ADM - Assignment 3

Group 2
Lucas van Rooij (s1695185)
Nick Radunovic (s2072724)
Srishankar Sundar (s3151298)
Luiza Plaiu (s3340228)

November 14, 2021

1 Introduction

Five different encoding techniques have been discussed and covered for various types of data. Each file provided has been encoded, and the encoded file has been decoded back. The compression ratios across the different types of data and the different techniques have then been discussed.

The encoding techniques and their subsequent decodings have been implemented on **Python 3.8**

1.1 Background

2 Methods

2.1 Uncompressed Binary Format

The uncompressed binary format is not a compression technique such as the remaining techniques used in this assignment. It merely writes the numerical data as machine-readable raw bytes instead of the human readable numerical representation of the data. As this technique does not actually compress the data, no decrease in file size is expected. For implementation the built in python method *to_bytes()* for encoding, and *from_bytes()* is used. For encoding and decoding, we have used little endian byte ordering.

2.2 Run-Length Encoding

Run-Length Encoding focuses on representing data with two pieces of information - The data itself and the number of successive repetitions for that data. It follows that Run-Length Encoding would be an efficient form of compression when the data contains a significant amount successive repetitions in data, and results in larger file sizes on files with no repetitions, or a small number of repetitions. This is discussed further under the Results section.

2.2.1 Encoding

The data is read as a list of strings, where each record is a separate string. This is then sent to the encode method, which uses the *itertools* module and underlying *chain* and *groupby* methods to perform the encoding.

Each entity in the encoded string is then separated by a # character, to aid in decoding. This made it so that the decoder could effectively identify where the data ends and the number signifying the repetitions begins.

It is to be noted that # was picked as a separator character as that specific character was not found in the data. Depending on the data, different separators could be used.

2.2.2 Decoding

Following from the implementation of the separators, the decoder simply iterates through the encoded file (loaded again as a list of strings) with the understanding that the format followed is *(data)#(number of reps.)#(data)#(number of reps.)...*

2.2.3 Modified Encoding/Decoding

This was implemented as part of the same script but only works for datasets which contain a single integer between 0-128. (It is therefore hyper-specific and not useful in a broad sense).

This can be tried out by using the argument *rle_mod* instead of *rle* on the script. The idea of this method is to convert all integers between 0-128 into their equivalent ASCII characters and store them as the ASCII characters. Thus this reduces the amount of space needed on the file drastically. The decoding re-converts these ASCII characters back into their corresponding integers.

2.3 Dictionary Encoding

Dictionary encoding is a lossless compression technique which means that it is encoding the data using fewer bits than the original form of representation of information which is reducing the storage needed for the file while allowing the option to be completely reversible (decoding the compressed file to its exact original form).

Dictionary encoding is highly efficient for files composed of strings (words) because a word of several bits can be replaced with its dictionary key which is usually represented by a very small int value. The process consists of creating a concordance of all the strings in a text and extracting all of the unique values which are then built into a dictionary (each unique value is assigned to a reference). A downside of dictionary encoding to take into consideration is the fact that the dictionary used for encoding must also be stored with the new compressed file in case decompression is needed. There are instances where said dictionary can be consuming more space than the actual encoded data (when there are only unique values, or the initial values are very small). In files consisting of only small integers, dictionary compression is highly ineffective because it is replacing small values with equally as small values while storing the extra information of the dictionary, which can make the encoded file to turn out larger than the original one.

In order to decompress a dictionary encoded file, it is necessary to also have access to the original dictionary used for the encoding process and to identify the patterns that match said dictionary. To get the information in its original form, the only necessary step is replacing patterns in data with their dictionary reference.

2.4 Frame of Reference Encoding

Frame of Reference Encoding is a compression technique that compresses numerical data by storing the offset of a given numerical value from a chosen frame of reference instead of the value itself. The data gets compressed only if the relevant offset value can be stored using less bits than otherwise would be needed for storing the original value. Therefore, the encoding only takes place when the offset can be stored in b bits. When the encoding obtains an offset that can't be stored in b bits, a special escape code is used indicating a difference larger than can be stored in b bits. After the escape code, the original (uncompressed) value is written to the compressed file).

For the experiments, the frame of reference was set to the median of all numerical values in the given dataset, minimizing the need for escape codes. For the maximum number of bits to be allowed for storing the encoding, $b = 4$ bits was used. For all encoding obtaining an offset that could not be stored in $b = 4$ bits, an escape code was used.

2.5 Differential Encoding

Differential Encoding is similar to Frame of Reference Encoding in that it compresses numerical datasets by storing the offset of each value based on a frame of reference. However, unlike Frame of Reference, Differential encoding make use of a ‘sliding’ frame of reference that gets updated after each record in the dataset, opposed to using a ‘fixed’ one. The frame of reference at each point in time is the uncompressed value of the previous record in the dataset. Justl ike Frame of Reference Encoding, an escape code was used to indicate a difference larger than can be stored in b bits. After the escape code, the original (uncompressed) value is written to the compressed file. For the maximum number of bits to be allowed for storing the encoding, $b = 4$ bits was used.

3 Results

The CPU specifications the various techniques were run on for recording the results is given below:

- **bin**: Intel Core i5-6300 CPU with 2.50GHz clock speed
- **rle**: Ryzen 5 3600H with 3.3GHz clock speed
- **dic**: Intel Core i7 with 2.5GHz clock speed
- **for** and **dif**: Intel Core i7-9750H CPU with 2.60GHz clock speed

3.1 Compressed File Sizes

The table below illustrates the file sizes on compression for the five techniques. Red cells represent a net increase in file size whereas cells in green imply a net decrease.

Datatype	**Filename**	**Input File Size(MB)**	**Output File Sizes(MB)**				
			bin	rle	dic	for	dif
String	l_comment	157.4	NA	168.8	316.8	NA	NA
	l_commitdate	63		73.5	20.9		
	l_linestatus	11.4		3.4	12		
	l_receiptdate	63		74	27.3		
	l_returnflag	11.4		8.1	12		
	l_shipdate	63		73.9	27.3		
	l_shipinstruct	74.4		64.4	12		
	l_shipmode	30.2		35.7	12		
int8	l_discount	12	92.7	21.3	12.5	5.9	6.4
	l_linenumber	11.4	94.9	22.1	12	5.9	5.9
	l_quantity	16.1	88.4	27	16.8	10	10.1
	l_tax	11.4	94.9	20.4	12	5.9	6
int16	l_suppkey	28	108.6	39.4	29.4	23.4	23.4
int32	l_extendedprice	45	141.6	56.5	47.4	46.9	46.9
	l_linenumber	11.4	166.9	22.1	12	23.4	23.4
	l_orderkey	44.7	142.2	14	47	46.9	46.9
	l_partkey	36.9	153.8	48.3	38.8	46.9	46.9
	l_suppkey	28	156.6	39.4	29.4	46.9	46.9
int64	l_discount	12	260.7	21.3	12.5	46.9	51.5
	l_extendedprice	45	237.7	56.5	47.4	93.8	93.8
	l_quantity	16.1	256.4	27	16.8	79.7	80.8
	l_tax	11.4	262.9	20.4	12	46.9	48

3.2 Execution Times for Encoding

File Type	File	Time for encoding(seconds)				
		bin	rle	dic	for	dif
string	l_comment	NA	3.2	2389.12	NA	NA
	l_commitdate		2.88	348.88		
	l_linestatus		0.44	24.13		
	l_receiptdate		2.82	909.23		
	l_returnflag		0.94	21.75		
	l_shipdate		2.86	228.29		
	l_shipinstruct		2.34	23.73		
	l_shipmode		2.3	16.04		
int8	l_discount	1.62	2.29	31.53	10.74	10.34
	l_linenumber	1.68	2.43	25.63	11	10.02
	l_quantity	1.63	2.65	70.29	14.49	13.3
	l_tax	1.62	2.33	32.76	11.86	9.76
int16	l_suppkey	1.82	2.7	231.2	16.37	15.19
int32	l_extendedprice	1.82	2.82	401.2	16.92	15.55
	l_linenumber	1.63	2.52	25.02	10.92	9.37
	l_orderkey	1.8	0.77	457.32	16.02	10.55
	l_partkey	1.79	2.86	222.82	16.05	15.51
	l_suppkey	1.84	2.65	229.34	15.68	15.34
int64	l_discount	2.14	2.3	32.12	10.53	10.53
	l_extendedprice	2.23	2.82	398.02	16.55	15.97
	l_quantity	2.14	2.71	108.47	14.45	13.86
	l_tax	2.16	2.27	25.14	10.64	9.71

3.3 Execution Times for Decoding

File Type	File	Time for decoding(seconds)				
		bin	rle	dic	for	dif
string	l_comment	NA	2.44	2112	NA	NA
	l_commitdate		2.51	711.2		
	l_linestatus		0.65	33.12		
	l_receiptdate		2.26	1742.22		
	l_returnflag		1	26.12		
	l_shipdate		2.22	621.12		
	l_shipinstruct		1.88	29.94		
	l_shipmode		1.93	30.13		
int8	l_discount	2.67	2.09	70.8	1.94	2.12
	l_linenumber	2.66	2	53.44	1.81	1.83
	l_quantity	2.69	2.1	168.62	2.73	2.93
	l_tax	2.7	2.02	56	1.96	1.93
int16	l_suppkey	2.64	2.3	524.01	7.39	7.57
int32	l_extendedprice	2.67	2.24	802.1	15.93	15.51
	l_linenumber	2.66	1.98	51.25	7.22	7.59
	l_orderkey	2.67	0.94	897.45	14.74	8.03
	l_partkey	2.68	2.25	414.01	14.9	15.52
	l_suppkey	2.67	2.14	519.02	14.6	15.37
int64	l_discount	2.64	2.33	71.2	14.63	17.27
	l_extendedprice	2.69	2.14	806.47	28.84	30.76
	l_quantity	2.68	2.1	192.11	24.31	26.45
	l_tax	2.71	1.87	44.13	14.36	15.77

4 Discussion & Conclusion

- Dictionary encoding does well for most of the string datasets. It particularly performs well for datasets containing a high amount of repetition, for example *l_commitdate*, *l_shipdate*, among others. On the other hand, Dictionary encoding, does not prove any efficiency when compressing lists of small integers, returning an encoded file of the same size or even larger (because of the added dictionary stored for the later decompression process).
- Both RLE and Dictionary Encoding result in an increase in file sizes in most cases. This implies that most files do not have a high degree of repetition
- The execution times for Dictionary Encoding/Decoding are significantly higher than that of the other techniques. This can be attributed to Dictionary encoding simply being a much more resource intensive technique, as well as relative differences in compute resources between the different systems the techniques were run on.
- When we look at the compressed file sizes we see that for uncompressed binary format the file size increases after applying the compression technique. This can be explained by the fact that it is not really a compression technique but rather a different way of representing the data without performing any kind of compression. We see that as the data type becomes larger, the resulting binary file also becomes larger. The execution times for both decoding and encoding to binary format show that converting the data to binary format is relatively fast when comparing to the other compression techniques.
- Regarding the performance of both Frame of Reference and Differential Encoding, the large compression size obtained for datasets consisting of 32-bit and 64-bit integers could be explained by the number of bits b allowed for storing the resulting offset for each value. Reasonably, it might be the case that

the compression performance would have been better for both techniques when the b bits allowed for storing the offset (i.e. encoding) scaled with increasing integer sizes. Hence, $b = 4$ might have been too small to obtain good compression results for datasets containing 32-bit and 64-bit integers. Further comparison can be conducted in order to see whether or not using $b = 6$ and $b = 8$ improve compression performance for both Frame of Reference Encoding and Differential Encoding.