted and sorted data and it could not provide any results, hence this graph shows a default value of 0 for all the loads. Figure 12 shows the execution patterns for each of the algorithms. ZStandard shows very high execution time for all the workloads. ZStandard combines a rapid entropy-coding stage, a huge search window, and a dictionary-matching stage (LZ77). It makes use of both finite-state entropy (FSE), a fast tabular version of ANS (tANS), and Huffman coding, which is utilized for entries in the Literals section [11]. Since, unsorted data takes a lot longer to find patterns and smaller patterns, this can be attributed to the high latency which is evident in Figure 12.

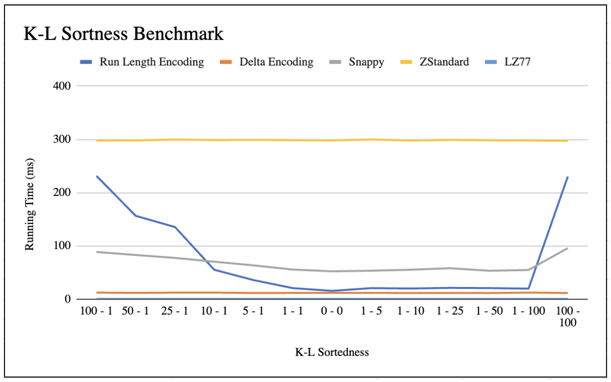

Figure 12: Running time for K-L Sortedness Workloads

Figure 13 shows the execution times for the algorithms but with Run Length, Delta and Snappy Algorithms in focus. We can see that both RLE and Snappy follow a parabolic pattern although, RLE follows a more prominent parabolic pattern. This behaviour of RLE can be attributed to how this algorithm runs and performs. The execution times for K=0 and L=0 configuration works the best for RLE which is completely sorted data. RLE performs worst for completely unsorted data which is being showed by the end points of the parabolic curve. Delta has a close to constant execution time for all the workloads and this can be attributed to how delta find the difference of each consecutive elements irrespective of how the data is being sorted. Although the execution time of Delta is the best, Snappy has a very uniform and constant values for the near sorted workloads.

Compression Ratio To further understand and benchmark these algorithms with another factor we started to look into the compression ratios of each of these algorithms. Figure 14 shows the compression which was achieved in terms of frequency tests. Delta Compression although the best in terms of execution times, the compression ratio takes a big hit. The compressed file is almost the same size as the original file. This is because the number of items in the original file would be the same as the compressed file. Algorithm

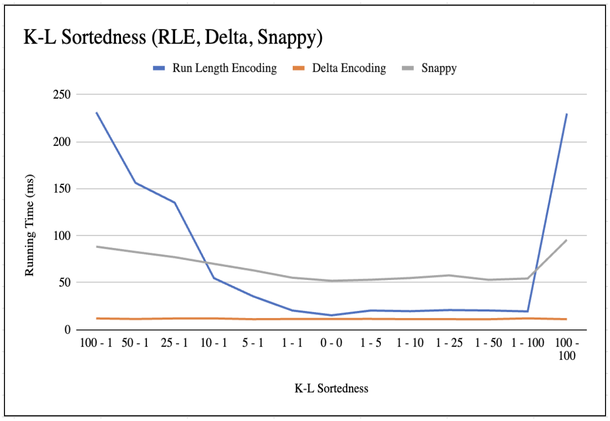

Figure 13: Running time for K-L Sortedness Workloads - RLE, Delta and Snappy

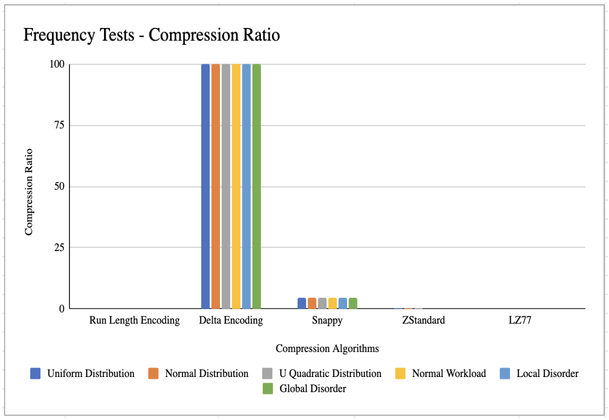

Figure 14: Running time for K-L Sortedness Workloads - RLE, Delta and Snappy

which is being considered as efficient as delta is performing way better that is Snappy compression. For the Frequency workloads Snappy has a very good compression ratio. Other algorithms like, Run Length Encoding, ZStandard and LZ77 are the best in terms of compression ratio as they are close to 0 to 1 percent.

These algorithms perform different for the K-L Sorted Workloads which is being illustrated in Figure 15. The Compression Ratio of RLE is directly proportional to the sortedness of the input workload. Delta is giving the same compression ratio of 100 percent. While Zstandard is performing better and Snappy has the best and consistent compression ratio. Overall when we consider both the factors which is the running time and compression ratio, we can see that Snappy although having comparatively higher running times, it performs well with compression ratios.

6 CONCLUSION

In this work we show how Compression Algorithms perform with variety of datasets. We implemented five different compression algorithms, to benchmark their performance against various sorted

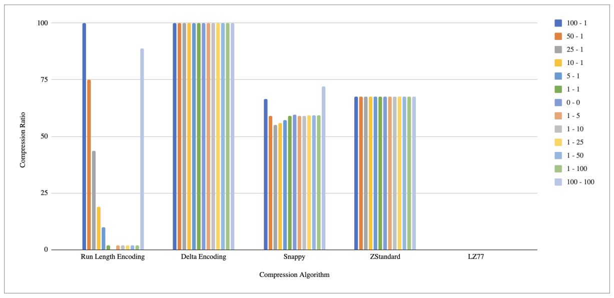

Figure 15: Compression Ratios for K-L Sortedness Workloads

workloads. The Compression Algorithms considered were Run-Length Encoding, Delta Encoding, ZStandard, Snappy and LZ77. Every Algorithm performed differently with different patterns of sorted data. RLE, Delta-encoding and Snappy had very less running time which is lesser than 50ms for Uniform, Normal and U Quadratic Distributions. ZStandard and LZ77 are taking significantly higher latency when compared with the other Algorithms.

Based on our analysis and research and the experiments which we have run, we can deduce that Snappy has a very good performance overall. Although, it has higher execution times when compared to delta in near sorted experiments, it out performs all the other algorithms in other experiments which we have run. One point which must be noted that compression ratio of Snappy is constant and around 21 for all the experiments. Other algorithms can be expected to have higher and better compression ratios but have volatile running times and are inconsistent. Future analysis could include more research on more efficient algorithms like GZIP, Huffman and Arithmetic encoding.

7 REFERENCE

1. John Gantz and David Reinsel, "Extracting value from chaos", IDC iview, vol. 1142, no. 2011, pp. 1-12, 2011.

2. Debra A. Lelewer and Daniel S. Hirschberg. 1987. Data compression. ACM Comput. Surv. 19, 3 (Sept. 1987), 261–296.

3. A. Mensikova and C. A. Mattmann, "Ensemble sentiment analysis to identify human trafficking in web data", Proceedings of ACM workshop on Graph Techniques for Adversarial Activity Analytics (GTA32018), pp. 6, 2018.

4. A. Ungethum, J. Pietrzyk, P. Damme, D. Habich and W. Lehner, "Conflict Detection-Based Run-Length Encoding - AVX-512 CD Instruction Set in Action," 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), Paris, France, 2018, pp. 96-101, doi: 10.1109/ICDEW.2018.00023.

5. S. Fiergolla and P. Wolf, "Improving Run Length Encoding by Preprocessing," 2021 Data Compression Conference (DCC), Snowbird, UT, USA, 2021, pp. 341-341, doi: 10.1109/DCC50243.2021.00051.

6.D. R. Vasanthi, R. Anusha and B. K. Vinay, "Implementation of

Robust Compression Technique Using LZ77 Algorithm on Tensilica's Xtensa Processor," 2016 International Conference on Information Technology (ICIT), Bhubaneswar, India, 2016, pp. 148-153, doi: 10.1109/ICIT.2016.041.

7. Kovacs, Kyle. "A Hardware Implementation of the Snappy Compression Algorithm." UC Berkeley EECS Masters Thesis (2019).

8. http://ijrar.com/upload_issue/ijrar_issue_2012.pdf

9. https://www.hypr.com/security-encyclopedia/running-time

10. Raman, A., Karatsenidis, K., Sarkar, S., Olma, M. and Athanassoulis, M., 2023, March. BoDS: A Benchmark on Data Sortedness. In Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers (pp. 17-32). Cham: Springer Nature Switzerland.

11. https://en.wikipedia.org/wiki/Zstd