(De)-Compression of Databases for high data throughput on FPGAs

Klemens Korherr — Mtr. 3715034
Institute of Computer Engineering, University of Heidelberg

October 27, 2023

Abstract

Your abstract.

1 Introduction

In 2020, 64.2 Zettabytes of data were generated and stored in databases. By 2025, this figure skyrocketed to 181 Zettabytes [1]. This exponential growth in data [2] can be attributed to various sectors, such as internet usage, social media, research, and financial markets, all contributing significantly to the digital data landscape. These sectors amass massive databases, each containing millions of data sets. It's not the computational tasks themselves that are the most energy-intensive, but rather the transfer and filtration of data within databases. These processes consume a substantial amount of power, underscoring the challenges of managing and harnessing the vast data resources of the digital age.

To reduce the energy consumption and increase the data throughput it is desirable to have the data located closely to the calculation unit. For databases it would be desirable to perform operations like 'Where' clauses close to the physical storage. This would remove memory contention on the remaining CPU subsystem and could save substantial energy. Furthermore a compression and decompression of the data inside the storage would have an positive effect on the energy consumption. We could increase the amount of data that could be stored while saving throughput on the communication channels. This means we can keep the channels busy with filtered data and don't need to move many raw data.

This method requires a fast decompression and filtering of the compressed data in the storage. By using data (de-)compression and filtering inside the storage elements of an FPGA chip, it is possible to store and access a lot of data in parallel. The filtered data can easily be transmitted to the remaining CPU subsystem. This requires data processing at line rates which may not economically suit all possible compression methods.

This paper deals with question which (de-)compression techniques are the best method to compress and decompress databases. The methods are investigate with the respect of the capability to be implemented on FPGAs for delivering high throughput requirements. Therefore we analyze existing compression techniques. We investigate the compression ratio as well as the decompression time needed for a small example database. This shall provide us a better understanding of (de-)compression requirements of different workloads and the FPGA implementation cost performance-tradeoffs for different tailored compression techniques.

We used three test databases. The databases containing different data-types and multiple tables. For compression, we have limited ourselves to the data types: float, integer, boolean, date, datetime and text. The first database is created by TPCH. That database represents a warehouse and is filled with random generated data. This test-database is a size of 1GB. The second database corresponds to the 'Stackoverflow' database. In this database the data is closer to real world data. By changing and adding some data types we increase the variety of data types and can build up a more complex database. This database is with 411MB smaller than the first one. As a last test database we used a small weather database provided by the 'National Centers for Environmental Information'. This database contains small tables with around 5-8MB filled with weather information of different weather stations.

Run-length, Delta-Encoding, Bitpacking and Huffman are the four compression and decompression methods we are using in this work. These are simple and commonly used compression techniques which are easy to be implemented on FPGAs. Between these four compression methods we tried to find the best corresponding method for each data type. We also combined some of these methods to receive a better compression ratio.

The paper is structured as follows: In Section 2 we describe the used (de-)compression algorithms in general and some technical background. Followed by the explanation of the benchmarks in section 3. The compression of the different data types is mentioned in Section 4. Continuing with the decompression on the FPGA in Section 5. In the last section we discuss the results and give a conclusion.

2 (De-)Compression Algorithms

The primary objective of data compression is to minimize the number of bits required to represent a symbol or numerical value. By reducing the bit count per symbol or number, it becomes possible to store more data in memory without the need to increase the memory capacity. Additionally, this approach allows for the transfer of more information between systems without necessitating an expansion of the existing bandwidth.

However, it's important to note that the implementation of compression and decompression comes with computational overhead. The processes of compressing and decompressing data can be quite computationally intensive. The associated costs vary depending on the specific algorithms employed. Some methods may incur high computational expenses during compression but are exceptionally efficient during decompression. This particular trait proves advantageous when it comes to maintaining static data in a compressed state in memory, as fast decompression algorithms can swiftly retrieve the data when needed.

In this research, we utilized straightforward and widely recognized lossless compression techniques. One such technique is run-length compression, which reduces the size of the database by grouping symbols together, consequently reducing the number of symbols needed to represent the database. Additionally, we employ compression algorithms like Delta, Bitpacking, and Huffman, all of which work by reducing the number of bits required to represent a symbol. Noting that it is possible to combine multiple compression algorithms, although as the number of employed compression algorithms increases, so does the time required for compression and decompression.

Given our primary focus on achieving rapid decompression on an FPGA, the compression time is not of importance in our context. The efficiency of encoding and decoding can vary depending on the specific compression algorithm used. Some algorithms may be relatively slow in the compression phase but fast in decompression speed.

Our approach involves storing data in the memory of an FPGA in a compressed format. By implementing the straightforward decompression algorithms mentioned earlier on the FPGA, our aim is to maximize bandwidth utilization and maintain constant data flow through the channels with valuable information. The decompressed data can than be used by a CPU-subsystem for further calculations.

The important metrics that are used to assess the (de)compression are:

- compression ratio: This is one of the most popular metrics used to evaluate the effectiveness of a data compression algorithm. It is defined as the ratio of the original data size to the compressed data size. The compression ratio indicates how much the data has been reduced or compressed by the algorithm. The ratio is a number between 0 and 1. A small ratio indicates an effective compression. It is calculated like in the formula below. [3]

$$compression\ ratio\ \eta = \frac{size(compressed)}{size(original)}$$

- decompression time: The time needed to decompress the data to receive a high throughput. For this purpose, the maximum possible clock speed at which decompression can be carried out on the FPGA is measured. Together with the analysis of how many clocks it takes until a decompressed value is output, the decompression time can be calculated. The compression time is not investigated, since the focus is on the fast decompression.

- throughput: The throughput gives the size of decompressed data that is the output of the decompressing and is applied to the remaining CPU-Subsystem. The throughput is calculated by multiplying the decompression time and the size of the decompressed data. The throughput metric give bytes per second.

- hardware usage: Gives information about the size of the needed hardware on the FPGA. It is desirable to keep the hardware overhead for the decompressing as small as possible.

We decided to use the following algorithms. These are simple to implement on FPGA and can reach high throughput. Furthermore the small hardware usage of these algorithms is also an advantage and the algorithms can easily be combined to receive better compression ratios.

2.1 Run-length

Run-length encoding (RLE) is a lossless data compression technique used to reduce the size of data by replacing consecutive repeated occurrences of the same value with a single representation of the count of it's occurrences followed by the value. The combination of value and occurrence is called tuple. A tuple is in the form (l,t), where t is the value or symbol and l the count of the repeating occurrence of that value. The fundamental idea behind RLE is to exploit the redundancy in sequential data, especially when there are long sequences of the same value or pattern.

For example the data-stream "AABBBBBCCCCD" is compressed into 4 tuples: (2,A), (5,B) (4,C) and (1,D). By this the data can be compressed from 12 symbols to 8 symbols, which is just 2/3 of the original data.

The compression ratio improves as the length of the sequences of symbols increases. Longer sequences allow a better compression, resulting in a higher ratio of data reduction. Conversely, when dealing with short sequences, the compression may yield a less favorable ratio, requiring more symbols to represent the data compared to its original form. This is due to the fact that each compressed tuple consists of two symbols. As a result, the best achievable compression ratio is determined by the length and frequency of repeating sequences within the data:

$$\frac{|t| + |l|}{|t| * l} = \eta RLE$$

Where —t— is the number of bits to encode the symbol and —l— is the number of bits used to encoded the run-length. $|t| + |l|$ is a always the size of one tuple and represents the best case $|t| * l$ bits. Where l is the length of the repeating sequence. The worst-case the compression ratio is calculated by [4]:

$$\frac{|t| + |l|}{|t|} = \eta RLE$$

In the worst-case the tuple of size $|t|+|l|$ just encodes the symbol —t—. This means the compressed data is —l—-bits larger than the original. When a fixed number of bits is allocated for representing both the symbol and the length in run-length encoding, the compression can potentially double the number of bits required to represent the data compared to its original form [4].

RLE is considered a simple and straightforward compression technique, especially for data that contains repeated patterns or long sequences of the same value. However, its effectiveness depends on the characteristics of the input data. It performs best on data with significant redundancy and regular patterns. For highly random or non-repetitive data, RLE may not achieve significant compression gains.

In some columns of the database, the values rarely change between entries. Run-length compression is suitable for this. In most cases, however, run-length compression is combined with delta, as will be shown in the compression section 4.

Run-length decompression is easy to implement on the FPGA and requires few resources. As will be shown later, decompression on FPGA can also achieve high performance.

2.1.1 Run-length for Boolean

Boolean values can be effectively compressed using a specialized version of run-length encoding tailored to their binary nature. As boolean values can only take two states, true or false, we can simplify the compression approach. We use the inherent two-state characteristic of boolean values to compress them. Unlike the symbol-based method previously described, we simplify the process by disregarding the symbol part. In this variant we count the occurrences in a consecutive sequence of trues or false. The counts are directly added to the list of compressed data. With each new number in the list the values switches from the previous state to the opposite state.

For example the sequence [True, True, True, False, False, True, False] is compressed in [1,3,2,1,1]. By this we reduced 7 symbols to 5 numbers. Where the first value gives with 1 or 0 the initial state true or false of the first symbol in the sequence.

2.2 Delta-Encoding

Delta-Encoding is another simple and lossless compression algorithm used to reduce the size of data by encoding the difference between consecutive elements in a sequence rather than encoding each element independently. That method calculates and represents the changes (deltas) between adjacent elements in the sequence and stores the result instead of each element in its original form [5].

As an example, let's consider the sequence of integer values $[7, 12, 15, 20, 22, 25]$. We can compress this sequence using delta encoding, which involves calculating the differences (deltas) between consecutive integers: $deltas = [7, 5, 3, 5, 2, 3]$.

With delta encoding, we reduce the size of the values, requiring fewer bits to represent the numbers. Assuming all values in the sequence are represented with the same number of bits, the largest number in the uncompressed sequence is 25, which needs 5 bits. If we represent each of the 6 values in the sequence, it would require $(6values * 5bits = 30bits)$.

However, the largest delta in the compressed sequence is 7, which can be represented with just 3 bits. As all deltas are positive values, we don't need to worry about negative numbers in this example.

The 5 deltas can be represented using 15 bits ($5\ deltas * 3\ bits$). Additionally, the initial value of the sequence, in this case, 7, requires another 3 bits to be represent. Therefore, the compressed data, including the 5 deltas and the initial value, can be represented with a total of 18 bits.

$$compression\ ratio\ \eta = \frac{18bits}{30bits} = 0,6$$

The compression ratio is 0.6, meaning that the compressed data is less than 2/3 of the original data.

Delta encoding is a useful compression method, particularly when dealing with input data that exhibits small deltas (differences) between consecutive values. When the deltas are small, fewer bits are required to represent them, resulting in efficient compression. However, as the size of the deltas increases, so does the number of bits needed to represent them, which can result in a larger compressed data set compared to the original.

Significant sign changes in the original data can contribute to faster increases in delta size. For example, consider a sequence that contains [7, -7]. In the original data, 3 bits are used to represent the absolute values, and 1 bit is needed for the sign, making a total of 4 bits for the sequence. However, the delta between these two numbers is -14, which requires 4 bits to represent the absolute value and an additional bit for the sign, resulting in a delta sequence size of 5 bits. When considering the initial value of 7, including the sign, it requires 4 bits for representation.

As a result, the compressed data, including the delta sequence and the initial value, requires a total of 9 bits. In this example, the compressed data is slightly larger than the original data (9 bits vs. 8 bits) due to the negative sign change and the larger magnitude of the delta.

In summary, delta encoding offers efficient compression for data with small deltas but may lead to increased compressed data size if the deltas become large or if there are frequent sign changes in the original data. The effectiveness of delta encoding depends on the specific characteristics of the input data, and it's essential to consider the trade-offs between compression and delta size for each application.

2.3 Bitpacking

Bitpacking is a common data compression technique that optimizes the storage or transfer of data by efficiently utilizing the binary representation of the elements. It is particularly effective when dealing with data-sets containing values that require fewer bits than the standard word size of a computer system [6].

This method involves grouping multiple data elements together and storing them compactly in binary form within a fixed-size storage unit, such as a byte or a word. This is achieved by assigning each data element a specific number of bits based on its value range, such that the smallest number of bits necessary are used [6].

For example, if we have a data-set with values ranging from 0 to 15, we could represent each value using only 4 bits ($2^4 = 16$) (excluded a sign bit). With such a sequence of values, bitpacking allows to store several values within a single byte, efficiently using the available space and reducing memory or transfer overhead.

Consider a sequence like $[4, 15, 2, 0, 12, 8, 6, 10]$ in which each element is represented using an 8-bit unsigned integer, resulting in a total of 8×8 bits $= 64$ bits for the entire representation.

Bitpacking compression involves analyzing the sequence to identify the largest number within it. The goal is to determine the minimum number of bits required to represent that largest number accurately. Once this bit count is established, all values in the sequence can be compactly represented using that fixed number of bits.

In the given sequence, the highest value is 15, which necessitates 4 bits for its representation. By applying this bit count to all elements, we achieve a compressed size of 8×4 bits $= 32$ bits. Consequently, the data is losslessly compressed to half of its original size.

It's important to note that bitpacking does not use a dynamic number of bits per value. This means that each value is represented by a uniform, fixed number of bits (in this case, 4 bits). This approach avoids introducing excessive overhead that would be required if each number needed its own varying number of bits, which would necessitate additional bits for specifying the bit count for each value and result in larger overall data size [6].

2.4 Huffman

Huffman encoding achieves lossless data compression by assigning new codes to symbols. To accomplish this, a dictionary containing the symbols intended for compression is employed. Each symbol in this dictionary is associated with a distinct code, varying in length. The symbols are organized within a binary tree. Symbols with higher frequency in the data being compressed are positioned higher in the tree, necessitating fewer bits for their representation. Conversely, symbols occurring less frequently are positioned in the lower levels of the tree, which might result in a greater number of bits needed for their representation compared to the original uncompressed data. As new symbols are added, the tree expands accordingly [7].

Huffman encoding reaches the best results when certain symbols exhibit high frequency while others are less common. The process of generating the compression and decompression dictionaries can be accomplished through two distinct methods.

2.4.1 Static-Huffman

In the static huffman compression the binary tree is pre-generated and is used to compress all the data. Means the huffman tree is build up not knowing the data which is going to be compressed with the tree. The significant benefit of a static tree is that it doesn't need to be included in the compressed data. Which reduces the size of compressed output. However the compression ratio can be worse than with the dynamic huffman tree, since the tree order is not optimized for the current input data.

2.4.2 Dynamic-Huffman

The dynamic huffman creates the binary tree while compressing the data. The algorithm counts the occurrence of each symbol in the uncompressed data, depending on that occurrence the symbol is ordered in the level of the binary tree. With the dynamic tree a better compression ratio can be achieved than with the static variant. However for decompressing the generated dictionary with keys and symbols needs to be saved and transmitted as well. This increases the compressed data.

3 Benchmarks

The aim of choosing the benchmarks is to be able to represent as many occurrences of data characteristics as possible in the data-sets. These data-sets should encompass all conceivable scenarios found in databases, allowing us to obtain compression and decompression results across a wide range of data characteristics. In our pursuit to test and validate the compression and decompression algorithms and determine the most effective combination of methods for achieving the highest data throughput on an FPGA, we have employed three data-sets. Each of these data-sets is readily available and comprises multiple tables of varying sizes. These tables within a data-set contain multiple columns, each with different data types. The data types are restricted to the following: Integer, Float, Boolean, Strings, Date, and Datetime.

The data-sets were optimized before compressing. Due to data protection, some columns in the public database were left blank. Since this does not reflect the real database, these columns were either filled with randomly generated values in the optimization or the column was removed from the data-set. Furthermore in one of the data-sets a column of the type Boolean is added, since that data type is missing in the benchmarks. This column is filled randomly with values.

The characteristics of the data within the tables also varies. To ensure that our results encompass a wide range of data types and characteristics, we employ multiple tables and databases with diverse data content. To get an overview of the data characteristic in the tables of the databases the Shannon entropy is calculated:

$$H(x) = \sum_{n}^{i=1} (p(xi) * log2(p(xi)))$$

H(x) is the entropy over a column in the database and p(xi) is the probability of each element in the column to occur. With higher probability the entropy is smaller. A small entropy indicates an ordered data-set with few different values. The greater the entropy, the greater the disorder in the data-sets. A high variety of values in a data-set create a small probability and a high entropy. This can be an indicator for a high information density. Whereas a small entropy indicates a smaller information density, since many values are redundant. The entropy for text-columns is calculated over the occurrence of single characters. This is because the subsequent compression compresses the individual characters and not the entire content of the cell as a whole. The characteristics of the data-sets and some reference compression's with 7Zip are described in the following subsections.

3.1 Stack Overflow - Database

The 'Stack Overflow' database is a freely available subset of the 'Stack Overflow' website's database [8] [9]. This means that the database includes actual data and real data structures, making it an ideal benchmark for testing algorithms on a real-world database. The database comprises various tables, including votes, users, posts, post links, badges etc. For our purposes, we have chosen to utilize the following tables: badges, users, link types, post links, post types, votes, and vote types. It's important to note that all user-contributed content within this data-set has been anonymised, ensuring that no identifiable information such as names, passwords, or emails is present. These leads to empty columns in the data-set. To receive a complete database without empty columns, the columns are filled with random values or are removed. The selected tables present a variety of different data-structures as well as different sizes of the tables and all data-types that are going to be investigated in the compression and decompression. Furthermore there are several sizes of the data-set available, this makes it possible to increase the data-set if necessary. In the following table 1 the entropy values of the database are shown. Each column of each table is given with its datatype and the average entropy of the column.

Entropy of Stack Overflow database						
LinkTypes	postTypes	voteTypes	badges	votes	postLinks	users
Int:1.0	Int:3.0	Int:3.91	Int:20.07	Int:23.27	Int:17.30	Float:18.19
Text:3.64	Text:4.27	Text:4.45	Text:4.49	Int:20.61	DT:16.73	Int:6.57
			Int:16.62	Int:2.14	Int:16.83	Date:9.43
			DT:19.58	Date:9.64	Int:16.13	Text:5.11
					Int:0.0	Int:2.17
						Int:18.19
						DT:18.19
						Text:4.33
						Float:14.65
						Float:5.24
						Float:7.18
						Text:4.29
						Float:18.19
						Bool:1.0
						Text:4.72
Total:2.32	Total:3.63	Total:4.18	Total:15.19	Total:13.92	Total:13.40	Total:8.38

Table 1: Entropy's of each column and table of the Stack Overflow database

For all seven used tables of the database is the entropy of each column and the average entropy of the table calculated. As indicated in the table 1, the database exhibits a wide range of entropy values. The data-set includes columns with frequent repetitions of numbers, as well as columns with a multitude of unique numbers. The same pattern extends to date columns. In text columns, the entropy is generally low. This is attributed to the fact that text columns typically have a maximum diversity of 128 different elements in ASCII representation. Once this maximum number of different elements is reached, the total probability remains constant, only the internal probability distribution of individual elements changes. Consequently, this leads to a lower average entropy, as fewer distinct elements repeat more frequently with increasing text length.

The first three tables are characterized by their limited data-set size They reach a small entropy and so a small information density. The remaining tables contain substantial amounts of data, and reach a higher entropy, leading to a higher information density. The data-set encompasses both ordered and unordered data, facilitating the coverage of a wide spectrum of data characteristics.

The Stack Overflow database can be accessed within an SQL environment. From there, we've extracted select tables and their data, exporting them as CSV files for compression. We've chosen CSV because it preserves the database's data structure and is easily accessible in Python. The exported data in the CSV files occupies a size of 394MB. Comparatively, the database size is smaller at 232MB.

The table 2 shows the size of the original data (database-size) and the memory space needed per table. The original size is calculated by the default amount of bytes that is used to represent the corresponding datatype in a database multiplied by the number of entries per column. Furthermore the table contains the compression of the single tables and the complete database with 7Zip. The compression is fulfilled with two different compression processes on the exported CSV-file. Therefore the size of the

Reference compressions of Stack Overflow								
	LinkTypes	postTypes	voteTypes	badges	votes	postLinks	users	Total
DB-Size	39B	181B	335B	36.8MB	152.2MB	3.9MB	38.3MB	232MB
CSV	30B	124B	228B	57.7MB	290MB	8.53MB	37.7	394MB
BZIP2	199B	258B	328B	9.29MB	52.6MB	1.98MB	9.63MB	78.4MB
LZMA	165B	230B	301B	8.03MB	41.0MB	1.47MB	10.1MB	64.8MB

Table 2: Benchmark size and the reference compression's with 7Zip

exported CSV-file is also given. In the BZIP2 compression a sliding-window size of 900KB used. This is the highest value that can be selected in the compression. The second compression algorithm used in 7Zip is LZMA. This method achieves a better compression result than BZIP2. Here a sliding-window size of 16MB is used. But for the smaller tables like LinkTypes or postTypes the compression result is just slightly better. However, with both methods the compressed files of the small tables are larger than the originally uncompressed files. The best compression ratio can be achieved in the compression of the vote-table.

These values are used to compare the compression results of our algorithm to well known ones.

3.2 TPC-H - Databse

The second benchmark database used is TPC-H. This is another free rational warehouse database. The TPC-H benchmark is often used to test the performance of database systems. Therefore a large warehouse database is created and different queries needs to be performed by the system. The data-set consists of eight tables with different sizes and number columns. We are using the database with its structure to perform the compression and decompression on it. The advantage of TPC-H is that the tables are filled with randomly generated values, which gives the opportunity to define the size of the database dynamically [10]. In this work a database of the size 1GB is used.

As indicated in the entropy table 3, the data characteristics in the TPC-H database exhibit variations similar to those in the 'Stack Overflow' database. This data-set encompasses both unstructured and structured data, with notable distinctions, particularly within the text columns.

In contrast to the 'Stack Overflow' database 3.1, which features real-world words and data as one would expect in an actual database, the TPC-H database primarily comprises randomly generated strings that lack a connection to real data-sets. This benchmark encompasses all data types except for datetime.

The table 4 shows the size of the database and its tables. The TPC-H database consists of multiple tables with different sizes. The largest table is LineItem which accounts for three quarters of the entire database. Like before the table gives the size of the database in the db-size, as well as the size of the exported tbl files. But in this case the sizes of the both uncompressed files do not differ. The compression process is carried out using the BZIP2 and LZMA compression methods within 7zip. In nearly all instances, compression successfully reduces the size of the table. Notably, the BZIP2 compression

Entropy of TPC-H database							
Customer	LineItem	Nation	Orders	part	partSupp	region	supplier
Int:17.19	Int:20.32	Int:4.58	Int:20.52	Int:17.61	Int:17.61	Int:2.0	Int:13.29
Text:3.92	Int:17.59	Text:4.08	Int:16.48	Text:4.29	Int:13.29	Text:3.52	Text:3.59
Text:6.0	Int:13.29	Float:2.32	Text: 1.15	Text:3.54	Int:13.28	Text:4.17	Text:6.00
Int:4.64	Int:2.61	Text:4.26	Float:20.47	Text:3.33	Float:16.52		Int:4.64
Text:3.36	Int:5.64		Date:11.23	Text:4.11	Text:4.28		Text:3.36
Float:17.06	Float:19.65		Text:4.35	Int:5.64			Float:13.28
Text:3.98	Float:3.46		Text:3.10	Text:4.04			Text:4.28
Text:4.28	Float:3.17		Int:0.0	Float:14.32			
	Text:1.49		Text:4.28	Text:4.28			
	Text:1.0						
	Date:11.27						
	Date:11.25						
	Date:11.27						
	Text:3.83						
	Text:3.80						
	Text:4.28						
Total:7.55	Total:8.37	Total:3.81	Total:9.06	Total:6.80	Total:12.99	Total:3.23	Total:6.92

Table 3: Entropy's of each column and table of the TPC-H database

Reference compressions of TPCH									
	Customer	LineItem	Nation	Orders	part	partSupp	region	supplier	Total
DB-Size	28.7MB	754.82MB	2KB	185.9MB	31.3MB	118.1MB	323B	1.6MB	1.1GB
TBL	23MB	718MB	2KB	162MB	23MB	112MB	382B	1.3MB	1.01GB
BZIP2	6.47MB	135MB	1KB	29.5MB	3.26MB	17.5MB	409B	426KB	192MB
LZMA	6.96MB	152MB	1KB	34MB	4.22MB	20.7MB	378B	465KB	218MB

Table 4: Benchmark size and the reference compression with 7Zip

method outperforms LZMA in achieving a more favorable compression result for the entire database.

The large amount of different data-characteristic and the dynamically select-able size of the database makes this a good benchmark for the compression.

3.3 Weather-Database

The weather database contains weather data from different US cities. There are many small tables freely available, each containing the data of a single weather station. Each table consists of multiple columns. Since some of the columns are left blank these are deleted and result in a table of 35 columns. The single tables are with 5MB to 6MB quite smaller than in the benchmarks before. Furthermore the data-set is not having that variety of data-characteristic nor many different data types. The tables only containing Text, Float, Int and Datetime. The size of the final database can be defined by the number of weather stations to be in the data-set, as well as the time span of the recorded data. This benchmark can be used to test the compression of text with many numbers. The most columns containing text with just small values per cell. Often some numbers followed by a few letters. This characteristic differs quite a lot from the previous databases where the text mainly consists of letters without numbers [11].

The entropy table 5 shows that the most used data type is Text in this database. The entropy's of the columns also differ quite a lot. Some of the columns of each table reaches an entropy of 0 since the values never change over the complete data-set. For example station-number, latitude and longitude are the same for each data entry in the table. The interesting part of the benchmark are the the text columns with many numbers in the cells.

The size of the database is with 24.8MB in the exported version and 38.8MB in the db size smaller than the previous described benchmarks 6. This is due to the small tables and the less number of tables used. However the selected tables are enough to test the behaviour of the compression, since the data-characteristic of each table is similar to each other like the entropy shows. Once again some compression references are created with BZIP2 and LZMA. Where BZIP2 achieve a slightly better compression result.

Entropy of weather database				
Bristol	FAYETTEVILLE	Fort-Smith	Nashville	Norfolk
Int:0.00	Int:0.00	Int:0.00	Int:0.00	Int:0.00
Datetime:13.40	Datetime:13.55	Datetime:13.81	Datetime:13.76	Datetime:13.80
Float:0.00	Float:0.00	Float:0.00	Float:0.00	Float:0.00
Float:0.00	Float:0.00	Float:0.00	Float:0.00	Float:0.00
Float:0.00	Float:0.00	Float:0.00	Float:0.00	Text:2.81
Text:3.19	Text:3.40	Text:3.68	Text:3.53	Text:3.40
Text:2.25	Text:2.25	Text:2.25	Text:2.25	Text:2.25
Text:2.65	Text:2.77	Text:2.84	Text:2.74	Text:2.74
Int:0.41	Text:0.75	Text:2.26	Int:2.28	Text:2.10
Text:3.91	Text:4.01	Text:3.99	Text:4.05	Text:4.02
Float:6.06	Text:3.38	Text:3.61	Float:6.36	Text:3.52
Text:3.86	Text:4.28	Text:3.99	Text:3.77	Text:3.66
Text:2.58	Text:2.77	Text:3.00	Text:2.86	Text:2.87
Text:4.35	Text:4.40	Text:4.43	Text:4.40	Text:4.45
Text:3.92	Text:3.88	Text:4.11	Text:4.12	Text:4.08
Text:4.29	Text:4.41	Text:4.34	Text:4.30	Text:4.55
Text:3.11	Text:3.36	Text:3.83	Text:3.53	Text:3.45
Text:4.60	Text:4.59	Text:4.31	Text:4.38	Text:4.27
Text:3.15	Text:3.23	Text:3.41	Text:3.30	Text:3.27
Text:4.45	Text:4.53	Text:4.45	Text:4.43	Text:4.39
Text:3.09	Text:3.69	Text:3.53	Text:3.16	Text:3.61
Text:4.50	Text:4.56	Text:4.51	Text:4.46	Text:4.38
Text:3.36	Text:3.41	Text:3.42	Text:3.36	Text:3.52
Text:4.63	Text:4.79	Text:4.50	Text:4.62	Text:4.46
Text:3.50	Text:3.53	Text:3.08	Text:3.59	Text:3.43
Text:3.90	Text:4.34	Text:3.82	Text:4.12	Text:3.93
Text:3.72	Text:3.82	Text:3.87	Text:3.55	Text:3.57
Text:3.42	Text:4.15	Text:3.43	Text:4.00	Text:3.63
Text:3.69	Text:4.19	Text:4.31	Text:3.35	Text:3.80
Text:4.60	Text:4.66	Text:4.54	Text:4.57	Text:4.69
Text:3.83	Text:4.25	Text:4.41	Text:3.38	Text:4.13
Text:4.80	Text:4.63	Text:4.32	Text:4.70	Text:4.76
Text:3.89	Text:3.93	Text:3.55	Text:4.14	Text:4.10
Text:4.39	Text:4.13	Text:3.25	Text:4.03	Text:4.08
Text:3.58	Text:2.88	Text:3.27	Text:4.35	Text:4.01
Total:3.57	Total:3.61	Total:3.60	Total:3.70	Total:3.71

Table 5: Entropy's of each column and table of the weather database

Reference compressions of weather database						
	Bristol	FAYETTEVILLE	Fort-Smith	Nashville	Norfolk	Total
DB-Size	6.2MB	7.1MB	9MB	8.2MB	8.3MB	38.8MB
CSV	4MB	4.52MB	5.81MB	5.36MB	5.15MB	24.8MB
BZIP2	330KB	350KB	434KB	444KB	422KB	1.92MB
LZMA	407KB	419KB	502KB	520KB	499KB	2.26MB

Table 6: Benchmark size and the reference compression with 7Zip

4 Database Compression

The compression of databases can be challenging. Especially with the various data types a database can contain. Depending on the data type and data structure different compression and decompression techniques achieve the best compression ratio and decompression time. This work is focused on warehouse databases and user databases. Like the benchmarks describes in the previous section 3, each database consists of multiple tables with multiple columns of different data types.

The compression time is not taken into account, which is why the compression of the database is implemented in Python. Various possible combinations of compression algorithms are combined and the variant with the best compression ratio is selected. The results between the different compression steps are plotted to receive an overview of the performance of each algorithm. Since the target is to receive a high decompression throughput on the FPGA, the achieved compression improvement of each algorithm applied to the data needs to be analysed. If a computational intensive decompression algorithm improves the compression result only by just a few bytes, than the decompression effort can be to high to receive a better data throughput on the FPGA. In addition, resources and energy on the FPGA would be used unnecessarily for this decompression.

In the compression we analysed each column of each table separately. This gives the opportunity to achieve the best compression ratio per column by applying the best fitting compression algorithms to that data. Therefore each table is splitted into separate columns. Then the data type is determined for each column. The information about the data type is used to choose the best fitting initial compression algorithm. The choice of compression algorithms depends not only on the data type, but also on the characteristics of the data. Data with a smaller entropy having a smaller information density. Compression is used to increase the density of information. It is assumed that data with a low entropy and thus a lower information density achieve a better compression rate than data with a high entropy. This is because a lot of data is redundant and compression can save redundancy and achieve a higher information density. Whereas data with a high entropy having many unique values with less redundancy, leading to a high information density which is harder to be improved. But even if the entropy is low, the data can be unorganized in the data-set, as the arrangement of the data is not taken into account. On the opposite data with a high entropy can result in good compression ratio if the data is sorted, like consecutive increasing IDs.

To keep the decompression effort on the FPGA and the required resources low, a fixed number of bits are used for the compression of numerical values. This number only varies between the columns and is determined by bitpacking 2.3. This also saves the use of separator bits between the individual values per cell.

4.1 Integer-Compression

Integers are commonly stored in databases using a word of memory, which is 4 bytes. With this numbers between a range of -2,147,483,647 to 2,147,483,647 can be represented. Compressing the data should reduce the number of bits required. Depending on the characteristics of the data different compression algorithms and combinations can be chosen. The first compression stage can be solved by Delta-compression or Huffman. The delta compression has the advantage that further compression steps can be added to the output. With Huffman a final compression step is applied to the data.

The main target is to reduce the number of bits used to represent the data. Therefore the number of bits used to represent a single integer needs to be reduced. On FPGAs the operations for the decoding are a fulfilled in parallel on hardware. Having the same bit length for each number improves the hardware efficiency, as less logic is needed. To decoded all values with the same bit length, the largest number after Delta or Run-length compression is used. The minimal needed bits to represent the value of that number is calculated and an additional bit for the sign is added. All values of the compression result are encoded with that amount of bits 2.3. This method entails a variable bit length between the individual columns. Again leading to a higher hardware usage and more complex logic on the FPGA.

Another option would be the usage of multiple quantization steps. By defining multiple fixed bit lengths, the hardware on the FPGA could be kept simple. Therefore multiple decoders of different bit lengths would need to be implemented. Such quantization steps would also result in a worse compression ratio since the overhead of leading zeros would increase. In our tests we use the calculated bit length and fulfill bitpacking 2.3, since the compression ratio is better and so a higher decompression throughput on the FPGA can be reached.

The Huffman compression can be used as an final compression step or as just a single compression. The algorithm takes the individual digits of the integer value and decodes them with a new bit value. The bit value for each digit is defined in a Huffman tree. In the compression of integers we use the following static huffman tree 1.

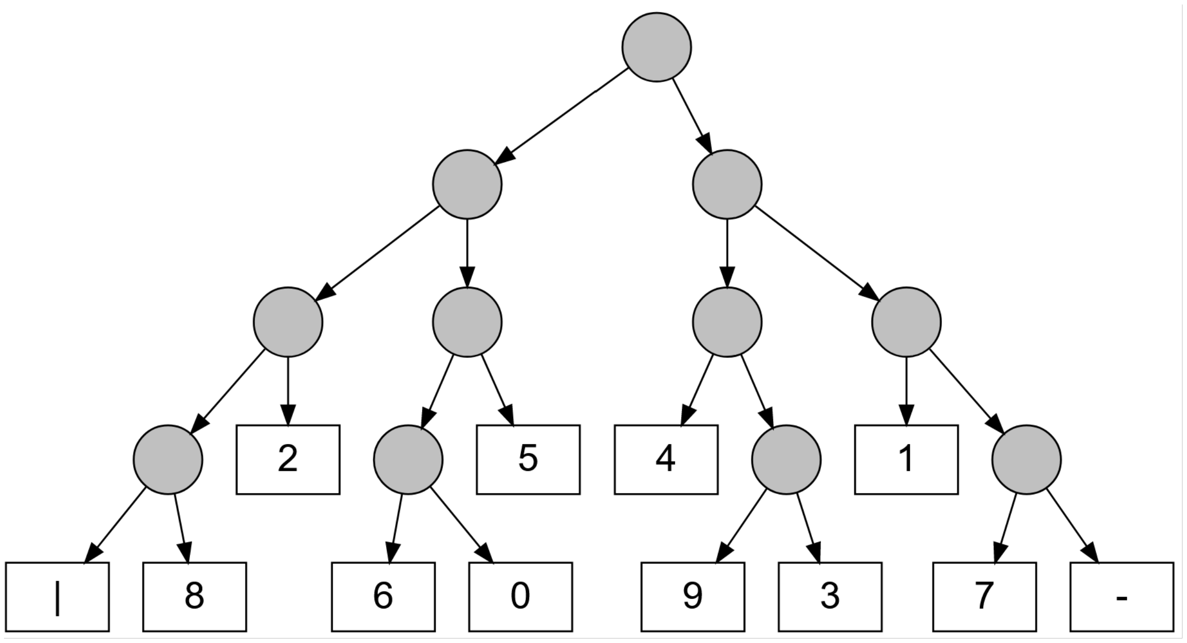
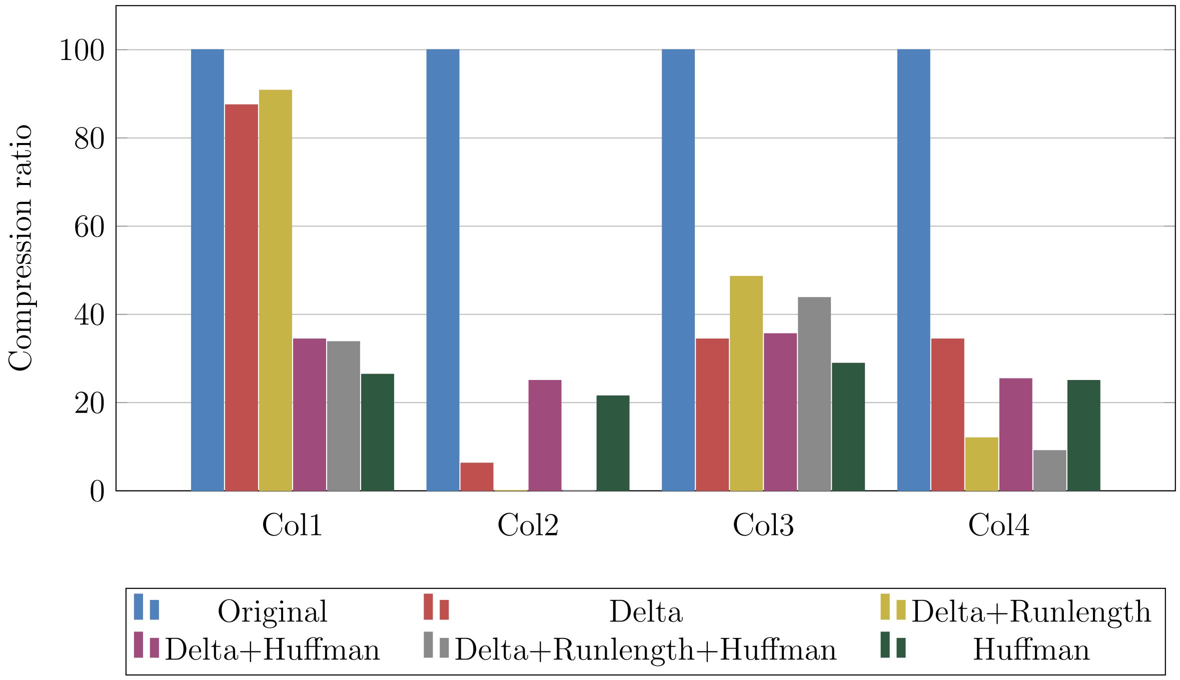


Figure 1: Static Huffman-Tree for compression of integers

This is a balanced tree with the maximal depth of 4 bits. With this tree each digit in the integer can be represented in the best case with 3 bits and in the worst case with 4 bits. By this each integer with less than 8 digits, including the sign, can be compressed by Huffman and reaches a compression result smaller than the original data. However for numbers larger than 8 digits the Huffman compression creates worse compression ratio. Furthermore for small values the Huffman compression can also generate worse compression ratios. Since with Huffman compression a dynamic bit length per number is created, an additional separator is necessary. This separator marks the end of a number and the start of a new number.

To find the best compression algorithm for integers we analysed the compression with Delta and Huffman, as well as the combination of them and an additional Run-length compression in between. It is expected to reach the best compression results with delta compression, since the data in databases are often sorted and small deltas can be reached, resulting in less needed bits.

Based on data with a small entropy and a structured arrangement, we expect a good compression rate. The small entropy is an indicator for many redundant values. If these are also structured in the data-set, a compression with Delta and Run-length should achieve the best result.



In Col1 the normalized compression ratio of VoteTypeId of the Votes table in the 'Stack Overflow' database is given. This column is having a small entropy of 3.91. The most values are varying in a range of 1 to 3, with a few outliers. Furthermore the data is structured in lists with repeating values. Contrary to the hypothesis, a poor result is achieved with Delta and Run-length. Due to the large outliers the maximal delta in the compressed data after Delta-compression is quite high, resulting in many bits needed to represent all the values. Since in Run-length the token and the length are both represented with the same amount of bits, the number of bits is doubled per tuple. Because the repeating sequences are to short and the values changing to often, the compression ratio gets worse. The break-point, at which a Run-length compression achieves better ratios is reached when the average value-change is smaller than 0,5. However, this is dependent on the value of the length not being greater than the value of the token.

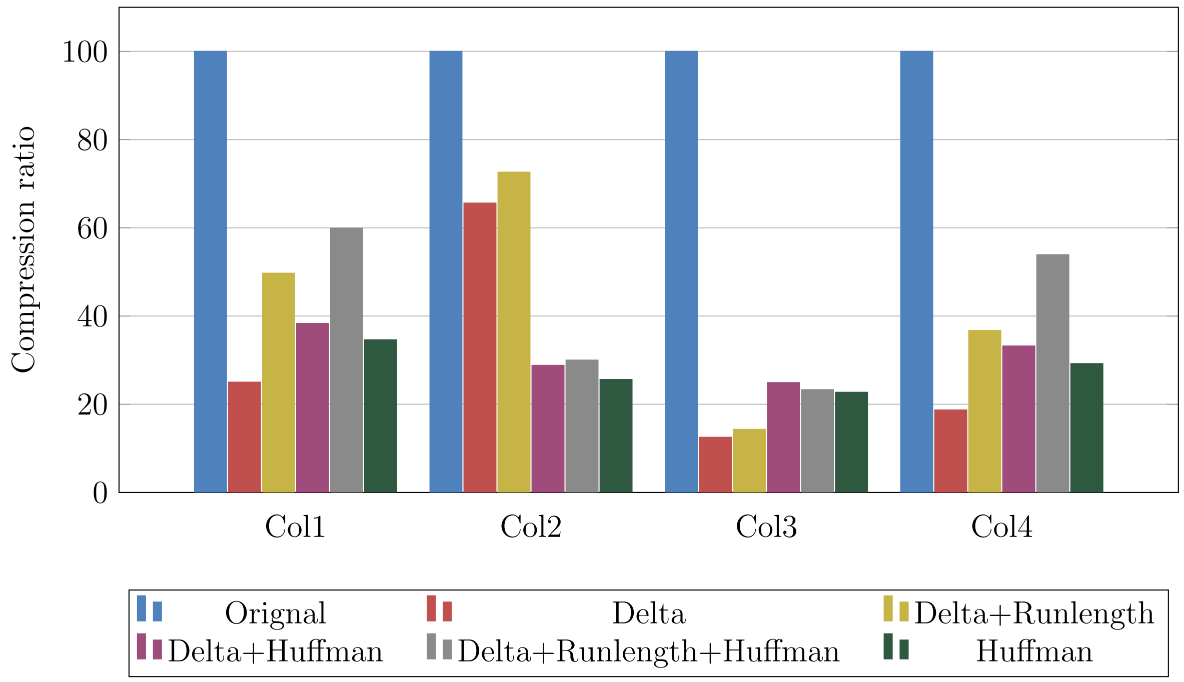
With the more dynamic Huffman compression the size can be reduce quite well, since the most values are small and only needs 3 to 4 bits to be represented.

Col2 contains the compression ratio of the column LinkTypeId in the postLinks table of the 'Stack Overflow' database. The entropy of this column is 0, which means that all values in the column have the same value. In this case the Delta compression reaches a good result, since the delta is for all values except the first one zero. With Run-length the compression ratio can be improved further. The amount of numbers in the compressed data is reduced to 4 values. 2 values for the initial value and 2 values for the token and the length. The Huffman compression on top of that can't improve that compression further. Specially the Delta + Huffman combination receive worse results, since for each value 3 to 4 bits are needed even its just an array of zeros.

Col3 and Col4 are the HourlyAltimeterSetting columns of the tables Weather-Nashville and Weather-Bristol of the Weather database. The entropy of both columns is small with 2.28(Nashville) and 0.41(Bristol). As like the two columns before the data is structured and contains longer sequences of repeating values. As the graph for Col3 shows the compression with all five methods achieve a similar result. Whereas the compression with Run-length makes the result a bit worse. This can be explained by the fact that the values change too often. In the last column the best compression ratio is achieved with Delta + Run-length and the additional Huffman. However the Huffman is improving the ratio just by a few points. The much better compression ratio compared to the Col3 can be explained by the longer sequences of repeating values.

Overall the compression of structured data with a low entropy can achieve a good compression ratio with Delta and Run-length. But it depends on the value of the outliers. Furthermore the number of changes in the in the data-set is a parameter which can lead to worse compression ratios. With a smaller entropy the chance to receive a good compression in structured data increases.

When dealing with unstructured data but low entropy the compression still should achieve a good compression ratio. But in difference to the case before the compression with Delta should reach a better compression ratio than the combination of Delta and Run-length since there are too many changes between the values. However a small entropy is not necessarily an indication of small deltas. Which can also lead to poor compression with Delta compression.



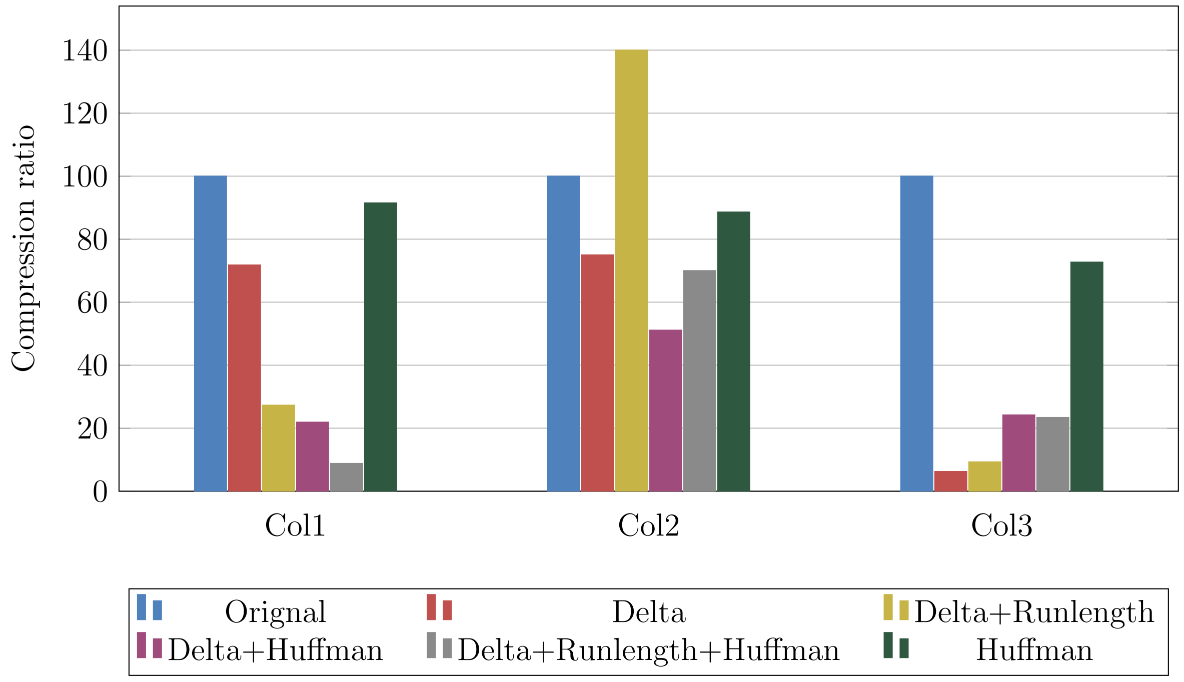
In Col1 the normalized compression ratio of the Age column in the users table of the 'Stack Overflow' database is shown. The column contains values between 6 to 100. This small amount of different values leads to the entropy of 6.57. Because the maximal delta can be ±94 and the maximal initial value 100, the number of bits needed per value drops 8 bits with sign. This results in a compressed size of 25% with Delta compression, compared to the uncompressed size. Through the disorder in the data-set the Run-length makes the ratio worse. There are to many changes in the values, leading to many tuples representing only one value. With Huffman the worst case needs 12 bits to represent the number and the additional 4 bits for the separator, resulting also in a worse compression ratio.

The second column represents the ratio of Downvotes of the users table. Since the values in that column are quite high and varying much, the deltas also keep high. Which leads to a slightly worse compression rate than within Col1. And as expected a further compression with Run-length is leading to an even worse ratio.

The last two columns in the chart showing the ratio of the fourth column of lineitem table and the fourth column of the suppliers table in the TPC-H database. They are having an entropy of 2.61 and 4.64 respectively. Like in the cases before the Delta compression achieves a better ratio then the combination with Run-length. Due to the small value in the data-set the Delta compression achieves the best result of all methods, for these two columns.

In conclusion for data-sets with unstructured data but a small entropy the Delta compression without a following Run-length achieves the best compression result. But if the values in the data having a large gaps between each other, the Huffman compression reaches a better compression ratio since less bits are needed to represent the values.

If the data-set consists of structured data with a high entropy the best compression ratio should also be achieved Delta compression. The large number of different values indicated by the high entropy leads to many changes that makes Run-length compression not profitable. Whereas the structure of the data can lead to small deltas between the values and result in a good compression.



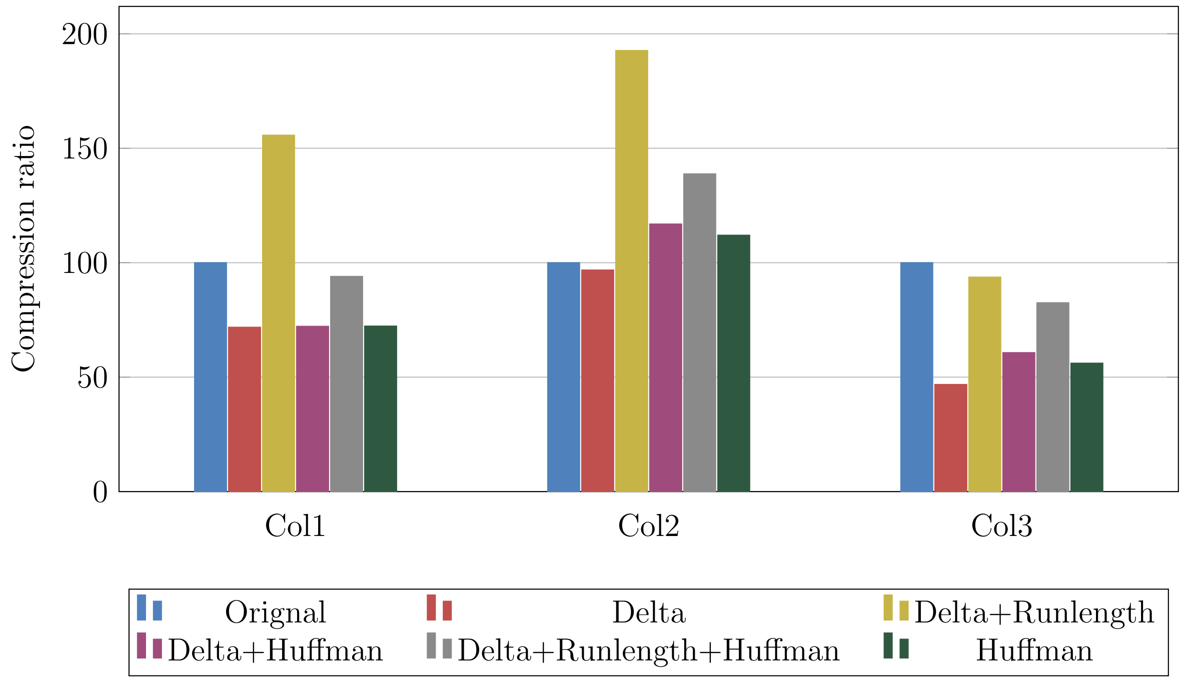
Col1 is the compression ratio of the column Id in the votes table of the 'Stack Overflow' database with an entropy of 23.2. The Ids are a list of consecutive numbers with mostly a delta of one between each other. However, there are sometimes deviations with higher deltas. Due to this outliers with high deltas, all deltas are encoded with that amount of bit, leading to a worse compression ratio. The additional Run-length improves the ratio, since the most deltas build up a long sequence of the same value. Since the outliers require many bits, the number of bits per number can be further reduced with Huffman. The Huffman compression itself gains a bad ratio, because the numbers are increasing up to 33.551.687. This means in the worst case $8 * 4 = 32$ bits for that word are needed plus additional 4 bits for the separator.

The second column represents the PostId (entropy 16.8) in PostLinks table of the 'Stack Overflow' database. As in the Col1 the numbers are in a consecutive order. But in contrast the deltas between the values are larger and varying more in their value. Like before due to some large deltas, the Delta compression achieves a bad compression ratio. Furthermore because the deltas varying much, a Run-length compression makes the ratio even worse. This kind of data is hard to compress, due to the large numbers and lack of patterns in the data. Best combination is d Delta+Huffman since the dynamic deltas can be compressed in a more efficient way with Huffman where the outliers do not have such an large impact.

The last column shows the compression ratio of Column1 (entropy 17.6) of the partsupp table of the TPC-H database. The data consists of an increasing consecutive number, where the most numbers repeat's four times before increasing. Delta compression achieve a good compression ratio, because the data-set isn't containing outliers. All deltas are between 0 and 1, leading to a bit length of 2 bits per value (including the sign). Since the most numbers are repeating four times the bit length for representing the length in a Run-length tuple becomes 4 bits. This leads to a tuple size of 8 bits. For the cases that the repeating sequence is shorter than the four, the Run-length generates a worse result. Huffman compression also needs more bits to represent the single values. In the best case 3 bits + separator are needed in the worst 4 bits + separator.

Compression of structured data with a high entropy can achieve good results with Delta compression, if the outliers are kept small. A further compression with Run-length can make sense if the deltas are not varying more often than 50% of data-set. Also the the length of the longest sequence should be smaller than the largest token. The pure Huffman compression does not work well, because most numbers having too many digits and therefore more bits are needed for the representation than the result with Delta compression. The same counts for a Huffman compression on the result of the Delta compression. Here only an improvement can be achieved if the outliers significantly increase the number of bits required and thus create a large overhead with the Delta compression.

The last possible data structure is the unstructured data with a high entropy. It is expected that the Huffman compression achieves the best compression results. Due to the many different values, the deltas are varying, what makes Run-length inefficient. Furthermore the deltas can become quite large. Therefore the Huffman compression on the raw uncompressed data should work out the best.



The compression of unstructured data with a high entropy achieves bad compression ratios. The Col1 represents the ratio of the UserId (entropy 16.6) column in the badges table of 'Stack Overflow'. The data reaches from 1 to around 300.000 and is randomly ordered. Therefore the deltas can vary in range of 300.000. Moreover no longer sequences of repeating deltas can be achieved. The large deltas leading to many bits needed to represent the values resulting in a bad compression ratio. Because after Delta compression the data doesn't contain longer sequences of the same value, the Run-length compression reach a ratio of 155. What is roughly double the size of the Delta compression result. Which means the compressed file contains more bits than the original uncompressed one. This is because each tuple represents just one value. Because many values are 7 characters (including separator) long, the Huffman compression needs in the worst case 28 bits to represent the value, which also leads to a small compression effort.

In Col2 the EmailHash of the User table is shown. It is having an entropy of 18.1 and contains a similar random structure as Col1 do. The difference is, that the numbers in that data-set are larger. This results in a similar behavior of the compression methods like before. But the compression ratios are even quite worse, since the larger numbers needs more bits to be represented. Nearly all methods generating larger compressed files than the original one.

The last column is Col3 of the lineitem table in the TPC-H database. This column has the lowest entropy of all 3 columns considered here, 13.29. The range of the randomly ordered data is just between 0 and 10.000. This makes a compression with Delta efficient, cause the deltas can kept small and a bit saving is possible. This can be seen in the result of the Delta compression. As expected with unstructured data, no longer sequences are included, which worsens the result of Run-length compression. With Huffman the worst case needs 24 bits to represent a value whereas the Delta compression just needs 15 Bits in the worst case.

Overall the best compression of integers depends on the entropy of the data as well as on the structure. If the data contains small outliers and small deltas between the values, than the Delta compression achieves a quite good compression ratio. The Run-length compression only makes sense if the number changes is less than 50% of the total amount of numbers in the data-set. However the length of the sequence also shouldn't be larger than the token. The combination of delta and run-length makes the most sense in this order, since the data structure is only slightly changed by the delta compression. A sequence of numbers $[3, 3, 4, 5, 5, 5]$ becomes $[3, 0, 1, 1, 1, 1]$ with Delta. With Delta, even the sequence of repeating values could be extended. Some Delta compression also creates the first sequence of repeating values, like in the chart 'structured data with high entropy (Col1)' shown.

A further compression with Huffman improves the ratio when a few outliers significantly increase the number of bits needed per value. The more dynamic Huffman compression can then compress the smaller values more efficiently. Like, if the fixed size requires more than 16 bits, Huffman can map the smaller ones up to 4095 more efficiently than delta. The large outliers are compressed with a worse ratio than with delta, but if they do not occur often, a better result is achieved on average.

The pure Huffman compression reaches in the most cases a similar result to the compression with Delta+Huffman. Except if the data is structured, having a high entropy.

4.2 Float-Compression

The compression of Floating-Point numbers is a bit more complex than integer compression. According to the IEEE 754 standard are floating points divided into sign, exponent and mantissa. Sign involves pre-drawing. The exponent indicates the value before the decimal point. The mantissa contains the decimal places. In figure 2 is a single precision float shown.

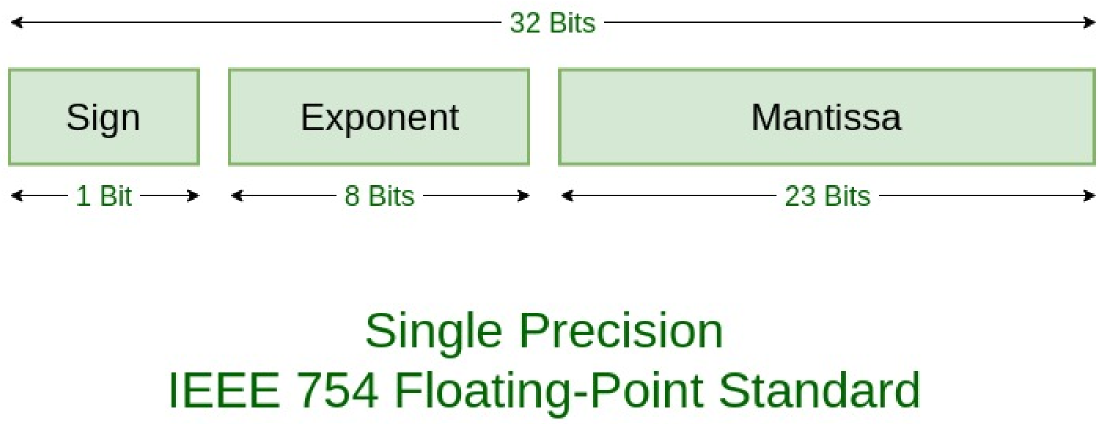


Figure 2: IEEE Floating Point Standard 754 [12]

In the single precision 32 Bits are used to represent the value. Problem with the compression of floating points is the representation always needs to be in the same precision than the original to not lose data. These means in the IEEE format the number -248.75 needs the same amount of bits like the number 2.3.

With Delta the differences between the floating point numbers can be calculated. However if the result is also a float than the IEEE format would require the same number of bits as the original data and therefore no reduction in data size is achieved. To reduce the number of bits per float we analysed three different methods and compared the result to each other.

The first method is separating the exponent and sign from the mantissa. This means we split the value at the decimal point into two values and compress them separately. This gives the opportunity to handle each separated value as an integer and can reduce the number of bits per values. In the decompression both values are set back together as one number.

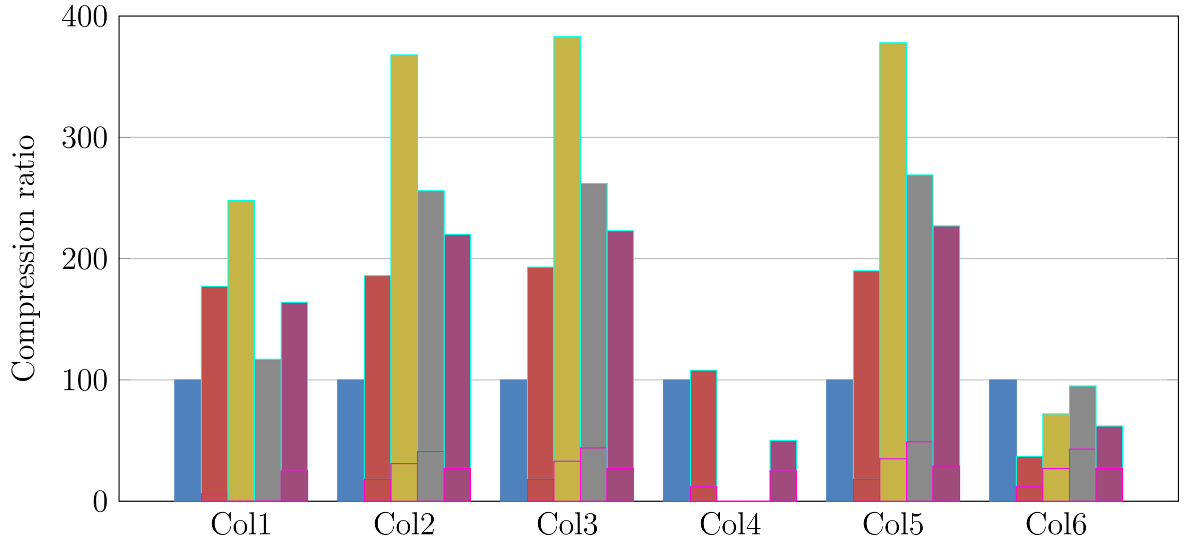
The second method shifts the decimal point to the right to receive an integer. For example the floating point number 334.56 becomes the integer 33456. The amount the decimal point that have to be shifted is depending on the largest mantissa, defined by finding the mantissa with the most characters in the data-set. Applying a uniform shift value to all floating-point numbers within a column minimizes the parameter overhead for value restoration, requiring only a single parameter to restore all values. Alternatively, each value would necessitate additional information regarding the shift, leading to increased overhead. However, a drawback of this approach is that integer numbers can potentially reach high values, demanding a greater number of bits for representation.

The last method is using Huffman compression. By using an advanced version of the static-tree (figure 1) including the decimal point, each character in the number can be compressed with maximal 4 Bits.

It is expected that the split of exponent and mantissa achieves the best result, since the resulting integer can be kept small. With the further integer compression, explained in section 4.1, the Delta compression gain the best compression ratio on that values.

We analysed the different compression methods on six different columns of the databases to get an overview of the behavior of the methods. The columns differ in the size of exponent, size of mantissa and the structure in the data-set. Col1 is a the Id column of the user table in the 'Stack Overflow' database. Containing a consecutive increasing floating point number with small deltas. The exponent increases up to a round about 1.000.000, whereas the mantissa keeps always zero. Col2 represents the fourth column of the orders table in the TPC-H database. This is unordered data with a large exponent and a small mantissa. The mantissa is just 2 characters. Col3 is the sixth column of the customer table of also of the TPC-H database. Here a small maximal 4-digit exponent followed by a 2-digit mantissa is contained. The data is like before unsorted without any pattern. The Latitude column in Weather-Bristol data-set is represented by Col4. This data consists of a long sequence of the same value, where the float is a small number. The Col5 is the Reputation column of the user table. Random unsorted values with large exponents and many fraction digits. The last column is seventh column of lineitem table in TPC-H. A static exponent of zero and a small 2 digit mantissa are contained in a unsorted manner.

In the following chart the achieved float compression ratio with a split between exponent and mantissa is represented. The color of the bars indicates the used compression methods and the thin colored line around the bar shows the share of mantissa and exponent respectively. The proportion of the compression result of the exponent is shown by the pink box. In green the share of the mantissa is marked.



The exponent and the mantissa are separated from one another and then considered separately in the compression. By extracting the exponent and mantissa components from the binary representation of the input number, a tuple consisting both values is created. So the tuple contains the normalized mantissa (fraction) and the exponent. The mantissa is always in the range $[0.5, 1.0]$. The exponent is an integer that represents the power of 2. The mathematical formula for this is:

$$number = m * 2^e$$

Where m is the mantissa and e the exponent. With a binary mantissa the base is

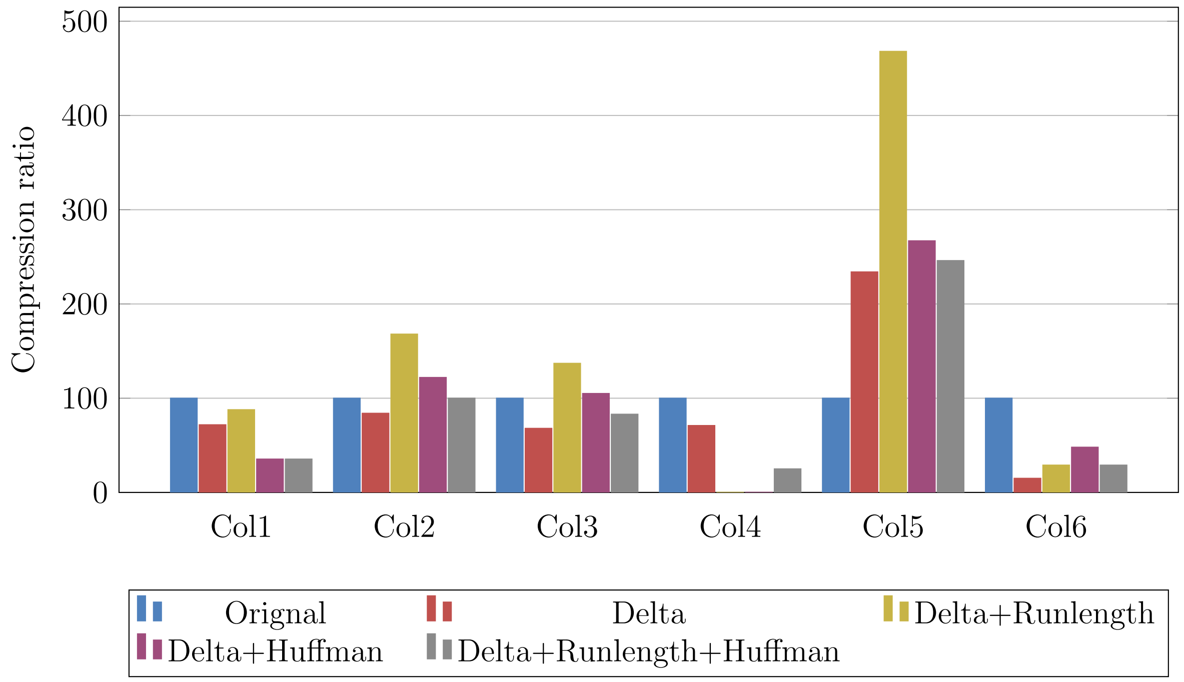
2 [13]. The mantissa is converted into integer by shifting the decimal point. To reduce the parameter overhead, the same shift is applied to all values. The fraction part with the most digits is searched for and the decimal point for all values is shifted by this number. Since the mantissa get larger even if the fraction stays the same value and only the exponent increases, the converted integer can become quite large. After both values are converted to integers, the values can be compressed like in the previous section 4.1. This leads to the results shown in the chart.

Due to the small exponents as a result of the split, the compression of the exponent works out the best with Delta compression. Whereas the mantissa makes up the largest part in the compression result. The large numbers resulting in large deltas. This has the consequence that many bits are needed for the representation. Since the separated mantissa changes with every change in exponent or mantissa mostly no longer sequences can be found, what leads to a even worse compression of the mantissa with Delta and Run-length. A further compression with Huffman improves the ratio, but still results in a worse compression ratio.

In most cases, the separate compression of exponent and mantissa results in more data than the original uncompressed data. For a static value over a longer sequence the compression with Delta results in a slightly higher bit usage but with the further Run-length compression the result can be improved, as seen in col4. With a small exponent as well as a small mantissa the compression with Delta also can achieve a good compression result. Because with the separation the mantissa stays small while the exponent do.

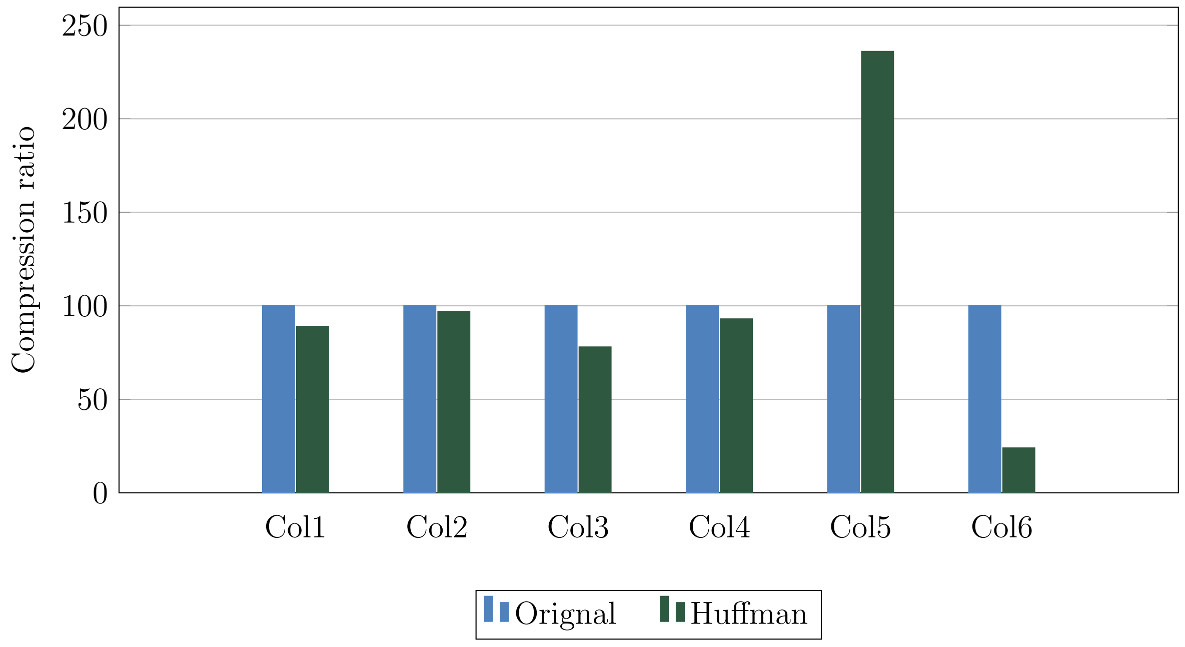
Overall the compression of floats with a separately considered compression of exponent and mantissa, leads in the most cases to a higher bit usage. Just in a view cases where the values are quite small and the deltas between them, a better compression result can be achieved.

The results of the second method to compress floats is represented in the next chart. Here the mantissa and exponent are not considered separately. The float is converted into an integer by shifting the decimal point. The number of places the floating points needs to be shifted is determined by the largest fraction part. This offers the possibility to compress the values as integer as described in section 4.1. The disadvantage of the conversion to integer is that the numbers can become very large and thus require many bits for the representation.



As the chart shows, this method reaches better compression ratios than the splitting of exponent and mantissa for the same data-sets. In nearly all cases the Delta compression reaches a positive compression ratio, since the numbers can be kept smaller than in the previous method, leading to less needed bits. The behaviour of the compression is analogous to the behaviour described in section 4.1. The large exponent and large mantissa results in a very large integer in Col5. For the representation of that integer more than 32 bits are required, resulting in a higher bit usage for the compressed values.

The last method is the compression with Huffman. The Huffman tree in figure 1 is extended by the addition of the decimal point. Then each digit of the number is compressed with Huffman like explained in the integer compression. With the new tree the worst case for representing a digit is still 4 bits whereas the best case is 3 bits. This leads to the following result.



The compression with Huffman reaches in the most cases just a slightly better bit usage. This is due to the fact that the most values at least consist of 6 to 7 characters, where in the worst case the 4 bits per character are needed, resulting in 28bit for a 7 character long float. Just for the Col6 with small values the Huffman compression reaches a good ratio.

The lossless compression of floats is challenging. The splitting of exponent and mantissa results in large numbers, requiring many bits in representation. While the exponent can be compressed with a good ratio using Delta, the mantissa compression is quite worse. The large numbers created by separating and converting to integers require many bits. Furthermore, there are often large deltas between individual values. This also makes further compression using Huffman inefficient. In general, the compression of the mantissa makes up the largest part of the result.

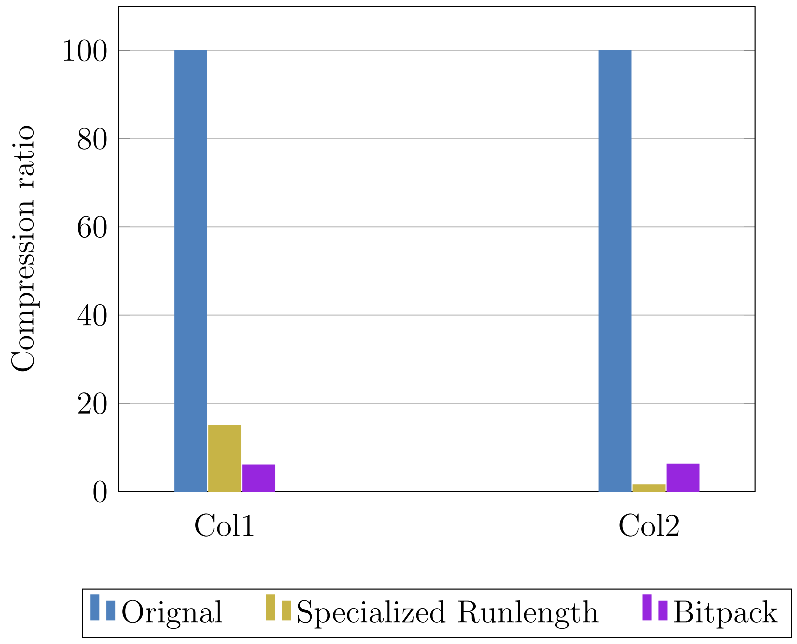
Huffman compression reaches a slightly better bit usage than the uncompressed data. However due to the decompression effort of Huffman the throughput can be lower than just passing the uncompressed data.

The delta compression after converting the float to a single integer achieves overall the best results. Where the results depend like in the integer compression before on the structure of the data as well as on the size of the values.

4.3 Boolean-Compression

The boolean datatype is quite simple, since a boolean just can be true or false. This also leads to an maximal entropy of 1. So only one bit per value is needed to represent a boolean. However, most databases use 2 bytes to store a boolean. In our work we analysed two methods to compress boolean values. The first method is simply representing each boolean with one bit and create a bitstream with that bits. The second method is a specialized version of the Run-length compression explained in section 2.1.1.

Depending on the length of the sequences of repeating values the best compression method varies. For short sequences with many changes the first method should work the best, whereas for longer data-sets with longer sequences of the same value the specialized run-length method should achieve better results.



The Col1 contains the values randomly without a pattern. There are no longer sequences of repeating values. This means that compression with bitpacking achieves a better result than Run-length compression, since for each value a tuple of two values is needed. This doubles the size of the compression result in respect to the bitpacking attempt.

The second column represents a data set where the values are in longer sequences of repeating values. Due to the many repetitions, Run-length compression achieves a better compression result. Unlike integer compression with Run-length, the break point is not 50%. Because the length of the sequence defines the amount of bits used to compress each value and not the size of the token anymore. If the length of the sequences contained in the data-set are similar, a good compression rate is achieved, since the bits generate little overhead to represent the individual values. However, if there are large differences between the sequence lengths, many overhead bits are generated for the shorter sequences. Leading to a worse compression rate.

4.4 Date/Datetime-Compression

The date and datetime datatype are widely used in databases. Especially with warehouse databases, dates are used to store the time of an action such as the receipt of an order. In the databases often the two different date types are used, depending on the required accuracy of the information. This also effects the amount of needed space in the memory to store the values.

The date type contains the values in the YYYY-MM-DD format. Leading to a maximal amount of 10 characters, 4 for the year and 2 characters for the month as well as for the day. Additional 2 characters are used to separate the values. The storing of a single date in a SQL database needs 3 bytes [14].

Datetime is the more precise format used in databases. The datetime is represented in the YYYY-MM-DD hh:mm:ss.nnn format. Due to the additional time values in that datatype, the number of bytes to store the value increases to 8 bytes [14].

For the compression of the date and datetime two different methods are analysed. The first method is using Huffman compression to encode each character in the date or datetime with a new bit value. Therefore an extended version of the Huffman tree from the integer compression is used (see Figure 3).

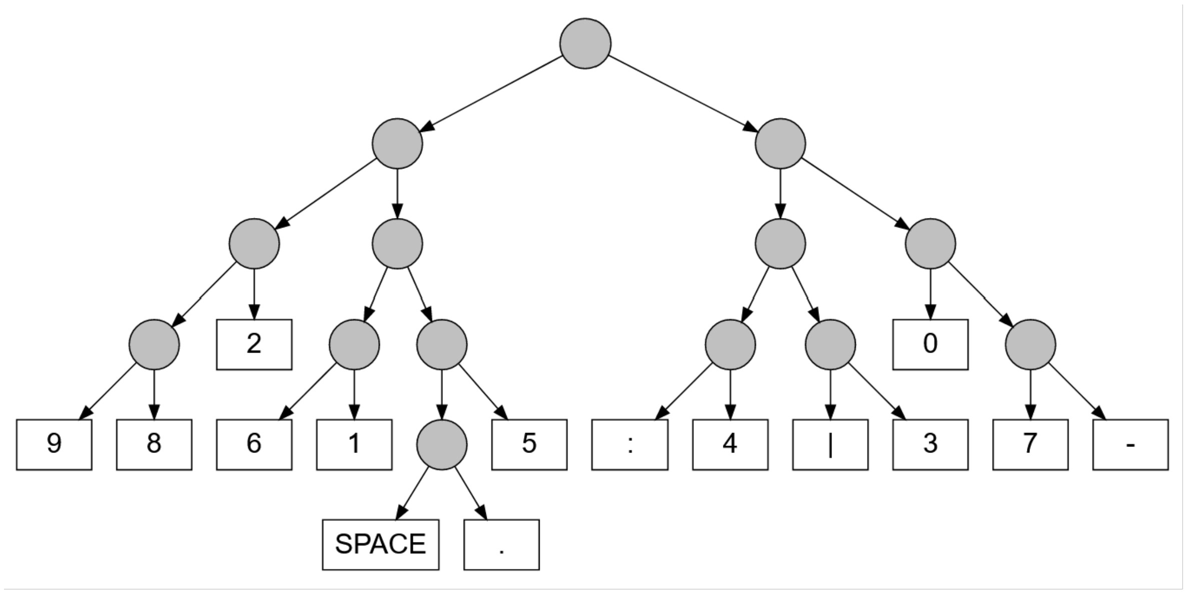


Figure 3: Static Huffman-Tree for compression of dates

The tree is structured with ’2’ and ’0’ as the most frequently occurring values at the top. This arrangement is influenced by the common occurrence of years in date formats beginning with ’2,’ often followed by ’0.’ These digits also often appear in the representation of month and day. In contrast, the characters ’Space’ and ’.’ are positioned at the lowest level since they only appear in the datetime format, and even then, just once. As the tree shows in the best case 3 bits are needed to encode a character and the worst case needs 5 bits. For the date datatype with 10 characters the best case needs also 3 bits but the in the worst case are just 4 bits needed, since the date datatype doesn’t contain ’Space’ and ’.’ characters. Leading to a compression result, for the date 2000-02-02 which is the best case value, of 32 bits. This is 8 bits larger than the original data. Therefore, the use of the static Huffman compression on dates is not appropriate.

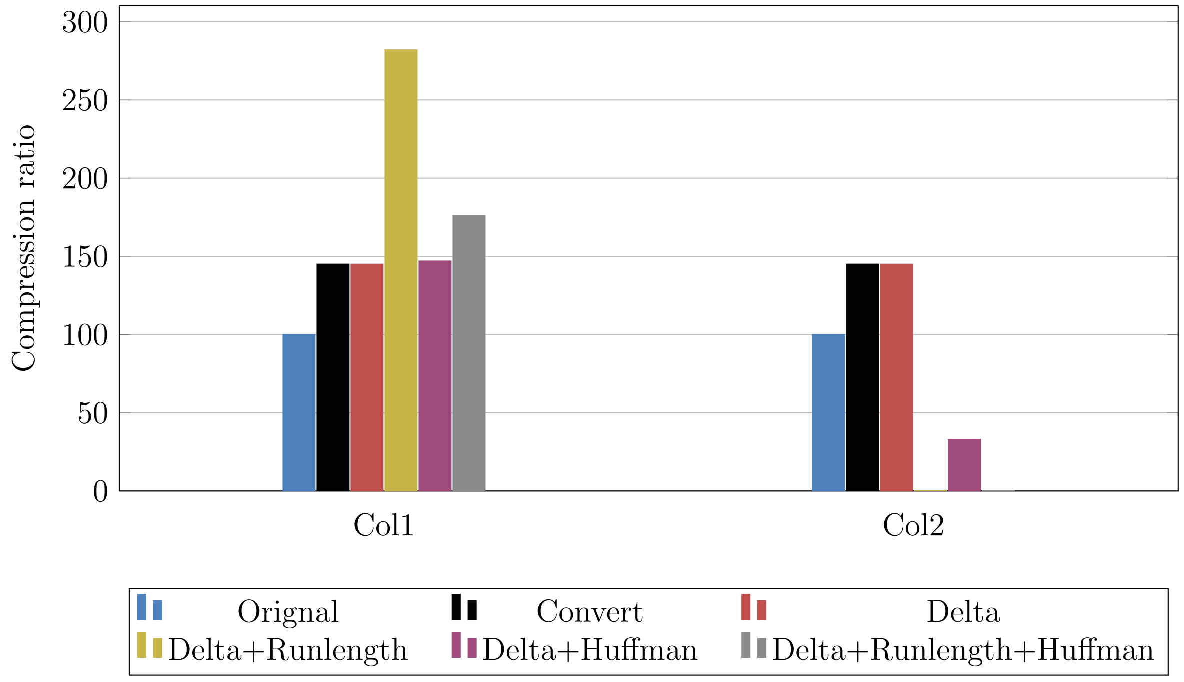
The result when compressing datetime with Huffman is the same. To encode the best case value: ”2000-02-02 00:00:00.000000000”, 95 bits are needed with that Huffman tree. Whereas the original representation just needs 64 bits.

Leading to the result that Huffman compression is not an option for date or datetime compression. However Huffman can be used as an final compression step in the second method.

The second method is to convert the date and datetime datatype into the Unix timestamp. Where the Unix timestamp gives the count of seconds from January 1st, 1970 to the given date. This number is an integer. It is also possible to get the milliseconds within that timestamp [15]. This is used to convert the datetime, but leads to a larger number needing more bit. After converting, the same compression steps like in integer compression explain can be used. The disadvantage of this method is that the timestamps are quite large numbers, needing many bits to represent the value. In delta compression, the initial value is uncompressed and typically the largest value within the data-set after delta compression. Consequently, all deltas are encoded using the number of bits required to represent that initial value. The timestamp of date is currently in the range of 1-2 billion and therefore needs 31 bits to be represented, which is 7 bits more than in the original representation. The timestamp of datetime is in the range of 1-2 trillion and needs 41 bits to be displayed. This is only about two third of the bits originally needed. A further compression with Run-length or Huffman is than possible.

Since the deltas are often having an overhead due to the large initial value a further compression with Huffman should achieve the best results, if the deltas have less than 8 digits for date respectively 10 digits for datetime on average.

The following chart shows the compression results of two columns with date format. The first column is part of the lineitem table in the TPC-H database. The data is unsorted and therefore random. The deltas between the numbers also differs. The second column shows the compression result of the votes table in the 'Stack Overflow' database. This data-set contains longer sequences of repeating values, leading to a smaller entropy.



As the chart shows the size of the converted date to an integer timestamp increases the size of the data, like explained before. Both columns in the chart reaches the same compression result for the convert. Also the further Delta compression is not improving the compression result. The compression ratio stays the same, since the first value in the Delta compression determines the required bit length.

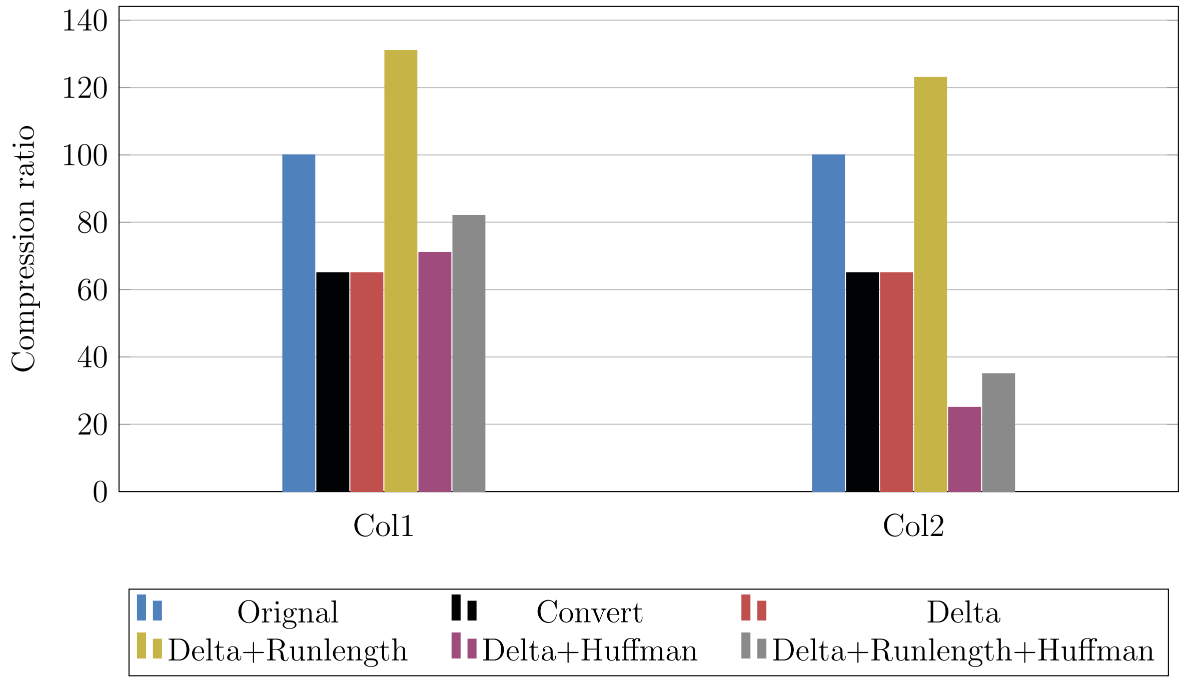
For Col1 the Run-length compression makes the ratio even worse. This was expected because the data structure in that column is random and no longer sequences of the same delta is contained. This behaviour is equivalent to them described in integer compression 4.1.

The second column achieves with the Run-length compression a much better ratio. The data-set of this column is structured and contains many long sequences of the same values, what leads to an efficient use of Run-length.

Compression with Delta and Huffman achieves in the first column a slightly worse ration than just Delta. This behaviour can be explained by the large deltas between the values. The numbers have on average a little more than 8 digits. The same counts for the Delta+Run-length and Huffman compression. The compression of the values is equivalent to the Delta+Huffman, but additionally the length of the tuple is needed to be compressed, which leads to a worse ratio.

In conclusion the compression of date is quite challenging. Due to the large amount of digits per value and the larger numbers after convert to the Unix timestamps. The compression results are the same as for integer compression. However, Delta+Huffman compression achieves in the average the best ratio over the different data structures. Even if in some cases the additional Huffman results in a slightly worse ratio. Due to the static amount of bits per value the Delta compression is not reducing the data-size. The more dynamic Huffman reduces the data-size if the limit values described earlier are complied with.

The compression results of datetime compression is presented in the next chart. In Col1 the compression of the lastAccessDate column of the user table in the 'Stack Overflow' database is given. The data-set consists of the data in datetime format and is unsorted. Leading to larger deltas between the values. The Col2 represents the Date column in the badges table. Here the data is sorted and with small deltas between the values. The entropy of both columns is high, since the most values are not repeating each other.



In difference to the compression of the date format, the compression of datetime achieves bit saving with convert and Delta compression. After the conversion to the timestamp the maximal needed bits per value is 41, which is 2 third of the original value. This can also be seen in the chart. The ratio of convert and Delta is roughly 66%. A further compression with Delta results in the same ratio, due to static bit length per value, like explain in the date compression. Also the compression with Run-length results in a worse ratio. Since there are no repeating values within the data, the size of the compressed data-set doubles.

Like before the compression with Delta and Huffman achieve in some cases a slightly worse compression result than just delta. But for sorted values with smaller deltas the Huffman compression improves the delta result, since the overhead of the delta compression with small deltas is reduced with the encoding of Huffman.

Overall the compression with Delta and Huffman after converting the dates to an integer timestamp achieves the best compression results. With Delta compression the size of the single values is reduced and with the 'dynamic' encoding of Huffman a large overhead is avoided and the compression ratio further improved.

4.5 Text-Compression

The compression of text is more dynamic than the compression of numbers. The length of the strings inside each cell of the database can vary. Furthermore, often the single characters do not repeat each other, what makes a Run-length compression inefficient [16]. Also the compression with Delta is not a valid option. The ascii code consists of 255 character where 94 characters are the standard printable characters appearing in text [17]. Assuming we look at the ascii table with only the 94 characters, 7 bits are needed to encode all the values in the table. This means that in the worst case the delta has the value 93. However, since the Delta compression can also have negative deltas, an additional bit must be used for the sign. This leads to an 8 bit coding per character. This corresponds to the same size as in the uncompressed variant. Similarly, if a Huffman is applied to the Delta compression. In the best case, the deltas are between 0 and 9. With the Huffman tree for the integer compression, this leads to a code word of length 3-4 bits. In addition, however, a separator is needed between the individual columns, which also consists of 4 bits. Thus, with Delta+Huffman compression, the same bit size per character is achieved as in the uncompressed data.

This leads to the conclusion that only with Huffman alone can a profitable compression can be achieved. The Huffman tree for the compression of text is shown in figure 4.

The tree consists of the 94 printable characters and is used in the static Huffman compression. This enables us to encode each character in the data-set with a new value. The structure of the tree is based on multiple samples of different tables of the benchmark databases. The symbols occurring the most are placed on a higher level of the tree and need less bits to be represented. Symbols with less occurrence are placed on lower levels of tree, resulting in a encoding with more bits per character. By encoding the most frequently occurring symbols with fewer bits than the original, the size of the file is reduced. The few symbols that are on a lower level of the tree and need more bits than the original ones do not appear as often and have less influence on the size. The goal is to use fewer bits on average than the original file.

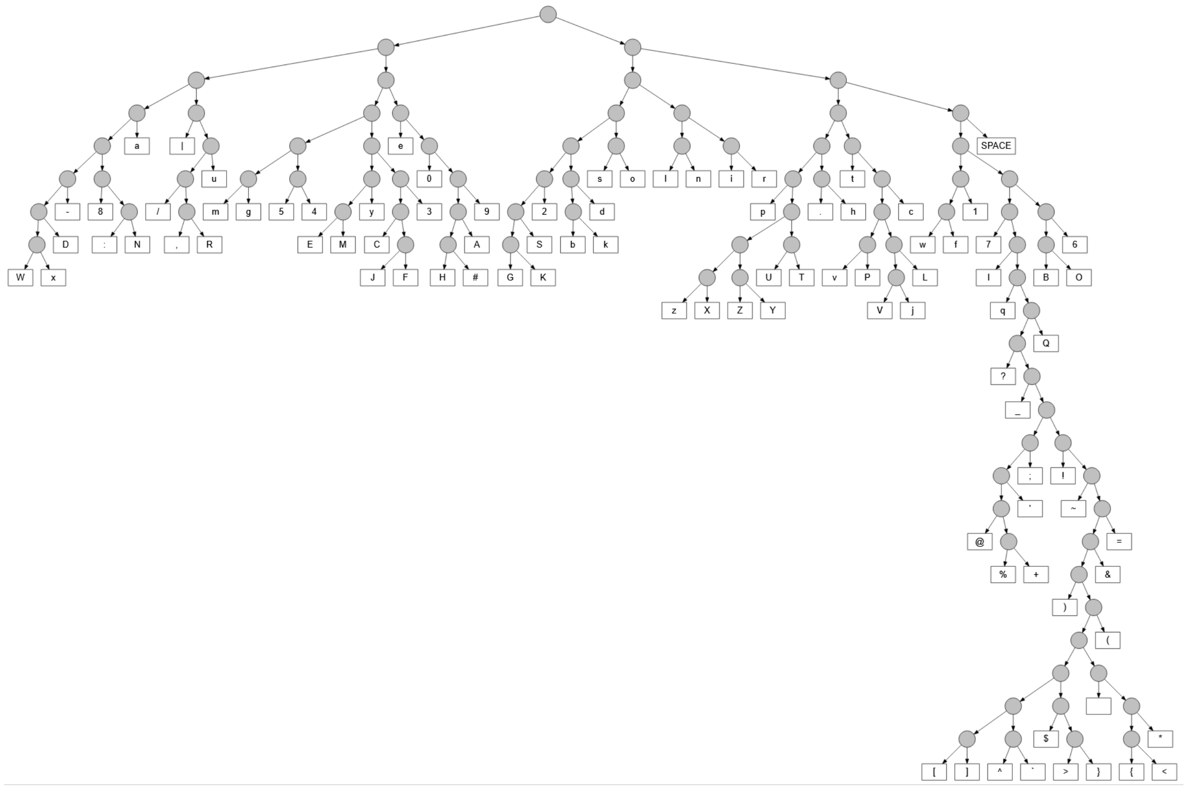
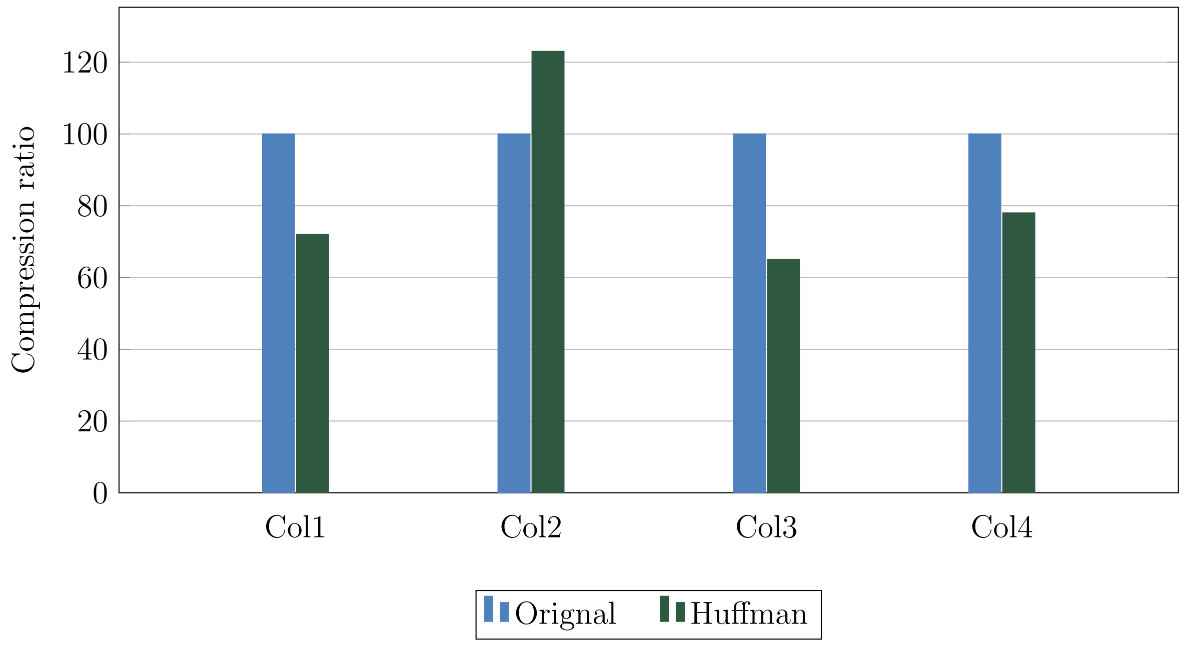


Figure 4: Static Huffman-Tree for compression of ascii values

Since the number of characters per cell differs and the compression result with Huffman also generates a dynamic compressed size. A separator like in the sections before described is necessary. This separator is needed in the decompression to mark where one cell is ending and a new one begins.

The chart shows the compression results with Huffman and the static tree from above (fig:4). Col1 represents the Location column in the user table of the 'Stack Overflow' database. The values in that column are in a random order, and consists of all different ascii symbols. Col2 also is part of the user table and shows the result of the CountryCode column compression. Here only 2 characters per cell are contained, as well as these characters are all upper case letters. The Col3 represents the second column of the part table in TPC-H database. The data only consists of letters of the lower case. Finally the Col4 is the DailyAverageRelativeHumidity column in the Weather-Fort-Smith database. This data-set consists of a combination of letters and many numbers. The entropy for all columns is around 4 to 5. This is because the maximal number of different symbols is fixed to 94.



As the compression results shows the compression with Huffman achieves in the most cases an efficient compression. Where the compressed data size is between 60% and 80% of the of the original data. In the case of Col2 the compression achieves a worse result. Because the most upper case letters need 7 to 9 bit to be encoded. On average, the same number of bits as in the original are required. There is no reduction in compression achieved. What exacerbates compression is the necessity of using separators after each cell, resulting in the need for more bits than before.

A better compression ratio could be achieve by using a dynamic Huffman tree, where the symbols are sorted optimized to each column specific. But this would lead to an larger overhead, since the compression tree also needs to be included in the compressed data. Furthermore the decompression effort on the FPGA would be larger and more resources needed.

Overall the compression with Huffman achieves a good compression result, if the data consists not only of upper case or special characters. Since the most lower case letters are on a higher level of the tree, the compression of achieves the best result.

4.6 Compression-Results

Overall, there is no one compression method that works best for all data types. But there is a pattern that for the compression of numbers (int, float) the Delta compression as the first step is the best variant. Further compression's with Run-length or Huffman then depend on the structure of the data. For the compression of float, the best result is achieved when the float value is considered as an integer. The same applies to the compression of date/date time. After converting to the Unix timestamp, a compression with Delta+Huffman achieves overall the best compression result. For the compression of text Huffman is the choice.

In some cases no compression method is able to reduce the size of the data. In these cases, the data is left uncompressed. This saves unnecessary decompression effort and resources. Also, the data throughput does not deteriorate, as the data is passed through directly without processing.

In the following tables the compression results of the complete benchmark databases are provided. The compression method achieving the best ratio is used as final compression step and the result applied to the compressed data.

Compressions of Stack Overflow								
	LinkTypes	postTypes	voteTypes	badges	votes	postLinks	users	Total
DB-Size	39B	181B	335B	36.8MB	152.2MB	3.9MB	38.3MB	232MB
CSV	30B	124B	228B	57.7MB	290MB	8.53MB	37.7	394MB
BZIP2	199B	258B	328B	9.29MB	52.6MB	1.98MB	9.63MB	78.4MB
LZMA	165B	230B	301B	8.03MB	41.0MB	1.47MB	10.1MB	64.8MB
Ours	12B	66B	118B	16MB	88.9MB	1.94MB	15.4MB	122MB

Table 7: The sizes of the Stack Overflow database before the compression and after compression with different methods

Compressions of TPCH									
	Customer	LineItem	Nation	Orders	part	partSupp	region	supplier	Total
DB-Size	28.7MB	754.82MB	2KB	185.9MB	31.3MB	118.1MB	323B	1.6MB	1.1GB
TBL	23MB	718MB	2KB	162MB	23MB	112MB	382B	1.3MB	1.01GB
BZIP2	6.47MB	135MB	1KB	29.5MB	3.26MB	17.5MB	409B	426KB	192MB
LZMA	6.96MB	152MB	1KB	34MB	4.22MB	20.7MB	378B	465KB	218MB
Ours	15.3MB	362MB	1.3KB	91.8MB	14.9MB	64.8MB	171B	884KB	550MB

Table 8: The sizes of the TPC-H database before the compression and after compression with different methods

Compressions of weather database						
	Bristol	FAYETTEVILLE	Fort-Smith	Nashville	Norfolk	Total
DB-Size	6.2MB	7.1MB	9MB	8.2MB	8.3MB	38.8MB
CSV	4MB	4.52MB	5.81MB	5.36MB	5.15MB	24.8MB
BZIP2	330KB	350KB	434KB	444KB	422KB	1.92MB
LZMA	407KB	419KB	502KB	520KB	499KB	2.26MB
Ours	2.9MB	3.4MB	4.4MB	4MB	3.9MB	18.7MB

Table 9: The sizes of the Weather database before the compression and after compression with different methods

As the tables illustrates, for small tables spanning a size of a few kilobytes, our compression method can achieve compression results comparable to those of BZIP2 and LZMA in the 7Zip compression. For the larger tables containing more data and complex structure in the columns, our compression cannot reach the compression ratio of the reference compression's. In total the compression with the methods explained above we can reduce the size of each of the database to roughly the half of its original size.

The tables also show that the database tables with a smaller entropy achieve better compression results than tables with a high entropy. The information density could thus be increased through compression. However, it should be noted that the calculation of the average entropy does not take into account the share of the column in the total size of the table. Therefore, a table with low entropy can achieve the same compression results as one with high entropy.

Additional to that compression result, a small header containing the used compression methods per column, as well as the decimal point shift in the float compressed columns is needed. But this header just needs a few bytes.

5 Database Decompression

Target of the decompression on the FPGA is to reach the highest possible throughput of data, while keeping the required resources low. The algorithms for decompression were determined based on the results of the compression. Compressions that reached poor ratios were not analyzed in decompression. Algorithms that have achieved a good ratio were implemented on the FPGA and tested for their performance in order to achieve the highest possible decompression throughput from the data that was compressed as small as possible.

In our work the compressed data is stored in a BRAM. From there, the values are read out in 32-bit words and entered into the decompression part. The decompressed data is than written to another BRAM, also in 32-Bit or 8-Bit words.

In the decompression, all the algorithms used to successfully compress the data are applied to the compressed data in reverse order. Since the last compression algorithm is always bitpacking or Huffman compression, these are the corresponding first decompression algorithms. The results of these decompression steps is than the input for the next decompression step.

All the decompression algorithms are implemented modular, so they easily can be combined, depending on the used compression methods. To maintain the order of the decompression steps and avoid loss of data between steps, each step has 'next_one_ready' as input and 'ready' as output. The 'ready' from the next decompression algorithm is connected to the input 'next_one_ready' of the previous step. Thus, the next step determines when it is ready for a new value from the previous step. This holds the current value for the time. This means that no data is lost between the steps. However, this also means that the slowest decompression algorithm in the chain determines the maximum performance. The output of the last decompression step is always 32-Bit words for numbers or 8-Bit words for characters. These are then the input for the BRAM containing the decompressed data.

The performance measurement of the decompression algorithms is carried out on the Zynq UltraScale+ ZCU106 Evaluation Platform. The individual decompression chains are considered separately and the performance is analysed for each chain. Therefore the maximum frequency and the required LUTs, FFs, IOs and BRAMs are measured. Where the maximum possible frequency with which the decompression can be run on the FPGA is calculated with the following formula.

$$Max\ Frequency\ f = \frac{1000}{T - WNS}$$

T is the target clock frequency in nanoseconds and WNS the resulting worst negative slack. With a negative WNS the maximal frequency can be calculated. The result is given in MHZ.

With the maximal frequency the decompression can run, the worst-case throughput as well as the best-case throughput is calculated. Therefore the the output 32-Bit or 8-Bit is taken as size of the decompression result. Depending on the number of cycles required to decompress a value, the throughput is calculated. The number of cycles per value varies between the individual decompression chains. The number of cycles can also vary within a decompression chain. This depends on the bitpacking decompression, or the number of digits of a number in the Huffman decompression.

$$Throughput \ = \frac{\frac{f}{c} * b}{32} / 1000000$$

f is the maximal frequency, c the needed cycles to decompress one value, b is the size of the output. As a result the throughput in MB/s is given.

5.1 Delta-Decoding

For the decoding of delta compressed integer numbers the bitpacking-decoder as well as the delta-decoder is necessary. Since each column of a database table is compressed separately, the number of bits used to represent the values varies between the columns. In order to avoid having to build an unnecessarily large number of bitpacking decoders, the decoder has an input which is used to enter the number of bits with which a number is encoded. This means that a separate decoder does not have to be implemented for every possible coding size and resources on the FPGA can be saved. The used bits to represent the numbers within a column is not varying like explained earlier. This means that it is sufficient to save only one value for decompressing the column. This also keeps the overhead for compression small. This value is then used as input for decompression during bitpacking.

The bitpacking decoder reads the compressed data in 32-bit words from the BRAM. The individual decoded values are then extracted from the 32-bit words by means of a sliding window. The sliding window has the size of the number of bits used to encode the numbers. Thus, the sliding window always contains the entire coding of a single number. After reading a value from the 32-bit input word, the window is shifted by its own size to extract a new number. If the window can no longer be completely filled with the remaining bits from the 32-bit input, the remaining bits are loaded into a buffer. Then a new 32-bit word is loaded from the BRAM. The bits from the buffer are then connected to the newly loaded bits and the sliding window is filled with this vector. The remaining bits are the MSBs and are completed with missing bits from the new value. In this way, encoding's that do not exactly fill a 32-bit word can also be decoded.

The bitpacking decoder can output a new encoded word with each clock cycle, which can then be further decompressed. When the end of the input word is reached, 3 clock cycles are needed to load a new word from the BRAM and to generate an output with the remaining bits. This can also be seen in figure 5. After the 'Input_Data' value has changed. The output becomes invalid for 3 clock cycles. In these bars, the new value is loaded in the bitpacking decoder and a new value is built for the following decompression.

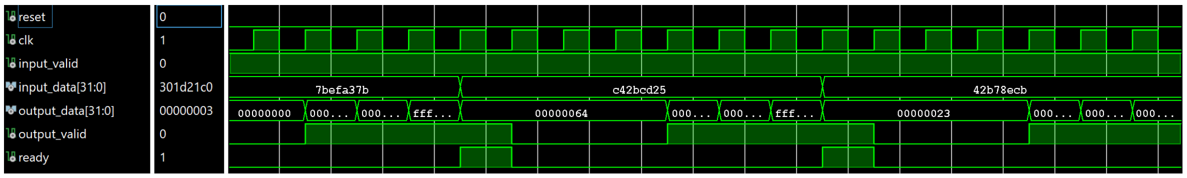


Figure 5: Bitpack-Delta Decompression with small sliding window

The figure 5 shows the input from the BRAM with the compressed data and the output of the uncompressed data, after applying the bitpacking and delta decoder to the data.

After bitpacking, the delta decompression of integer numbers is quite simple. If the current input is valid, than with each clock cycle that number is going to be added to the previous number. For the beginning the previous number is just zero and with each adding that number is going to be updated. Since compressed deltas can also be negative, the bitpacking decompression calculates the twos complement. By this the delta decompression only needs to add up the output of the bitpacking decoding.

As the table 10 show, with this decompression chain, an throughput of around 800MB/s can be achieved in the best case. This result is possible if the sliding window size of 2 bits. Meaning that each compressed number is represented with just 2 bits. By this the number of input changes to the bitpacking decoder is minimal. This leads to the fact that the 3 clocks cycles required for the change of the input only occur every 16 values. Thus, each value only needs 1.2 clocks cycles on average. Multiplied with the maximal running frequency and the output size of 32Bits that throughput can achieved.

The worst case throughput is achieved with a sliding window size of 32 bits. This means that each value needs 3 clock cycles to be decompressed. Leading to a much worse throughput of just 320MB/s, for an output size of 32 bits. The table 10 also

Delta-Decompression results on FPGA		
Max-Frequency	Worst-Case Throughput	Best-Case Throughput
240MHZ	320MB/s	800MB/s

Hardware usage					
Total Power	LUTs	FF	BRAM	IO	BUFG
0.661Watt	1528	320	2	42	1

Table 10: Delta decompression values

shows the required resources on the FPGA. The resources for this decompression chain can be kept small. With the exception of the IO ports, all resources use less than 1% of the available resources of the Zynq UltraScale+ ZCU106 Evaluation Platform. But the bitpacking decoding needs more than 1000 lookup tables more than the huffman decoder. This is due to the flexible sliding windows size, which is given per input, as in opposite to the fixed sliding window size of the Huffman decoder. The most IO ports are used to access the BRAM with the decompressed data. Therefore a 32 bit output for the data is used, as well as 4 control IOs and another 6 bits are used as an input of the sliding window size.

The decompression of floating point numbers on the FPGA is more complex than the decompression of integer numbers. During compression, the float is converted into an integer and then compressed. This means that the decompression chain for integer can also be used to decompress float values. However, the conversion from integer to float is necessary as the last step. The number of digits by which the decimal point was shifted is used for this. This must be stored in the compression header for each column. As with the sliding window, the overhead can be kept low by storing only one value, which counts for all entries in the column. The number of shifts is then used to convert the integer back into a float. However, since dividing values on the FPGA is resource intensive and very slow, only a low maximum frequency is achieved. This also reduces the throughput. Therefore, a back conversion from integer to float on the FPGA is inefficient and should be done on the remaining subsystem.

5.2 Delta-Runlength-Decoding

The decompression with delta and runlength is based on the previous explained delta decompression. But between the bitpacking decoder and the delta decompression, the runlength decompression is added. However, runlength decompression requires two input values. The length of the token as well as the token itself. In the compressed data, this is in form of a list. In the first place the length of the token occurs, followed by the token. This means that two values must always be read in order to start the decompression. Since both the tokens and the length of the tokens are encoded with the same number of bits, both values can be read out with a bitpacking decoder. The runlength decoder remains in the 'ready' state until it has received two values from the bitpacking decoder. Depending on the length of the token, the token is then output for the corresponding number of clocks cylces. In this way, the runlength decompression can output a value with each clock cycle. The valid output is interrupted when changing the input for the bitpacking decoder and when waiting for the two inputs for decoding with runlength. Therefore more than 1 clock cycle per value is needed. This can be seen in figure 6. The change of the input value in the bitpacking as well as the loading of two new inputs to the runlength decoder takes 3 clock cycles. Therefore the best throughput can be achieved with long token lengths, since less changes are needed. The remaining invalid outputs that can be seen in the figure 10 are related to the further decompression with delta. This also leads to that gap between the first decompressed value and the following once, even if the value is not changing in between.

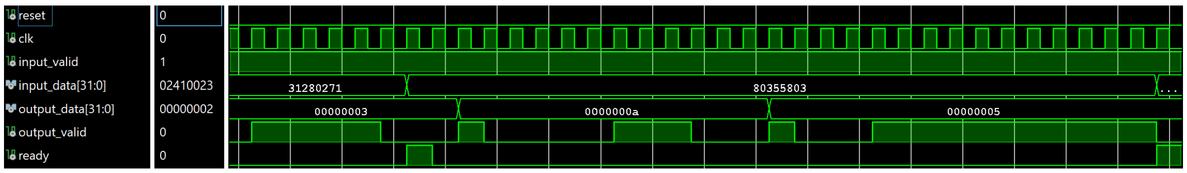


Figure 6: Bitpack-Delta-Runlength with small sliding window

Like the table 11 shows, the worst-case and the best-case decompression throughput are worse than just the decompression with bitpacking and delta 5.1. In the worst case, the run length of the individual values is only 1, so each value must be changed. In addition, the individual values are coded with 32 bits, which also makes it necessary to load new values after each word from the BRAM. Loading these two values requires three clock cycles in addition to the change of inputs for the bitpacking decoder. This means that in the worst case, 6 clock cycles are needed per value. Due to the maximum frequency of 240MHZ with which this decompression can run, a value needs 40MHZ for decompression. This results in a throughput of 160MB/s with a 32-bit output.

The longer the run length of the token the larger the decompression throughput on the FPGA. Due to a long run length, new values have to be read in less frequently. This means that the 3 bits needed for this do not occur as often. If the values are encoded with 32 bits, a new value must be read from the BRAM each time and encoded with bit packing. However, if the run length is the maximum value that can be represented with 32 bits, on average only a little more than 1 clock cycle is needed to decode the runlength of a value. With the additional delta decompression the number of needed clock cycles increases. Leading to 1.7 clock cycles per value for decoding. With the maximal frequency of 240MHZ the delta-runlength decompression can reach a maximal throughput of 560MB/s.

The maximal running frequency is therefore the same as in the delta-decompression before. But with the additional decompression method more clock cycles are needed for the decompression, leading to a worse result.

Delta-Runlength-Decompression results on FPGA		
Max-Frequency	Worst-Case Throughput	Best-Case Throughput
240MHZ	160MB/s	560MB/s

Hardware usage					
Total Power	LUTs	FF	BRAM	IO	BUFG
0.662Watt	1577	420	2	42	1

Table 11: Delta-Runlength decompression values

The additional decompression method is also reflected in the consumption of FPGA resources. Energy consumption has increased slightly. Furthermore, around 50 more LUTs are used than with just the delta decompression. The number of flip flops used has also increased. 100 more flip flops are needed here. However, the IOs and BRAM used have remained the same. Because the control is no different from delta decompression.

As with the delta decompression before, the decompression of float values is computationally and time-intensive. Here, the conversion back into float values is also carried out at the end of the decompression. This works in the same way as described above and results in a poor decompression rate. Therefore the converting back to float is more efficient on the remaining subsystem.

5.3 Huffman-Decoding

The bitpacking decoder is not needed for compression with Huffman. Huffman is always the last step in compression and the words are encoded as already described in compression. In contrast to bitpacking, the length of the encoded words varies. This means that each word is encoded with a different number of bits. In this work, the values were coded with static trees, so the overhead in compression can be kept small. In order to recognize during decoding when all values of a database cell have been decoded and a new cell begins, a separator is inserted at the end of a cell during compression.

The Huffman decompression on the FPGA is performed similarly to what Tom Wada described in his work [18]. This method is similar to the previously described Bitpacking decoding.

First, 32-bit words are read from the BRAM and a sliding window is applied to them, just like with bitpacking. However, the size of the sliding window is not entered as an input into the decompression but is a constant. The size of the sliding window is determined by the depth of the decompression tree. This means that the deeper the tree, the larger the sliding window. The word with the longest coding as well as all words with shorter coding always needs to fit into the sliding window.

Due to the different lengths of the coding, the sliding window cannot always be moved by the size of the sliding window, as is the case with bitpacking. A sliding window can contain several words, which would then be lost. Moving the sliding window therefore depends on the length of the encoding of the first occurring word in the window.

Since the values are encoded using a static tree, this does not change for the encoding of a database and can therefore be used to decode all columns in the database. The static tree therefore does not have to be present as overhead in the compressed file, but can be stored directly on the FPGA. On the FPGA, the tree is stored in look up tables (LUTs). A LUT is created for each level of the tree. This means that the maximum number of LUTs required depends on the tree depth. The LUTs are then nested within each other to create a tree structure.

In the part of the 32-bit input word extracted by the sliding window, the MSBs (most significant bit) are the encoding of the next word. The value in the sliding window is then given as input to the LUTs. The LUTs are then checking to see whether there is an entry for the input. Since the coding for the top level, for example, is only 2 bits long, only the 2 MSBs of the input are considered here and searched for entries. If no value is found here, the search is carried out in the next lower level. The 3 MSBs are then considered here. This is done until a match is found. If a value was found, the decoded value and the number of bits with which it was encoded are returned.

The decoded value is then the output. The number of bits with which the value was encoded is then used to shift the sliding window. So the MSBs in the sliding window are again the encoding of the next word.

As in the bitpacking decoding, the remaining bits of an input word from the BRAM are stored in a buffer if they are no longer sufficient to fill the sliding window. A new 32-bit is then read from the BRAM and attached to the remaining bits from the buffer. The sliding window is then applied on this.

With the nested LUTs a value can be output every clock with Huffman decompression. This can be seen in figure 7. As with bitpacking, 3 clocks are required when changing the input from the BRAM. This also means that a better decompression rate is achieved with flat trees, as new values have to be read in less frequently.

In figure 7 the decoding of text with huffman is represented. The static tree contains the 94 printable characters of the ASCII table. This means that each letter is individually encoded and therefore individually decoded. The separator at the end of the cell signals that a cell has now been decoded and a new cell begins. The static tree used here (Figure 4) is 19-bit deep, which leads to frequent loading of new values from the BRAM.

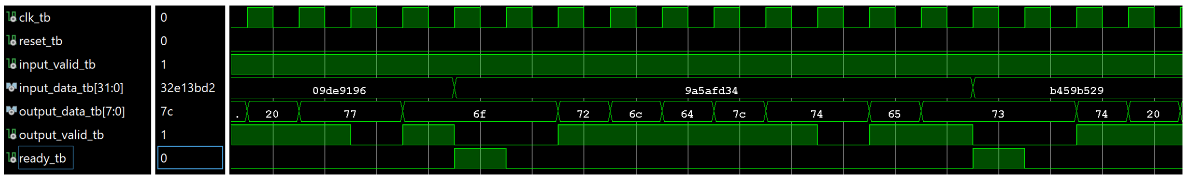


Figure 7: Huffman decoding of text

Since the tree is a maximum of 19 bits deep, in the worst case scenario a new word must be loaded from the BRAM after each coded word. This takes 3 cycles, which means that values can only be decoded every 3 cycles.

As can be seen from table 12, the Huffman decoder on the Zynq UltraScale+ ZCU106 Evaluation Platform can be clocked faster than the bitpacking decompression. This is related to the dynamic sliding window size of the bitpacking decoder. Due to the faster clocking of 420MHZ, a frequency of 140MHZ for decoding values is achieved. Since the text is output in ASCII encoding, only 8 bits are used for the encoding output. This results in a worst case throughput of 140MB/s.

In opposite, if only the coded values from the top level appear in the same tree the best case throughput is achieved. The top level characters are coded with 4 bits, which means that new values from the BRAM have to be loaded less frequently. On average, a value only needs 1.4 cycles for decoding. With a clock speed of 420MHZ, decoded values are output with a frequency of 300MHZ. With 8 bit output a throughput of 300MB/s is achieved.

Although Huffman decompression can be run faster than decompression with bitpacking, Huffman decompression with text achieves a worse throughput. This is due to the significantly smaller output size of the decompression.

Huffman-Decompression results on FPGA		
Max-Frequency	Worst-Case Throughput	Best-Case Throughput
420MHZ	140MB/s	300MB/s

Hardware usage					
Total Power	LUTs	FF	BRAM	IO	BUFG
0.617Watt	291	196	2	36	1

Table 12: Huffman decompression values

The resources required for the Huffman decoder are also lower. This means that slightly less energy is required than with bit packing. However, the biggest difference is the consumption of the LUTs. The Huffman decoder uses more than 5 times fewer LUTs than bitpacking decoding. This is due to the static sliding window size of the Huffman decoder. This also has an impact on the flip flops used. The Huffman decoder only needs about half the flip flops compared to bitpacking. The IO ports used could also be reduced. Since the decompression does not require any input for the size of the sliding window, 6 bits of input could be saved.

For the compression and decompression of numbers a smaller huffman tree is used (see figure 1). The smaller tree has the advantage that the values can be encoded with fewer bits on average. Since the tree is not as deep as the ASCII tree, a smaller sliding window can be selected. The extracted values in the sliding window are also smaller on average than in the ASCII tree. Therefore, new values have to be read from the BRAM less frequently and the number of cycles per decompression decreases.

The actual decompression then works in the same way as with text. The individual digits of the number are decoded and a separator marks the end of each number. To restore the original number, Huffman decoding is followed by digit concatenation. The existing number is multiplied by 10 and then the new decoded number is added to it. If the first character was a minus character, the two's complement is calculated at the end of the number. Only individual digits are compressed and these only range from 0-9. In order to represent all these values, the binary tree must have a depth of 4 bits. With the maximum of 4 bits, 16 values can then be represented. However, since only 10 numbers have to be represented, there are possible 6 values left to code functions without expanding the tree further. But only 2 functions are needed for decoding. On the one hand, the separator, which marks the end of a number, and the minus sign for negative numbers. The separator is coded with the bit '0000' while a negative number is marked by the leading code '1111'.

However, the digit concatenation reduces the throughput because connecting each digit requires 1 clock cycle. This means that as the length of the number increases, the required clock cycles increase and the throughput decreases.

Figure 8 shows the decompression of numbers of different lengths. As can be seen, different numbers of clock cycles are required for the different numbers until the valid output appears with the result. The output is then either the decompressed value or can be further decompressed in further decompression steps such as delta or runlength.

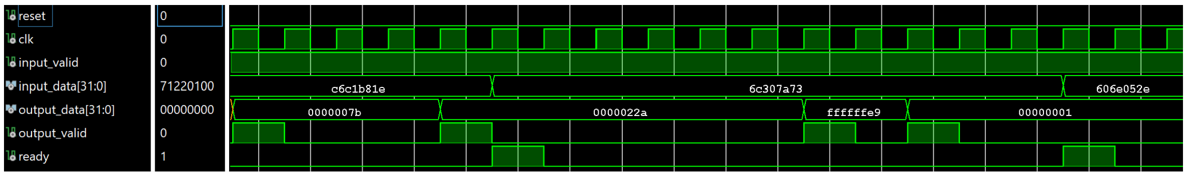


Figure 8: Huffman-Numbers decompression

As with the Huffman decompression before, this decompression can also be run at 420MHZ on the evaluation board. However, more clock cycles are needed to decompress the values than with Huffman for text.

The largest number that can be represented with 32 bits consists of 11 digits. Therefore, in the worst case, 11 cycles are needed to decode the number and then put it together. In addition, 1 clock cycle must be added to process the separator and output the decompressed number. If the number is also negative, another 3 clock cycles are needed to process the negative character and calculate the two's complement. So in the worst case, 15 clock cycles are needed to decode a value and the values are output at a frequency of 28MHZ. Assuming the output has a size of 32-bit, a worst case throughput of 112MB/s is achieved.

The best case throughput is achieved when decoding single-digit positive numbers. Here only 1 clock cycle is needed to decode the number and another cycle to process the separator. This means that a decoded value can be output every two clocks or with a frequency of 210MHZ. If the value is output with an output size of 32-bit, the best case throughput reaches 840MB/s.

Huffman-Numbers-Decompression results on FPGA		
Max-Frequency	Worst-Case Throughput	Best-Case Throughput
420MHZ	112MB/s	840MB/s

Hardware usage					
Total Power	LUTs	FF	BRAM	IO	BUFG
0.629Watt	324	265	2	36	1

Table 13: Huffman-Numbers decompression values

The Huffman for number decompression requires a similar number of resources as the Huffman for text. Only 30 LUTs and 70 more flip flops are needed for digit concatenation.

5.4 Delta-Huffman-Decoding

Delta Huffman decompression uses the delta decompression from section 5.1. Here, however, the Bitpacking decoder is replaced by Huffman decoding for numbers. The chain therefore consists of Huffman decoding, number concatenation and finally delta decompression. The throughput depends on the length of the individual numbers. The longer the number, the more cycles are needed for decompression. The bitpacking decoder, on the other hand, does not depend on the length of the number and can therefore output values at regular intervals.

The irregularity of Huffman decoding for numbers leads to a larger gap between the best case and worst case throughput. However, the subsequent delta compression does not require an additional clock cycle. Because while the number concatenation takes at least two clocks to assemble a new number, the delta can decompress the current value during this time. This works because Delta decompression only requires 1 clock cycle. This can be seen in figure 9. In the figure you can also see that the longer a number is, the more cycles are needed before the actual valid output of the number. This means that the previous number is present for longer but is not valid.

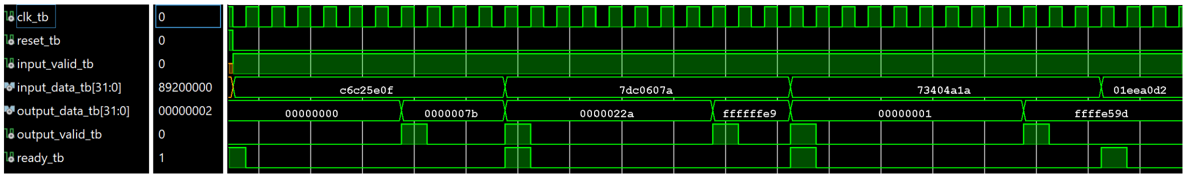


Figure 9: Huffman delta numbers

The Delta Huffman decompression can also be run on the board at 420MHZ. Since no additional clock cycle is needed for the delta decompression, 15 clocks are needed for the worst case decompression, as with pure Huffman decompression. This means that the same throughput of 112MB/s is achieved. The same applies to the best case throughput. Only 2 clock cycles are needed to decompress a value, which leads to a maximum throughput of 840MB/s. This means that you can freely choose between the compression variants huffman or delta huffman, as both achieve the same performance in decompression.

Delta-Huffman-Decompression results on FPGA		
Max-Frequency	Worst-Case Throughput	Best-Case Throughput
420MHZ	112MB/s	840MB/s

Hardware usage					
Total Power	LUTs	FF	BRAM	IO	BUFG
0.632Watt	375	299	2	36	1

Table 14: Delta-Huffman decompression values

However, decompression with delta-Huffman requires slightly more resources and energy than pure Huffman decompression. So 50 more LUTs and 30 more flip flops are needed. This still represents less than 1% of the available resources on the Zync UltraScale+ ZCU106 board, but may be important for running on smaller boards with fewer resources.

5.5 Boolean-Decoding

The last decompression algorithm considered is boolean decompression. The boolean can be compressed in two ways. On the one hand, True and False are only converted into 1-bit long values of '1' and '0'. In the second variant, the number of times a value repeats itself before it changes is counted 5.5.

In both cases, the first step is to use the bitpacking decoder. In the first case, the bitpacking decoder has a sliding window size of 1 because each value is encoded with only one bit. The extracted bit is then also the output. The output bit then represents false or true with '0' or '1' respectively.

The second variant is to decompress the boolean values with the specialized run-length. Here the bitpacking decoder has a larger sliding window. The size depends on the maximum number of repetitions of a value. The number of bits required then determines the size of the sliding window. The sliding window extracts the number of repetitions of a value. These values are then the input for the run-length decompression. The first input value of the decompression is either '1' or '0' and indicates whether the initial value is true or false. The run length of the first value is then entered into the runlength decompression. Since boolean can only assume two states, we don't have to specify the token for decompression, just a run length. This means that the compressed file does not consist of tuples with two numbers but only a list of numbers. Therefore, the special variant of the runlength only needs one input and only one value has to be extracted with bitpacking. With each new input number, the token is changed from true to false or vice versa. The new token is then valid output for the number of cycles that were entered as input. This can be seen in figure 10. This also means the longer the run length, the higher the throughput, since new values have to be read from the BRAM with bit packing less frequently or new values have to be extracted from the 32-bit word.

The change between two values takes 2 clock cycles. During this time, the new value is read and entered into the runlength decompression. In the worst case, only a

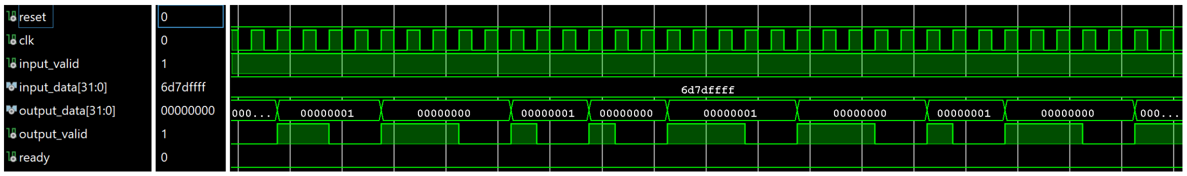


Figure 10: Bitpack-Runlength-Bool

run length of 1 is decompressed. Decompressing this run length then requires another clock cycle. Resulting in 3 clock cycles to decompress a value. Decompression with bitpacking and runlength can be carried out on the board at 300MHZ. In the worst case, decompressed values are output at a 100 MHZ clock rate. With an output size of 16-bit per value as used in most databases, a throughput of 200MB/s is achieved.

With longer run length, the throughput increases. For example, with a run length of 10, only 1.3 clock cycles per value are required for decompression. This is more than twice as fast as the worst case. A throughput of 460MB/s is achieved with an output frequency of the decompressed values of 230MHZ. However, the running length cannot be increased arbitrarily. Run lengths that require more than 32 bits for display cannot be decompressed. The sliding window can also become larger due to longer running lengths and more cycles are necessary due to the more frequent changing of the input. This is where maximum performance stagnates.

Boolean-Decompression results on FPGA		
Max-Frequency	Worst-Case Throughput	Best-Case Throughput
300MHZ	200MB/s	460MB/s

Hardware usage					
Total Power	LUTs	FF	BRAM	IO	BUFG
0.653Watt	1016	293	2	42	1

Table 15: Boolean decompression values

Decompression with bitpacking and runlength requires fewer resources than bitpacking-delta decompression. About 500 LUTs and 30 flip flops can be saved. Overall, however, this decompression requires more resources than decompression with Huffman.

6 Discussion

With the compression methods we considered, the benchmark databases could each be compressed to around half their original size. However, the baseline compression with 7Zip could not be outperformed. The time it takes to compress the data is not relevant, as our focus is to achieve the highest possible throughput for decompression on the FPGA. This means that various possible methods can be applied to the data during compression. The method with the highest compression ratio is used as the final compression. Since databases consist of many tables and the individual columns of a table are viewed separately, the best fitting compression is used for each column. Since the characteristics of the data vary greatly, there is no method that achieves the best ratio for all possibilities.

When decompressing on the FPGA, two basic models emerged, which largely determine the possible throughput. On the one hand the decompression with Bitpacking decoder and on the other hand the Huffman decoder. The two models differ mainly in the possible maximum clocking of the decompression as well as in resource consumption. While the bitpacking decoder uses more resources and has to be clocked significantly slower, the Huffman is more efficient in this respect. However, no matter what model they are based on, the compression's achieve a similar best case throughput. The worst case throughput of the decompression algorithms based on the Bitpacking model are even higher than that of the Huffman-based algorithms. This is due to the fact that with bitpacking a complete number is always extracted and then decompressed. With Huffman, however, the number must first be assembled from the individual digits before further decompression. Depending on the length of the number, this significantly reduces the throughput.

When compressing integers, bitpacking model with delta or delta runlength often achieve the best compression result. In the cases where a better result is achieved on the Huffman basis, there are a few large outliers in the data sets which increase the bits required for coding in the bit packing model. For decompression's based on bitpacking, in the worst case scenario, 3 cycles are required for delta or 6 cycles for delta runlength decompression. So with Huffman, the word can not be longer than 3 or 6 characters in order to perform better. If the numbers on average have more than this number of digits, the Huffman achieves a poorer throughput because putting the numbers together takes longer. This also includes the minus sign and the separator. This means that Huffman can only outperform the bitpacking models when it comes to integer decompression if the numbers are 1 digit (Delta) or 3 digits (Delta-Runlength) long.

Bitpacking and delta achieve the best compression ratio in almost all cases when compressing floating point numbers. Converting from floating point to integer is an important step in being able to compress the numbers. As with integers, the best decompression throughput on the FPGA is achieved by the combination of bitpacking and delta or bitpacking and delta runlength. However, the reverse conversion from integer to float on the FPGA is very expensive. It is more efficient to do this on the remaining CPU subsystem. Compressing floats with Huffman achieves a worse ratio, but with Huffman the floating point value can be restored during decompression on the FPGA. This means that no further re-conversion would have to be carried out on the remaining CPU subsystem.

Two methods were considered for compressing Boolean values. On the one hand pure bitpacking and on the other bitpacking my specialized Runlength. Both methods achieve similarly good compression ratios on the benchmarks. The bitpacking method achieves a constant throughput on the FPGA during decompression. The values in bit packing are encoded with only one bit each. A word read from the BRAM therefore contains 32 values. This corresponds to the best case for bitpacking decoding, since the fewest clocks are required per value and the BRAM is read the least. The pure bitpacking decoding method therefore achieves a throughput of 600MB/s. The bitpacking method with the special runlength has a dynamic throughput, which depends on the run length of the values. The longer these are, the higher the throughput. However, this cannot outperform the throughput of pure bitpacking, as it outputs decompressed values with a clock rate of 1.0625. Decompression with runlength, on the other hand, only achieves this value in the best case if the runlength corresponds to the largest number that can be represented with 32-bit.

Datetime is difficult to compress because it often consists of long strings of numbers and characters. For compression with Delta, the values were converted into integer timestamps. This resulted in large numbers with many digits. This means that compression with delta needs more bits than the original encoding. A compression effort can only be achieved if there are sequences of identical datetimes in the data set and these can be compressed to a few values using runlength. Since the timestamps of dates do not have as many digits as datetime, bits can be saved when compressing dates by converting them to integer timestamps. Further compression with Delta does not bring any further improvement because the timestamps all have a similar size and the initial value therefore already specifies a large number of bits with which the values are encoded. Pure Huffman compression does not provide any benefit, as the encoding of the individual values requires more bits than the original encoding. However, delta with Huffman can achieve a good ratio in some cases. If a few large deltas push up the number of bits for encoding, but most deltas are small numbers, Huffman can save more bits on average.

The decompression of date and datetime on the FPGA is then analogous to integer decompression. Decompression's based on the bitpacking model achieve higher throughput for large number values. While Huffman based decompression achieves better throughput at small values. However, since most deltas are larger than 3 digits even after compression, Huffman based decompression delivers poorer throughput. However, it should also be noted that, as with float decompression, only the integer values of the timestamps are decompressed on the FPGA. The conversion back to dates or datetime should also be carried out on the remaining CPU subsystem. Because this can make the conversion more efficient.

When compressing the text, only the Huffman was considered as a method. With the static Huffman tree used, different compression rates could be achieved depending on the characteristics of the data. The compressed data is on average compressed to 60-70% of the original size. When decompressing on the FPGA, the throughput depends on the position of the symbols in the tree. The higher the symbols are in the tree, the higher the throughput. These symbols are encoded with fewer bits. This means that more symbols can be contained in a 32-bit word. This reduces the average clock rate that a symbol needs to be decoded. Because loading a new 32-bit word from the BRAM takes 3 cycles. With long encodings, a new word has to be loaded more often, which increases the average clock rate per decoding. The static tree should therefore be balanced and not contain too deep coding. However, good throughput can be achieved with an unbalanced tree if there are only a few symbols deep in the tree and they occur rarely.

If resources need to be saved, good throughput can be achieved with Huffman-based decompression. However, in most cases compression can only achieve a worse compression ratio than the bitpacking method.

The decompression of the individual columns requires a header, which indicates which column was compressed using which compression method. In addition, the header must contain how many bits the words were encoded in bitpacking. For floating pointing numbers, the header must contain the position of the decimal point. Therefore, when compressing a table, a few bytes of overhead are needed to configure the decompression.

This header must be read on the FPGA in order to decode the individual columns with the correct decode.

Furthermore, by reading longer words from the BRAM, the decompression throughput could be further increased, as new values have to be read in less frequently.

7 Conclusion

References

[1] F. Tenzer, "Volumen der jährlich generierten/replizierten digitalen datenmenge weltweit von 2010 bis 2022 und prognose bis 2027," Available at https://de.statista.com/statistik/daten/studie/267974/umfrage/prognose-zum-weltweit-generierten-datenvolumen/ (2023/07/26).

[2] X. Z. Q. L. Haoliang Tan, Zhiyuan Zhang and W. Xia, "Exploring the potential of fast delta encoding: Marching to a higher compression ratio," *IEEE International Conference on Cluster Computing (CLUSTER)*, 2020.

[3] M. A. Roth and S. J. V. Horn, "Database compression," *ACM SIGMOD Record*, vol. 22, 1993.

[4] C. B. Dirk Koch and J. Teich, "Hardware decompression techniques for fpga-based embedded systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, 2009.

[5] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 1st ed. California Technical Publishing, 1997.

[6] D. H. A. K. W. L. Nusrat Jahan Lisa, Tuan Duy Anh Nguyen, "High-throughput bitpacking compression," *Euromicro Conference on Digital System Design (DSD)*, vol. 22, pp. 643–646, 2019.

[7] I. Schnell, "Huffman coding in python using bitarray," Available at http://ilan.schnell-web.net/prog/huffman/ (2019/04/03).

[8] S. Exchange, "Stack exchange data dump," Available at https://archive.org/details/stackexchange (2014/01/14).

[9] B. Ozar, "How to download the stack overflow database," Available at https://www.brentozar.com/archive/2015/10/how-to-download-the-stack-overflow-database-via-bittorrent/ (2015/10/14).

[10] TPC, "Tpc-h - homepage," Available at https://www.tpc.org/tpch/default5.asp (2023/10/03).

[11] N. C. for Environmental Information, "U.s. local climatological data (lcd)," Available at https://www.ncei.noaa.gov/access/search/data-search/local-climatological-data?pageNum=2 (2023/10/03).

[12] Ayusharma, "Ieee standard 754 floating point numbers," Available at https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/ (2020/03/16).

[13] D. Ibrahim, *SD Card Projects Using the PIC Microcontroller*, 1st ed. Butterworth-Heinemann Ltd. 313 Washington St. Newton, MA,United States, 2010.

[14] MySQL, "Data type sizes in mysql," Available at https://dev.mysql.com/doc/refman/8.0/en/storage-requirements.html (2023/10/05).

[15] D. Tools, "Unix timestamp," Available at https://www.unixtimestamp.com/ (2023/10/05).

[16] D. Salomon, *Data Compression*, 3rd ed. New York, US: Springer Verlag, 2004.

[17] "Ascii table , ascii codes," Available at https://theasciicode.com.ar/ (2023/10/05).

[18] T. Wada, "Variable-length decoder for static huffman code (version 1.0)," Available at https://ie.u-ryukyu.ac.jp/~wada/design03/spec_e.html (2023/10/25).