Heidelberg University

Student research project

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

Klemens Korherr

November 15, 2023

supervised by
Prof. Dr. Dirk Koch

Abstract

This work presents various methods for compressing databases in order to then decompress them with the highest possible throughput on an FPGA. The methods were analysed for compression ratio, resource expenditure and decompression throughput. For this purpose, three different benchmarks were used and the compression results, achieved by the examined methods were compared with a baseline compression (7ZIP). With the compression methods used, the benchmarks could be reduced to half the original size. By using different compression methods, several decompression combinations were also examined on the FPGA. Some of the examined decompression methods achieve a throughput of 800 megabytes per second under optimal conditions, while only requiring 1500LUTs.

Contents

1 Introduction

In 2020, 64.2 Zettabytes of data were generated and stored in databases. By 2025, this figure is expected to reach 181 Zettabytes [1]. This exponential growth in data can be attributed to various sectors, such as internet usage, social media, research, and financial markets, all contributing significantly to the digital data landscape [2]. These sectors amass massive databases, each containing millions of data sets. It is not the computing tasks themselves that are the most energy-intensive, but the transfer and filtering of data in 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 a positive effect on the energy consumption. Compressing data also increases the amount of data that can be stored while saving throughput on the communication channels. This means that the channels can be loaded with filtered data and the movement of raw data is reduced. IBM has implemented this approach with Netezza. They reduced the IO bottleneck by using an FPGA close to the SQL, which also improves IO performance. [3]

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 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. IBM's Netezza uses fast engines, which run in parallel on the FPGAs, decompress and filter out 95-98% of the table data and only keep data relevant to the query. This reduces the load on the IO ports and the remaining CPU subsystem only processes the relevant data. [4]

This paper discusses, which (de-)compression methods are the best methods to compress and decompress databases. The methods were investigated with respect to the capability to be implemented on FPGAs for delivering high throughput requirements. Therefore we analyse elementary compression methods. We investigated the compression ratio as well as the decompression throughput and the required resources on the FPGA. This gives a better understanding of the (de-)compression requirements of different workloads and the FPGA implementation cost performance-tradeoffs for different tailored compression methods.

Three databases were used to analyse the compression methods. These databases contain 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 TPC-H. That database represents a warehouse and is filled with randomly generated data. This database used as a benchmark is 1GB in size. The second database corresponds to the 'Stackoverflow' database. In this database the data is related to real world data. By changing and adding some data types, we increase the variety of data types and were able build a more complex database. This

database is with 411MB smaller than the first one. As a last benchmark, a small weather database provided by the 'National Centers for Environmental Information' is used. This database contains small tables with around 6-9MB, filled with weather information from different weather stations.

This work examines the four compression methods: Run-length, Delta-Encoding, Bitpacking and Huffman. These are commonly used compression methods that are suitable for implementation 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 achieve a better compression ratio. The main focus is on the fast decompression of the data on the FPGA in order to generate a high throughput.

2 (De-)Compression Methods

The primary objective of data compression is to minimise the number of bits required to represent a set of symbols or numerical values. 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. In addition, this approach allows more information to be transferred between systems without the need to expand the existing bandwidth, and power consumption can be kept low.

However, it is important to note that the implementation of compression and decompression comes with computational overhead. Compressing and decompressing data can be very computationally intensive. The associated costs vary depending on the specific methods employed. Some methods may incur high computational expenses during compression but are efficient during decompression. This is particularly advantageous for static data that does not change. This is because the data can be compressed once and stored in the memory without time and performance requirements. The remaining CPU system can than be supplied with a high throughput of data due to the efficient decompression.

In this research we have used widely known lossless compression methods. One of these methods is Run-length compression, which reduces the size of the database by substituting symbols in a sequence within a "run", consequently reducing the number of symbols needed to represent the data. Additionally, we employed compression methods like Delta, Bitpacking, and Huffman, all of which work by reducing the number of bits required to represent a symbol. It is possible to combine several compression methods. Combining different methods increases the required compression and decompression time. The resource consumption also increases as a result.

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

Our approach is similar to that of IBM Netezza [3]. The data is stored in a compressed format in the memory close to the processing unit (FPGA). By implementing the straightforward decompression methods mentioned earlier on the FPGA, our aim is to maximise bandwidth utilization and maintain constant data flow through the channels with valuable information. The decompressed data can then 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 important metrics used to evaluate the effectiveness of a data compression method. 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 method. The ratio is a number between 0 and 1. A small ratio indicates an effective compression. It is calculated like in the formula below. [5]

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

- decompression time: The time needed to decompress 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 put out, the decompression time can be calculated. The compression time is not investigated, since the focus of this work is on the fast decompression.

- throughput: The throughput denotes 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 dividing the size of the decompressed data with the decompression time. The throughput metric is bytes per second.

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

We decided to use the methods described in the following. These are suitable to be implemented on FPGAs and can reach a high throughput. Furthermore the small hardware usage of these methods is also an advantage and the methods can easily be combined to receive better compression ratios.

2.1 Run-length

Run-length encoding (RLE) is a lossless data compression method 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 'run'. A run 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 runs: (2,A), (5,B) (4,C) and (1,D). By this the data can be compressed from 12 symbols to 8 symbols, which is a compression ratio of 66%.

The compression ratio improves with the length of the runs. Longer runs allow a better compression, resulting in better compression ratios. Conversely, when dealing with short runs, the compression may yield a less favourable ratio, requiring more

symbols to represent the data compared to its original form. This is due to the fact that each compressed run consists of two symbols. In our approach, the token and the length of the word are added directly to each other without separators during encoding. This has the advantage of a simple decoding implementation on the FPGA, needing less resources. In addition, compression can be profitable from a run length $>$ 2. However, with this compression method, the length of the coding of the token and the run length must be the same. Else this leads to a large overhead in compression if the maximum value of the token and the maximum value of the run length have a large difference, which reduces the compression rate. It is also not possible to compress sequences without the run length value, as it is not possible to recognise whether the current word is a token or a run length during decompression.

As a result, the best achievable compression ratio of this approach 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 run and represents the best case $|t| * l$ bits. Where l is the length of the run. In the worst-case the compression ratio is calculated by [6]:

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

In the worst-case the run 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. [6]

RLE is considered a simple and straightforward compression method, 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. For example, Run-length compression is used when compressing image files into PCX image format, where a good compression ratio is achieved through the many repeating pixel values [7].

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 compression, as described in the Section 4.

Run-length decompression is suitable to be implemented on FPGAs and requires few resources. As will be shown later, decompression on an FPGA can also achieve high performance.

2.1.1 Run-length for Boolean

Boolean values can be effectively compressed using a specialised 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 1s and 0s. 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 were able to reduced the 7 symbols to 5 numbers. Where the first number states with 1 or 0 the initial state true or false of the first symbol in the sequence.

2.2 Delta-Encoding

Delta encoding is not a really a compression method, as no bits can be saved by the encoding. However Delta is used to reduce the value of elements by encoding the difference between consecutive elements in a sequence rather than encoding each number independently. This method calculates and represents the changes (deltas) between adjacent elements in the sequence and stores the delta values instead of each element in its original form [8]. The result of the delta encoding is then compressed using compression methods such as Bitpacking, Huffman or Run-length.

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 can 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 represented. 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 60% of the size 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. For example, Delta encoding can be used for data compression when the values in the original data are smooth, such as audio signals. Here, Delta encoding reduces the amplitude of the signal compared to the original signal. Delta encoding increases the probability that the value of each sample will be close to zero. If the

original signal does not change or changes in a straight line, Delta encoding results in sample sequences with the same value. Accordingly, Delta encoding followed by Huffman and/or Run-length compression is a common signal compression strategy. [8]

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 this 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 such cases, an escape sign can be used. This sign ensures that the encoding result is not larger than the original data. The larger values are encoded uncompressed and the following deltas are calculated starting from this value. This prevents the data from being enlarged during compression and a better compression ratio is achieved as the overhead is kept low.

Delta compression in this work was analysed without the use of an escape signal. The escape sign can lead to an increase in data size if there are many large deltas in the data set. As a result, both the uncompressed value and the escape sign must be encoded, which means higher bit consumption. This also leads to a more complex decoding on the FPGA, which requires more resources.

In summary, our 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. In this work, the combination of Delta encoding and the minimum bits required to represent all deltas is termed 'Delta compression'.

2.3 Bitpacking

Bitpacking is a common data compression method that optimises the storage or transfer of data by efficiently utilizing the binary representation of the symbols or numbers to be transmitted/stored. It is particularly effective when dealing with data-sets containing values that require fewer bits than the standard word size of a computer system. [9]

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. [9]

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 analysing the sequence to identify the largest number within it. The goal is to determine the minimum number of bits required to represent the 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. [9]

2.4 Huffman

Huffman encoding achieves lossless compression by assigning unique codes to symbols. To accomplish this, a dictionary is used, containing the symbols to be compressed. Each symbol in this dictionary is assigned a unique code with varying lengths. The symbols are structured in a binary tree, with more frequently occurring symbols positioned higher in the tree, requiring fewer bits for representation. Conversely, less frequent symbols are placed in the lower tree levels, potentially requiring more bits for representation than the uncompressed original data. The path to the symbol in the binary tree specifies the coding of the symbol. As new symbols are added, the tree expands accordingly. [10]

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 the data. The huffman tree is not formed on the basis of the data to be compressed. Instead, the symbols are placed in the tree based on their probability of occurrence. To determine the probability, for example, a text can be analysed which comes close to the texts to be compressed later. All symbols that occur during compression must be taken into account when building 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 optimised for the current input data.

2.4.2 Dynamic-Huffman

The dynamic Huffman compression creates the binary tree while compressing the data. The method 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 methods 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.

However, due to data security, columns are left blank in some tables. This does not correspond to reality. In order to have benchmarks available that are as close to reality as possible, these columns were filled with randomly generated data.

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 employed multiple tables and databases with diverse data content. To get an overview of the data characteristic in the tables of the databases, we calculated the Shannon entropy:

$$H(x) = \sum_{i=1}^{n} p(xi) * log_2(p(xi))$$

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. The entropy is smaller with higher probability. 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 creates a low 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 [11] [12]. This means that the database includes actual data and real data structures, making it an ideal benchmark for testing the compression methods on a real-world database. The database comprises various tables, including votes, users, posts, post links, badges etc. The data characteristics are repeated in the tables, which is why not all tables from the database are required for testing the compression methods. For our purposes, we have chosen to utilise the following tables: badges, users, link types, post links, post types, votes, and vote types. These database tables were selected because they contain all the data types we require. Furthermore, the tables consist of columns with different data characteristics and represent a wide spectrum of data that occurs in the real world. 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. This leads to empty columns in the data-set. To receive a complete database without empty columns, the columns are filled with random values. 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, which 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 shown 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.

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 characterised by their limited data-set size. They reach a small entropy and by that a low 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 a SQL environment. From there, we've extracted selected tables and their data, exporting them as CSV files for compression. We've chosen CSV as 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. This is due to the fact that CSV is text-based and therefore each digit is encoded with 8 bits. Values that are encoded in the database with a fixed size (e.g. integer with 4 bytes) can therefore require more bits or less bits in the CSV file, leading to a different file size.

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 of the Stack Overflow database and the reference compression's with 7Zip.

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 result of the single tables and the complete database with 7Zip. The compression is fulfilled with two different compression methods on the exported CSV-file. Therefore the size of the exported CSV-file is also given. In the BZIP2 compression, a sliding-window size of 900KB is used. The second compression method 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 methods 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 need to be performed by the system. The data-set consists of eight tables with different sizes and number of 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 allows to define the size of the database dynamically [13]. Even if the randomly generated data brings with it a large entropy and thus a large variance of data, this usually does not reflect reality. In real databases, a structure can often be recognised. For example, consecutive IDs or ascending dates. Even randomly generated text does not reflect the real world. In real text, certain characters occur more frequently then others. With random data all characters occur statistically equally often. However, data with such characteristics can also appear in databases, which is why the TPC-H database is used as a benchmark. This allows to investigate the behaviour of the compression methods on data with high entropy. In this work a 1GB database is used.

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.

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. 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

Reference compressions of TPC-H									
	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 of the TPC-H database and the reference compression with 7Zip.

of multiple tables with different sizes. The largest table is LineItem which accounts for three quarters of the entire database. As before, the table gives the size of the database in the db-size, as well as the size of the exported tbl files. The exported table represents a table structure. Similar to CSV, the cells are separated by separators. Where each symbol is represented as text with an 8-bit coding, while the database uses defined data types. This means that in databases the required bits remain the same regardless of the length of the entry. However, when exporting the database to a .tbl-file, the size of the exported file depends on the length of the data in the fields. This leads to a different size of the data.

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

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

3.3 Weather - Database

The weather database contains weather data from different cities in the United States of America. There are many small tables publicly available, each containing data from 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 tables are with a size of 6MB to 9MB quite small compared to the tables in the benchmarks before. Furthermore, the data-set does not have that variety of data-characteristic nor many different data types. The tables only contain Text, Float, Int and Datetime. The size of the final database can be defined by the number of weather stations 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 a large amount of numbers. Most columns contain text with just small values per cell. Often some numbers are followed by a few letters. This characteristic differs from the previous databases where the text mainly consists of letters without numbers. [14]

Only the first 12 columns of each table are shown in Table 5. The following columns are all of data type text and vary just slightly in the entropy. Thus, the text is the data type that occurs most frequently in the database. Some of the columns of each table reach 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

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
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.

in the table. The interesting part of the benchmark are the text columns with many numbers in the cells.

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 of the Weather database and the reference compression with 7Zip.

The size of the database is with 24.8MB in the exported version and 38.8MB in the original database size smaller than the previously described benchmarks (see Table 6). However, the selected tables are sufficient to test the behaviour of the compression, since the data-characteristic of each table is similar to each other as the entropy shows. Once again, some compression references are created with BZIP2 and LZMA.

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 methods achieve the best compression ratio and decompression time. This work puts focus on warehouse databases and user databases. As the benchmarks described 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 methods are combined and the variant with the best compression ratio is selected. The results between the different compression steps are plotted to provide an overview of the performance of each method. Since the target is to receive a high decompression throughput on the FPGA, the achieved compression improvement for each method applied to the data needs to be analysed. If a computational intensive decompression method improves the compression result only by just a few bytes, than the decompression effort can be too 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 gave us the opportunity to achieve the best compression ratio per column by applying the best fitting compression method to that data. Therefore each table is splitted into separate columns. Then the data type for each column is determined. The information about the data type is used to choose the best fitting initial compression method. The choice of compression method depends not only on the data type, but also on the characteristics of the data. Data with a smaller entropy has 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 achieves 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 has 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 unorganised 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 a 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 is used for the compression of numerical values. This number only varies between the columns and is determined by bitpacking described in Section 2.3. This also saves the use of separator bits between the individual values per cell.

4.1 Integer-Compression

Integers are usually stored in databases in 4-byte words. With 4 bytes, numbers in the range from -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 methods and combinations of compression methods 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 applied on the output. Huffman assigns a new coding to the symbols. As this coding is already in binary format, further compression with Delta or Run-length does not achieve any further improvement. For further Run-length compression, longer runs would have to be present in the data after Huffman coding. Therefore, Huffman can only be the last compression step 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 executed in parallel on hardware. Having the same bit length for each number improves the hardware efficiency, as less logic elements are needed. To decoded all values with the same bit length, the largest number after Delta or Run-length compression is used. The minimally needed bits to represent that number is calculated and an additional bit for the sign is added. This determines the minimum number of bits with which each number can be represented after Delta or Delta+Run-length compression. Bitpacking 2.3 is then used to encode all values in the column with the calculated number of bits. This method entails a variable bit length between the individual columns of a table. The compression effect increases the smaller the maximum number is. However, the number of bits can also be greater than the original number of bits. This can occur with large deltas, especially with delta encoding with sign changes, which require many bits for representation. In such cases, an escape character can be set and these values can be encoded uncompressed. This prevents the compressed data from being larger than the uncompressed data. However, this also leads to larger hardware requirements in decompression on the FPGA.

Another option is the usage of multiple quantization steps. By defining multiple fixed bit lengths, the hardware on the FPGA can be kept simple. However, such quantisation steps would also result in a worse compression ratio since the overhead of leading zeros would increase. By using the calculated maximal number of bits needed to represent the largest number in a column the overhead can be kept small, resulting in a better compression ratio. Therefore, Bitpacking 2.3 or Huffman 2.4 is the final compression step in integer compression.

The Huffman compression can be used as a final compression step or just as a single compression. The method 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 static huffman tree shown in Figure 1.

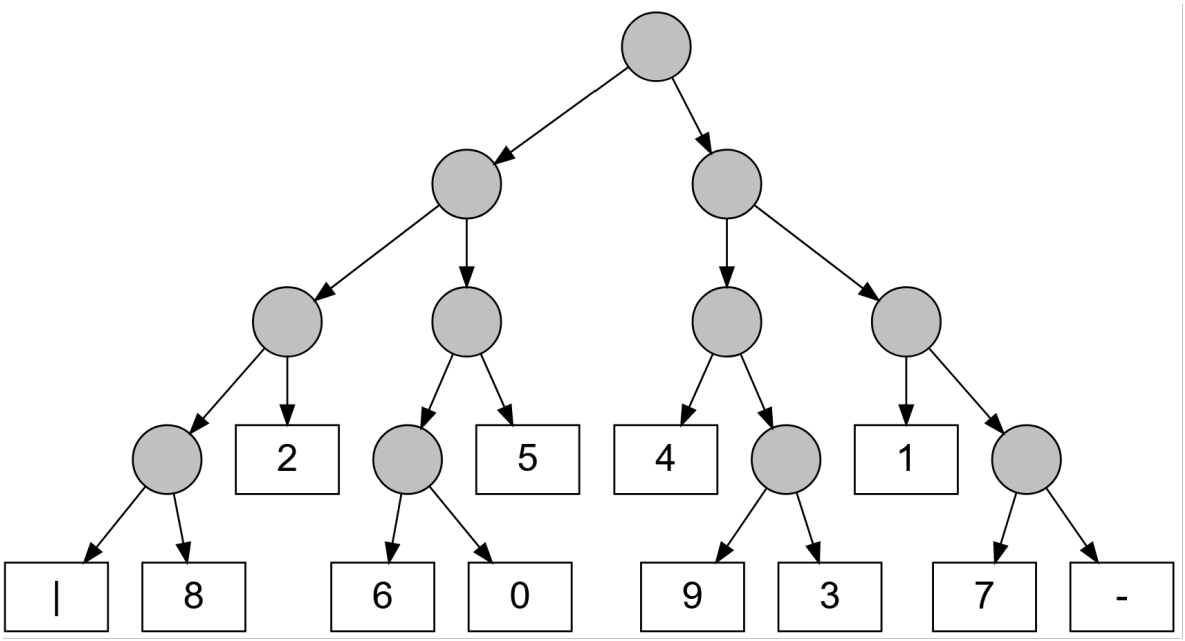

Figure 1: Static Huffman-Tree for compression of integers.

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 achieves a worse compression ratio. And even with small values, a poor compression rate can be achieved by Huffman compression. Since with Huffman compression a dynamic bit length per number is created, an additional separator is necessary. This separator does not specify the start or end of the coding of a symbol with Huffman, as Huffman encodes prefix free. However, a separator must be used to mark the start and end of a cell in the database. This is important in the decompression to recognise which decoded symbol belongs to which cell.

To find the best compression method for integers we analysed the compression with Delta and Huffman, as well as the combination of both 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 the data-set A in Figure 2, the normalised 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. 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 run. As the repeating sequences are too short and the values changes too 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. By separating token and length using a marker, the two values can be encoded with a different number of bits. The large outliers would not affect the bits required for the length. This leads to a lower break-point and makes Run-length compression more effective. However, this also increases the overhead during compression, as an additional character is used.

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

Data-set B in the Figure 2 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 zero except the first one. With Run-length, the compression ratio can be further improved. The amount of numbers in the compressed data is reduced to 4. 2 for the initial value and

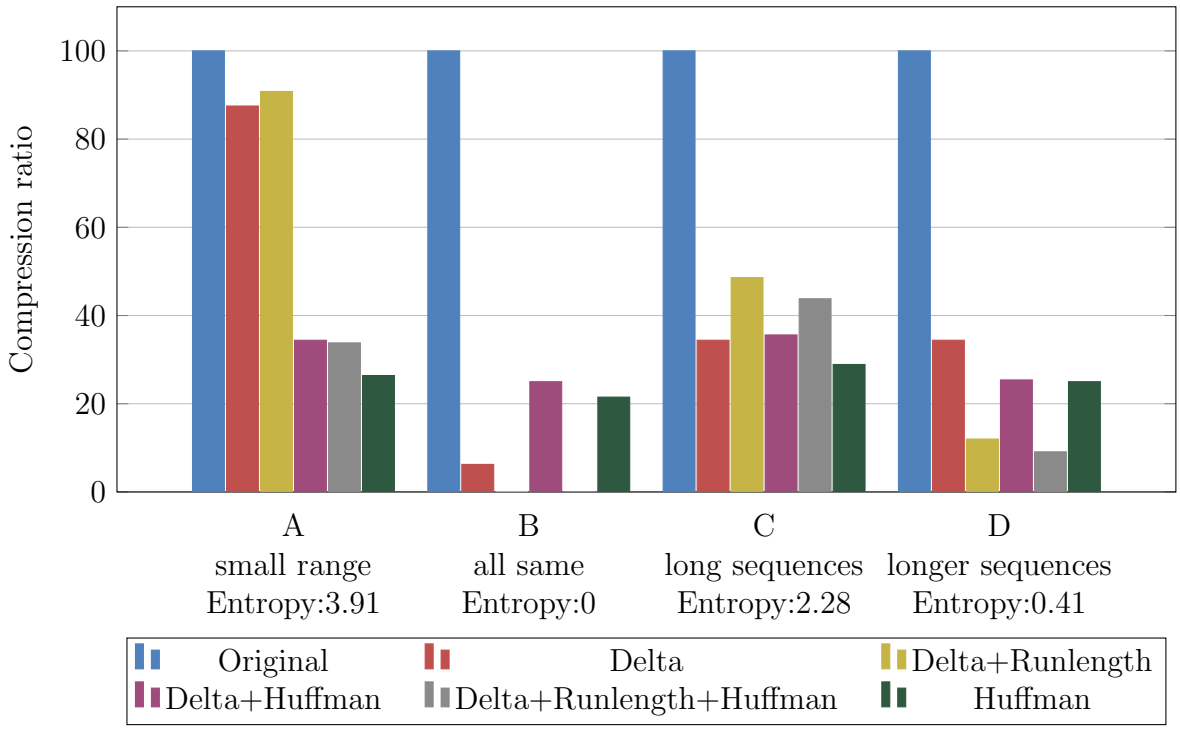

Figure 2: This chart shows the compression results(ratio) of different compression methods and combinations of compression methods. The compression is applied on structured data (integers) with low entropy. The compressions are applied on the data: A=VoteTypeId(Stack Overflow), B=LinkTypId(Stack Overflow), C=HourlyAltimeterSetting(Weather-Db), D=HourlyAltimeterSetting(Weather-Db).

2 for the token and the length. The Huffman compression on top of that can't improve that compression further. Especially the Delta + Huffman combination receives worse results, since 3 to 4 bits are needed for each value even, if its just an array of zeros.

Data-set C and D in Figure 2 are the HourlyAltimeterSetting columns of the tables Weather-Nashville and Weather-Bristol from the Weather database. The entropy of both columns is small with 2.28(Nashville) and 0.41(Bristol). Like the two columns before, the data is structured and contains longer sequences of repeating values. The graph for data-set C shows the compression where all five methods achieve a similar result. Whereas the compression with Run-length makes the result slightly 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 data-set C 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. Nevertheless, it depends on the value of the outliers. Furthermore, the number of changes 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 and low entropy, the compression 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.

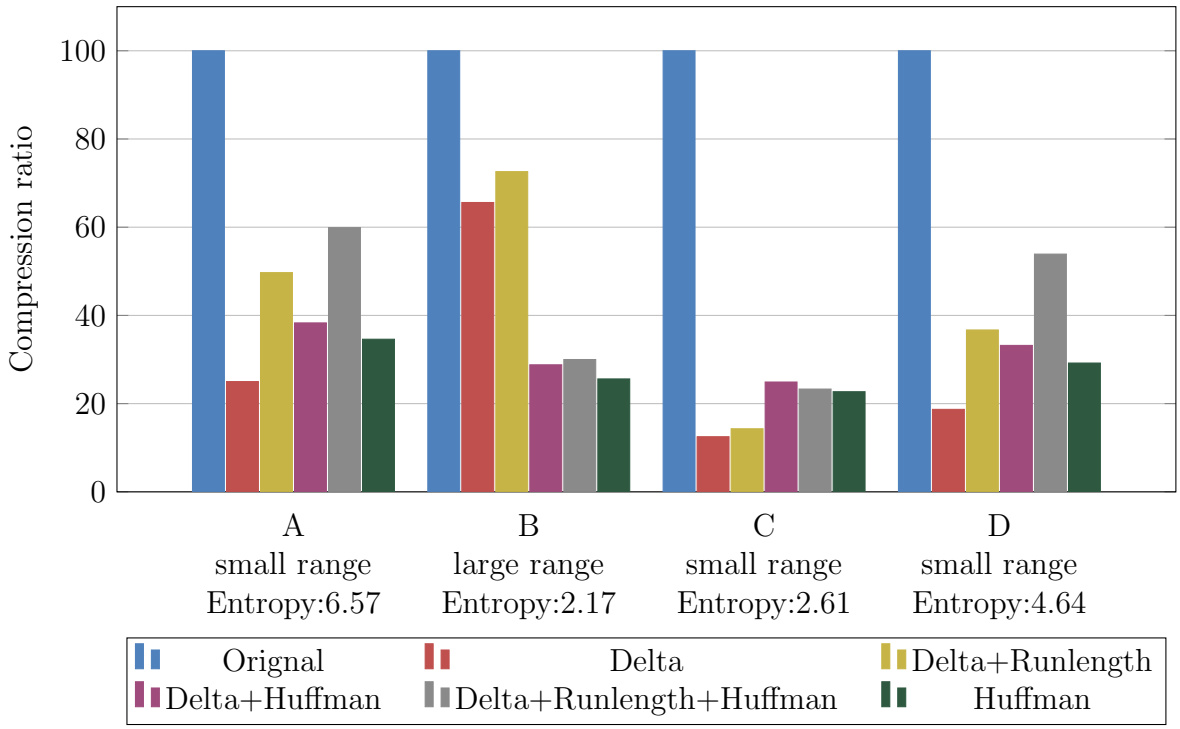

Figure 3: This chart shows the compression results(ratio) of different compression methods and combinations of compression methods. The compression is applied on unstructured data (integers) with low entropy. The compressions are applied on the data: A=Age(Stack Overflow), B=Downvotes(Stack Overflow), C=fourth column of Lineitem(TPC-H), D=fourth column of suppliers(TPC-H).

In first data-set A of Figure 3, the normalised 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 to 8 bits including the sign. This results in a compressed size of 25% with Delta compression, compared to the uncompressed size. Because of the disorder in the data-set, the Run-length achieves a ratio worse. There are too many changes in the values, leading to many runs representing only one value. With Huffman, the worst case needs 12 bits to represent the number and an additional 4 bits for the separator, resulting also in a worse compression ratio. Where the separator is used again to mark the end cell in the table.

The second data-set represents the compression ratio of the Downvotes column in the users table. Since the values in that column are quite high and vary much, the deltas also stay high. This leads to a slightly worse compression rate than within the

data-set A. As expected a further compression with Run-length is leading to an even worse ratio.

The last two columns in the Figure 3 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. As in the cases before, the Delta compression achieves a better ratio than 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 have large gaps between each other, the Huffman compression reaches a better compression ratio since fewer 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 by Delta compression. The large number of different values indicated by the high entropy leads to many changes that makes Run-length compression un-profitable. Whereas the structure of the data can lead to small deltas between the values and result in a good compression.

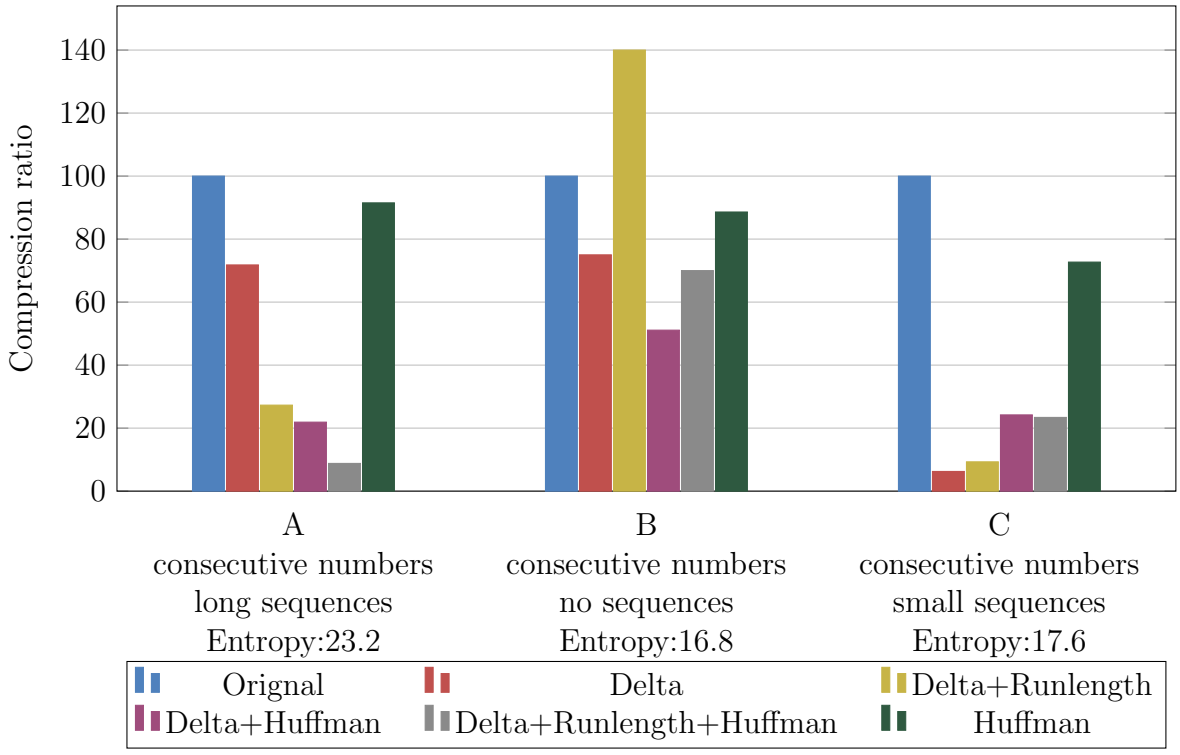

Figure 4: This chart shows the compression results(ratio) of different compression methods and combinations of compression methods. The compression is applied on structured data (integers) with high entropy. The compressions are applied on the data: A=Id(Stack Overflow), B=PostId(Stack Overflow), C=first column of Partsupp(TPC-H).

Column A in Figure 4 shows 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, which can be easily compressed into runs, thus reducing the values required for representation. However, the outliers still generate a large overhead in the representation of the runs. This overhead can be reduced with additional Huffman compression. Although the large outliers require more bits for Huffman compression, they occur rarely. The many small values can be compressed efficiently and reduce the average consumption of bits per value, which leads to a further improvement in the compression ratio. The single Huffman compression itself gains a bad ratio, because the numbers are increasing up to 33.551.687. This means that for 2/3 of all numbers, 8 digits must be encoded. This leads to a consumption of $8 * 3 = 24$ bits to $8 * 4 = 32$ bits in the worst case. In addition, 4 bits are required for coding the separator, which marks the end of a table cell and the start of the next cell. The worst case value is obtained because the 8 digits of the number are coded individually with Huffman. According to the tree for encoding integers, described in Figure 1, the digit is encoded with 4 bits in the worst case. This leads to the result of 36 bits in the worst case. The numbers in the first third can be encoded with fewer bits and reduce the average bit consumption. On average, however, only a slight bit saving can be achieved.

The column B represents the PostId (entropy 16.8) in PostLinks table of the Stack Overflow database. As in column A, the numbers are in a consecutive order. In contrast, the deltas between the values are larger and vary more in their value. As before, due to some large deltas, the Delta compression achieves a bad compression ratio. Furthermore because the deltas varying a lot, 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. The best combination is Delta+Huffman since the dynamic deltas can be compressed in a more efficient way with Huffman where the outliers do not have such a large impact.

The last data-set C 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 most numbers repeat four times before increasing. Delta compression achieves 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 most numbers are repeating four times, the bit length for representing the length in a Run-length run becomes 4 bits. This leads to a run size of 8 bits. For the case 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 + cell separator are needed, and in the worst cases 4 bits + cell separator are needed.

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 than 50% of data-set. This is because each run is encoded with 2 values with the same number of bits. The number of bits is usually determined by the preceding delta compression. Also, the length of the longest sequence should be smaller than the largest token, otherwise the length of the running length determines the number of bits. In such a case it is better to

divide the sequence into two runs to keep the number of bits per value small. The pure Huffman compression does not work well, because most numbers have too many digits and therefore more bits are needed for the representation than the result with Delta compression. However, the combination of Delta and Huffman usually achieves a good compression rate. Delta reduces the size of the numbers and they consist of fewer digits. This allows Huffman to compress them efficiently. This is particularly noticeable when a few large outliers drive up the number of bits during delta compression, which leads to large overheads. With Huffman, these overheads can be reduced efficiently and a significantly better compression ratio is achieved.

The last possible data structure is unstructured data with a high entropy. It is expected that here 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.

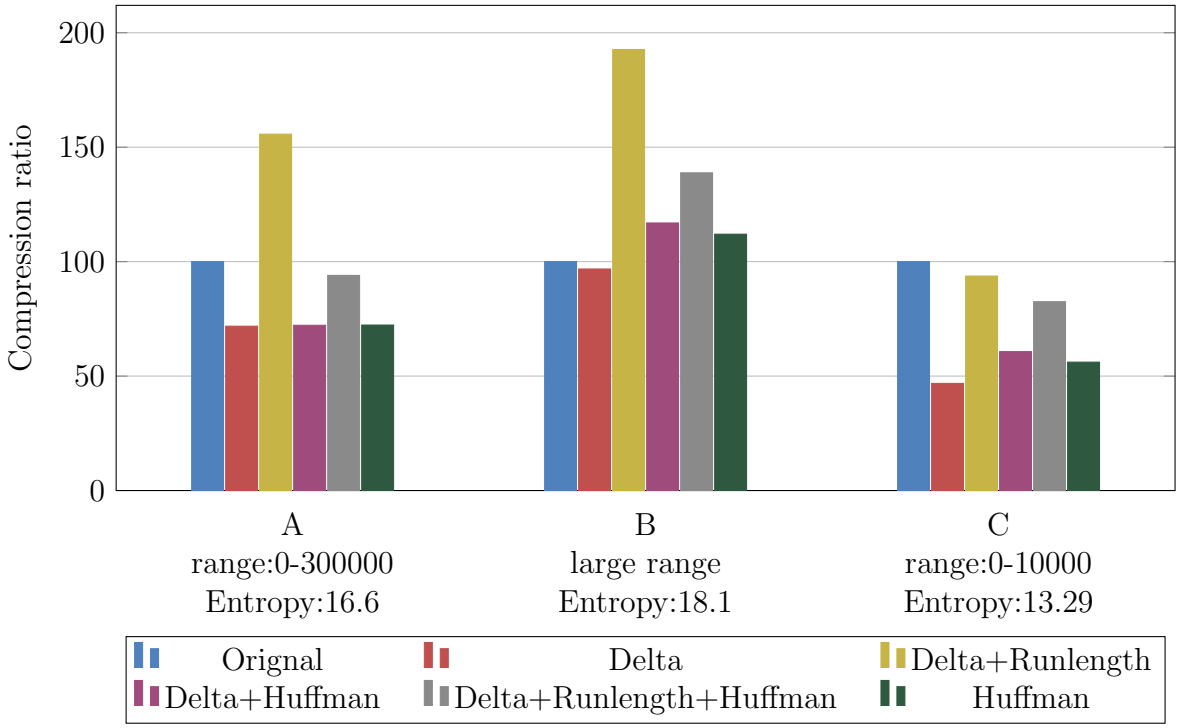

Figure 5: This chart shows the compression results(ratio) of different compression methods and combinations of compression methods. The compression is applied on unstructured data (integers) with high entropy. The compressions are applied on the data: A=UserId(Stack Overflow), B=EmailHash(Stack Overflow), C=third column of lineitem(TPC-H).

The compression of unstructured data with a high entropy achieves bad compression ratios. Data-set A in Figure 5 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 a 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 the data doesn't contain longer sequences of the same value after Delta compression, the Run-length compression reaches a ratio of 155%. That is roughly double the size of the Delta compression result, which means that the compressed file contains more bits than the original uncompressed one. This is because each run represents just one value. Because many values are 7 digits long (including separator), the Huffman compression needs 28 bits to represent the value in the worst case, which also leads to a small compression effort. However, this also means that other more frequently occurring digits require significantly fewer bits and improve the compression ratio. The worst case shows that a reduction of data size is possible with Huffman in any case.

In data-set B in Figure 5 the EmailHash of the User table is shown. It has an entropy of 18.1 and contains a similar random structure as data-set A. The difference is, that the numbers in that data-set are larger. This results in a similar behavior of the compression methods as seen before. However the compression ratios are even worse, since the larger numbers need more bits to be represented. Nearly all methods generating larger compressed files than the uncompressed file.

The last data-set C is Col3 of the lineitem table in the TPC-H database. This column has with 13.29 the lowest entropy of all 3 columns considered. The range of the randomly ordered data is between 0 and 10.000. This makes a compression with Delta efficient, cause the deltas can be kept small which leads to a lower consumption of bits. As expected with unstructured data, no longer sequences are included, which worsens the result of Run-length compression. With Huffman, the worst case encoding needs 24 bits to represent a value whereas the Delta compression just needs 15 Bits in the worst case. However, mentioned before, the worst case with Huffman also means that more frequently occurring numbers can be represented with significantly fewer bits. This leads to a significant improvement in the compression ratio, which can also be seen in the Figure 5. Huffman achieves a slightly worse ratio than Delta.

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, then the Delta compression achieves a good compression ratio. As the numbers and thus the bits required for the representation can be kept small. If both values in the run are coded with the same number of bits, than the number of character changes may only be 50% of the total number of values in order to achieve a positive compression effect. Otherwise the representation of the run requires more bits than the uncompressed representation. This behaviour appears also if the length of the run is larger than the token. The break-point can be shifted by using a marker. This allows the values in the run to be encoded with a different length and run-length can have a positive effect on the compression rate even with more character changes. With Delta, the structure of the data is only slightly changed. This means that sequences of characters are retained. A sequence of numbers $[3, 3, 5, 5, 5, 5]$ becomes $[3, 0, 2, 0, 0, 0]$ with Delta. If the size of the numbers has been reduced with Delta, the data size can be further reduced with Run-length by summarising the sequences into runs. This leads to a better compression ratio. In some cases, Delta can even be used to create a sequence of identical characters. For example with ascending IDs, which always increase by 1. Delta then outputs a sequence of 1, which is easy to compress with Run-length. Like in the chart 'structured data with high entropy (A)' (Figure: 4) shows.

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. The large outliers are compressed with a worse ratio than with delta, but as they do not occur often, a better result is achieved on average.

The pure Huffman compression reaches a similar result to the compression with Delta+Huffman in most cases. Except if the data is structured, having a high entropy. Since the number of characters per word has already been reduced with Delta. This means that Huffman in combination with Delta has to recode fewer characters in this case than pure Huffman, which has to encode the full word length.

4.2 Float-Compression

The compression of Floating-Point numbers is more complex than integer compression. According to the IEEE 754 standard, floating points are divided into sign, exponent and mantissa (fraction). Sign involves pre-drawing. The exponent indicates the value before the decimal point. The mantissa (fraction) contains the decimal places. Figure 6 shows a single precision float.

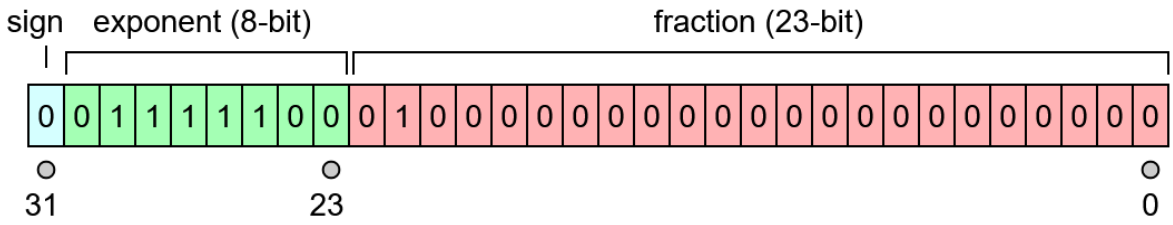

Figure 6: IEEE Floating Point Standard 754. [15]

The single precision IEEE 754 format uses 32 bits to represent floating point values. If delta compression is applied directly to the floating point values, as is the case with integer compression, the data is not reduced. This is because the resulting values are floating point numbers, which are still represented with 32 bits according to IEEE 754. It is possible to reduce the value but not the necessary bits, as these cannot be reduced for a lossless representation. Compression with Run-length, on the other hand, can reduce the data if the data changes less frequently and therefore sequences of length > 2 are available. However, the Run-length we use, without a marker, requires 64 bits per run, as the token (float values) and the length of the run (integer) are encoded with the same number of bits. This can lead to a poor compression ratio. As with integer compression, direct compression of floats with Huffman achieves a good compression ratio if the data set consists of less than 8 digits per value on average. Since applying compression methods directly to the floating point values often leads to poor compression ratios, we have investigated three different approaches to compress floating points more effectively. The aim is to use the same compression methods as for the compression of integers. This means that resources on the FPGA can be saved during decompression, as the decompression modules can be reused and no new decompression modules have to be implemented.

The first approach is separating the exponent and sign from the mantissa. This means that 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. The splitted values are compressed using the same compression methods as used in the integer compression. Thus, the existing decompression modules can then be used for decompression, which saves resources. Only the merging of exponent and mantissa at the end of the decompression needs to be implemented additionally. By splitting the exponent and mantissa, it is also possible to decompress them in parallel, which increases throughput.

The second approach shifts the decimal point of a floating point value to the right in order to receive an integer. For example the floating point number 334.56 becomes the integer 33456. The amount the decimal point has to be shifted is depending on the largest mantissa. Therefore the number of digits of the largest mantissa is counted and the decimal point is shifted by this value. Applying a uniform shift value to all floating-point numbers within a column minimises 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 an 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. This approach is only effective if the resulting integers are smaller than 2.147.483.647, otherwise more than 32 bits are be required for representation and no compression effect can be achieved. However, this approach can lead to good compression rates if the exponent and mantissa consist of a few digits. Delta compression and Huffman compression can then compress effectively, just as in the integer compression (see Section 4.1).

The last approach is using Huffman compression. As already described, Huffman compression can achieve a good compression ratio when applied directly to the floating point numbers. Therefore, the data is not pre-processed as in the other two approaches. By shifting the decimal point it would be possible to save a character when compressing with Huffman. However, this would lead to more complex decompression, which would reduce throughput. Therefore, Huffman compression is applied directly to the floating point values. A advanced version of the static tree (Figure 1) is used for Huffman compression. This was expanded to include the decimal point, which did not lead to a new level in the tree, so all values can still be encoded with a maximum of 4 bits.

It is expected that the split of the 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 gains the best compression ratio on these values.

The three approaches are applied to six different data sets to determine the behaviour of the individual approach. The data sets used differ in terms of data structure and data characteristics. By testing the approaches on different combinations of structure and data characteristics, it is determined which approach achieves the best compression ratio in each case. The data characteristics differs in the length of the exponent or the mantissa. The first data-set A, contains consecutively increasing floating point numbers with small deltas. The exponent increases up to approximately 1.000.000, whereas the mantissa stays at zero. Thus, this data-set has an entropy of 18.9. The second data-set B consists of unordered data with a large exponent and a small mantissa. The mantissa

is just 2 characters in size. This data-set has an entropy of 20.47. The third data-set C contains a small maximal 4-digit exponent followed by a 2-digit mantissa. The data is, as before, unsorted and without any pattern, but has a slightly smaller entropy of 17.06. Data-set D consists of a long sequence of the same value, where the float represents a small number. The entropy of this data-set is 6.06. The data-set E contains unsorted values with large exponents and many decimal places. This data-set has an entropy of 14.65. The last data-set F contains a static exponent of zero and a small 2 digit mantissa in an unsorted manner. With 3.46, this data-set has the smallest entropy. All three approaches for compressing float are tested with these data-sets. This allows us to compare the approaches and determine which approach compresses best.

In the following Figure 7, 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 the mantissa and exponent respectively. The proportion of the compression result of the exponent is shown by the pink box. The share of the mantissa is marked in green.

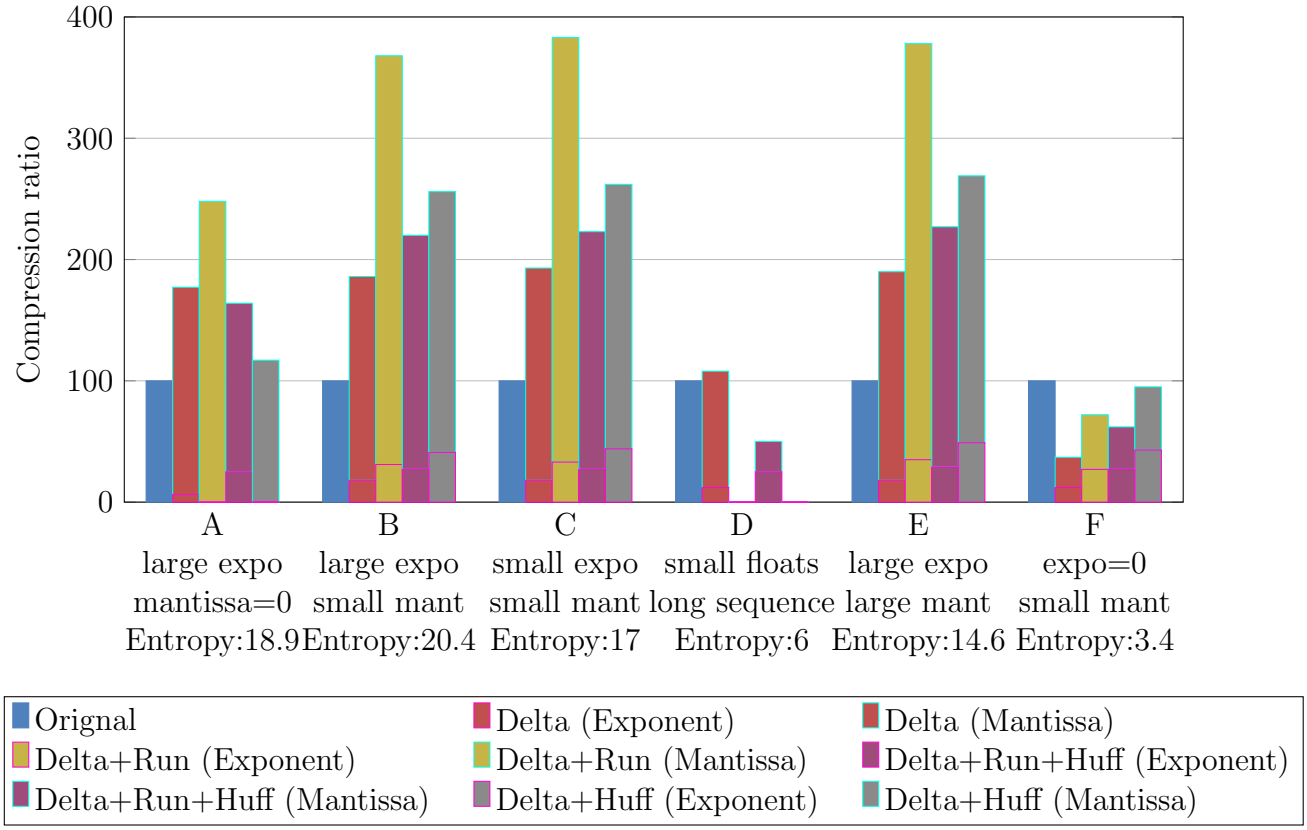

Figure 7: This chart shows the compression ratio of the individual compression methods. The first approach, in which exponent and mantissa are considered separately, was used. The compression methods are applied on float values with different characteristics. The compressions were performed on the following data-sets: A=Id(Stack Overflow), B=fourth column of orders(TPC-H), C=sixth column of customers(TPC-H), D=Latitude(Weather-Bristol), E=Reputation(Stack Overflow), F=seventh column of lineitem(TPC-H).

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. The tuple contains the normalized mantissa (fraction) and the exponent. The mantissa is always in the range $[1.0, 2.0]$. The exponent is an integer that represents the power of 2. The mathematical formula for this is:

$$number = m * 2^e$$

Hereby, m represents the mantissa and e the exponent. With a binary mantissa, the base is 2 [16]. The mantissa is converted into an 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 at the same value and only the exponent increases, the converted integer can become quite large. After both values are converted into integers, the values can be compressed like in the previous Section 4.1. This leads to the results shown in the Figure 7.

Due to the small exponents as a result of the split, the compression of the exponent works out best with Delta compression. The small values lead to small deltas, which can be represented with a few bits. Further compression can be applied to this as described in Section 4.1. However, the mantissa makes up the majority of the compressed data. This is because the exponent is represented with a maximum of 8 bits (including sign). This means that the exponent cannot contain values larger than 127. This can be easily compressed using the integer compression described in Section 4.1. The mantissa determines the accuracy. The larger the mantissa, the more precise the number after the decimal point. As the mantissa is always in the range from 1.0 to 2.0, it is a floating point number in the decimal system. Where the mantissa can contain many digits after the decimal point. The more precise the number, the more digits the mantissa has. If the mantissa and exponent are considered separately as integers, the integer of the mantissa has a large range, which means that many bits are required to represent it. Furthermore because the separated mantissa changes with every change in decimal or fraction part, mostly no longer sequences can be found, which leads to a even worse compression with Delta and Run-length. A further compression with Huffman improves the ratio, but still results in an inferior compression ratio.

In most cases, the separate compression of exponent and mantissa results in more data than in 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 the compression of dataset D. With a small exponent as well as a small mantissa, the compression with Delta can also achieve a good compression result. This is because the separation keeps both the mantissa and the exponent small.

Overall, the compression of floats with a separately considered compression of exponent and mantissa leads to a higher bit usage in most cases. With the exception, if the mantissa and the exponent are small, a better compression result can be achieved by considering them separately.

The results of the second approach to compress floats is represented in the next Figure 8.

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 provides the opportunity to compress the values as an integer as described in Section 4.1. The disadvantage of the conversion into an integer is that the numbers can become very large and thus require many bits for their representation.

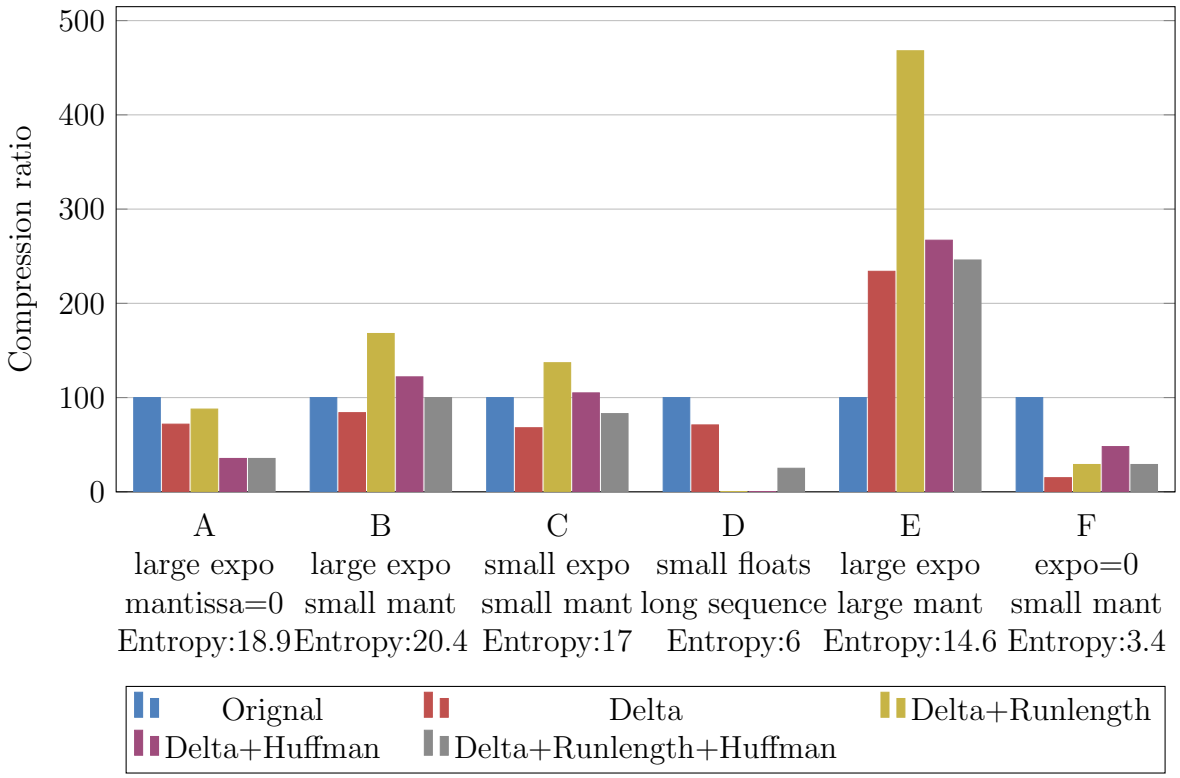

Figure 8: This chart shows the compression ratio of the individual compression methods. The second approach was used for this. The float values are converted to integers by shifting the decimal point. The compression methods are applied on float values with different characteristics. The compressions were performed on the following data-sets: A=Id(Stack Overflow), B=fourth column of orders(TPC-H), C=sixth column of customers(TPC-H), D=Latitude(Weather-Bristol), E=Reputation(Stack Overflow), F=seventh column of lineitem(TPC-H).

As the Figure 8 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 approach, leading to fewer bits needed. The behaviour of the compression is analogous to the behaviour described in Section 4.1. The combination of large exponent and large mantissa results in a very large integer. This is the case in data-set E, as the Figure 8 shows, the values cannot be compressed profitably. More than 32 bits per value are required for the representation of integers, which leads to a higher bit consumption for the compressed values. This could be prevented by using an escape character. Values greater than 32 bits would thus be encoded uncompressed. This leads to a better compression ratio, as the large outliers have no impact on the

other values. However, the use of an escape character also leads to more decompression effort.

The last approach investigated is 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 in Section 4.1. With the new tree, the worst case for representing a digit is still at 4 bits, whereas the best case is 3 bits. The result of the Huffman compression on the different data-sets is shown in Figure 9.

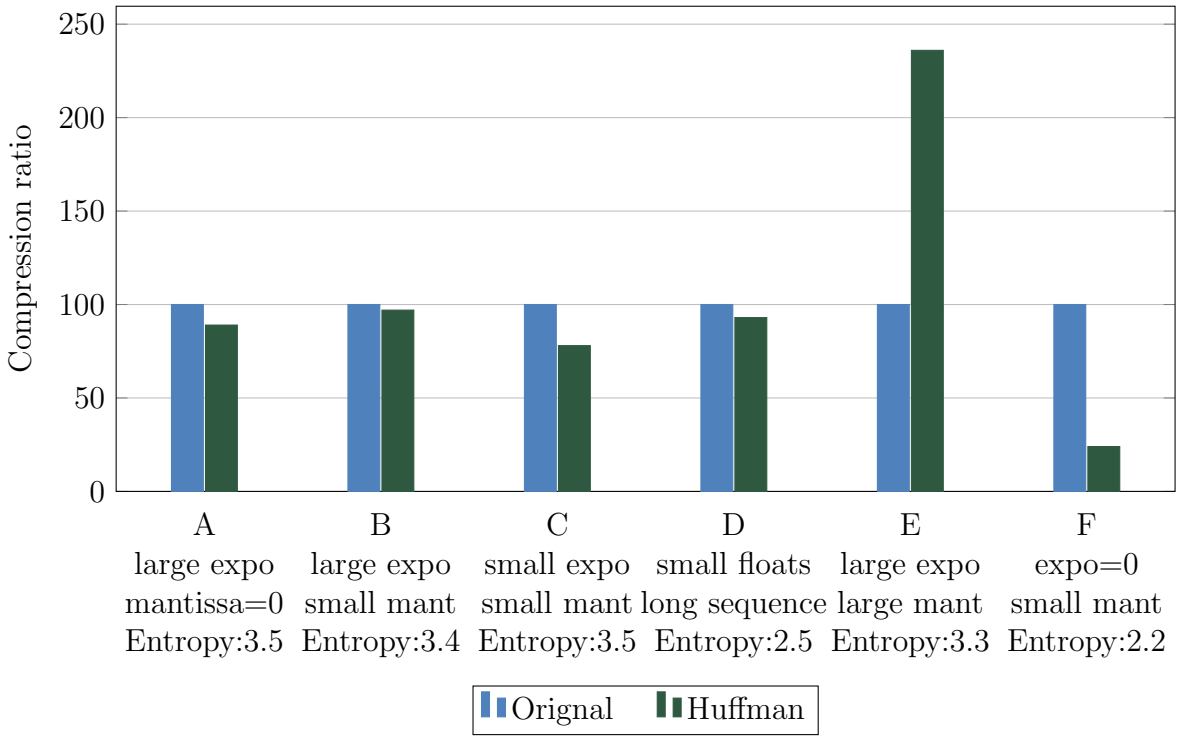

Figure 9: This chart shows the compression ratio of the individual compression methods. The third approach was used for this compression. The float values are not converted or considered separately, but compressed directly with Huffman. The Huffman compression is applied on float values with different characteristics. Entropy is calculated over the individual digits of the numbers and not over the number as a whole as with the other approaches. The following data-sets are used: A=Id(Stack Overflow), B=fourth column of orders(TPC-H), C=sixth column of customers(TPC-H), D=Latitude(Weather-Bristol), E=Reputation(Stack Overflow), F=seventh column of lineitem(TPC-H).

The compression through Huffman reaches just a slightly better bit usage in most cases. This is due to the fact that most values consist at least of 6 to 7 characters, whereas in the worst case, the 4 bits per character are needed, resulting in 28bit for a 7 character long float. This is reflected in the compression ratios in Figure 9. As most values consist of 6 to 7 characters and therefore 24-28 bits are required for encoding, on average only 4-8 bits can be saved per value. This leads to a poor compression ratio for the data-sets we use. The fewer characters a float value has, the better Huffman

compresses it. This is particularly noticeable when compressing data-set F. Due to the small exponent and the small mantissa, the float consists of few characters. Huffman has a particular advantage over the other compression methods when a few outliers in the data drive up the bits required for representation. The few outliers mean that Delta and Run-length encode all data with this number of bits. Huffman, in contrast, encodes the values dynamically with only the minimum number of bits required per value.

The lossless compression of floats is challenging. The splitting of exponent and mantissa results in large numbers, requiring many bits for its representation. While the exponent can be compressed with a good ratio using Delta, the mantissa compression is worse. The large numbers created by separating and converting the floats to integers require many bits. Furthermore, there are often large deltas between individual values. This also makes further compression using Huffman inefficient, as the deltas also consist of many characters. 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, when the data mostly consists of many characters. Huffman can achieve a much better compression ratio, if on average few characters are used per float and there are just few outliers. However, due to the decompression effort of Huffman, the throughput can be lower than just passing the uncompressed data, if the compression ratio is too bad.

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

4.3 Boolean-Compression

The Boolean datatype is quite simple, since a Boolean just can be true or false. This also leads to a 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 those bits. The second method is a specialised 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 out best, whereas for longer data-sets with longer sequences of the same value, the specialised run-length method should achieve better results.

The first data-set in Figure 10 contains random values 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 run of two values is needed. This doubles the size of the compression result in respect to the bitpacking attempt.

The second tested data-set 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 combined with Run-length compression, the break point is not 50%. Because the length of the sequence

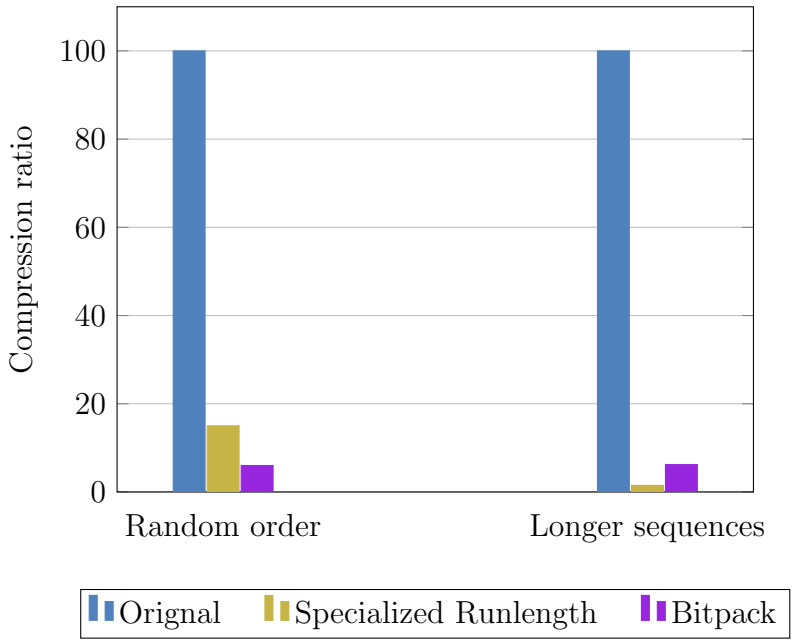

Figure 10: This chart shows the compression of Boolean values. The compression ratio which is achieved using specialised run-length compression as well as compression using bitpacking can be found. The two compression methods were applied to two data-sets. The first data-set consists of random numbers without a sorted order. The second data-set consists of Boolean values, which form longer sequences.

defines the amount of bits used to compress each value and not the value of the run anymore. If the length of the sequences contained in the data-set are similar, a good compression rate is achieved. The compression generates few overhead bits, as all values require the same or similar number of bits for representation. However, if there are large differences between the sequence lengths, many overhead bits are generated for the shorter sequences, leading to a worse compression ratio.

4.4 Date/Datetime-Compression

The date and datetime datatypes 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 these 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 requires 3 bytes [17].

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 [17].

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. For this purpose, an extended version of the Huffman tree is used, which has already been used for integer compression (see Figure 11).

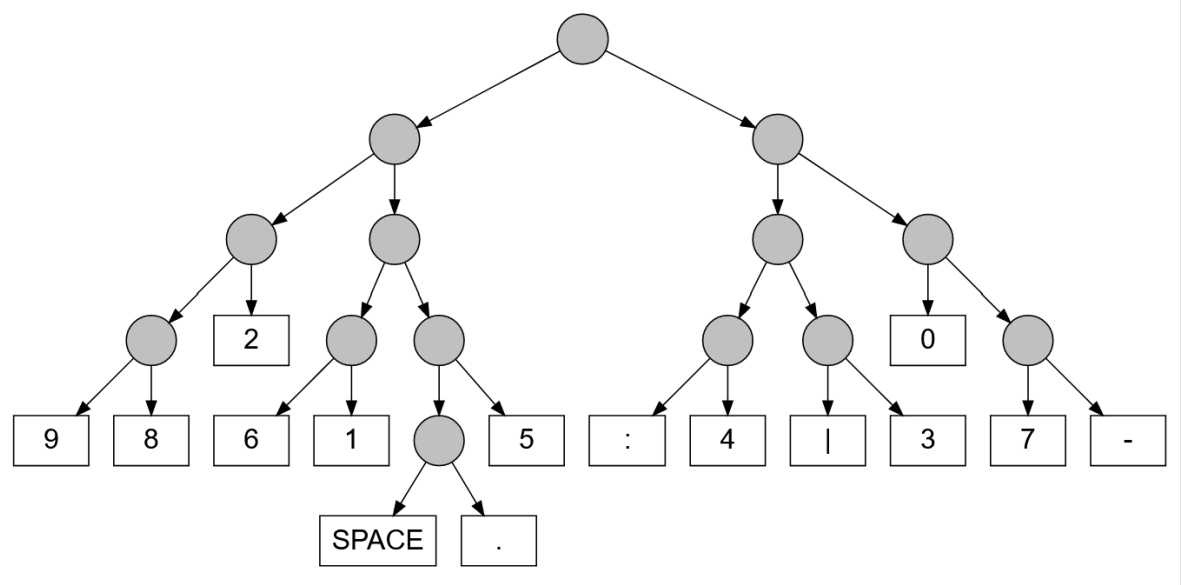

Figure 11: Static Huffman-Tree for compression of dates and datetimes.

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 in the worst case only 4 bits are 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 suitable.

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 following the Huffman tree. Whereas the original representation just needs 64 bits.

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

The second method aims to convert the date and datetime datatype into the Unix timestamp, where the Unix timestamp provides the count of seconds from January 1st, 1970 to the given date. This number is an integer. In addition to representing the Unix timestamp in seconds, it can also be represented in milliseconds. Although this requires more characters, it achieves higher accuracy [18]. The timestamp can therefore also be used to represent the datetime datatype. However, as the number consists of more characters, more bits are required for the representation. Because the Unix timestamp is an integer, the integer compression can be used as already described in Section 4.1.

The results from the Integer Compression 4.1 can also be transferred to the compression of the timestamp. The disadvantage of this method is that 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. However, the timestamp becomes smaller the closer the date is to the initial value, 1970. This means that fewer bits are required for the representation. With 3 byte the maximum timestamp that can be represented is 16777215. This means that only dates up to 14 July 1970 can be represented more efficiently than the database. The timestamp of datetime is in the range of 1-2 trillion and needs 41 bits to be displayed. This is only about two thirds of the bits originally needed. A further compression with Run-length or Huffman is 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 Figure 12 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 differ. 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 Figure 12 shows, the size of the converted date to an integer timestamp increases the size of the data, as explained before. Both columns in the chart reach 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 the random order, 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 the results described in Integer Compression 4.1.

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

As shown in the first column the compression with Delta and Huffman achieves a slightly worse ration than just Delta. This behaviour can be explained by the large deltas between the values. The numbers have a little more than 8 digits on average. 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 run is needed to be compressed, which leads to an inferior compression ratio.

In conclusion, the compression of the date datatype is quite challenging due to the large amount of digits per value and the larger numbers after converting to the Unix timestamps. The compression results are the same as for integer compression. However, Delta+Huffman compression achieves on average the best ratio over the different data structures. Even if in some cases, the additional Huffman results in a slightly worse

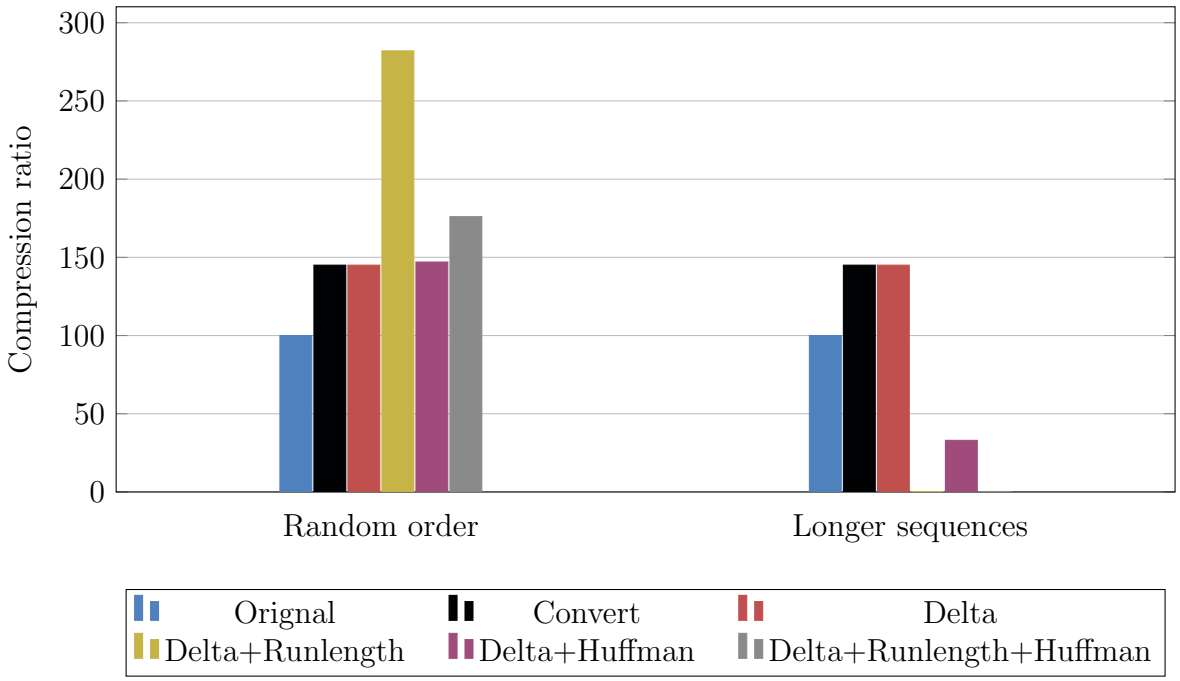

Figure 12: This chart shows the compression of date datatype. The compression ratios of the conversion to the Unix timestamp (convert) are shown. The compression ratios of the compression methods used are also shown in the chart. The compression methods were applied to two data-sets. The first data-set consists of random unsorted dates with an entropy of 11.27. The second data-set consists of dates values, which form longer sequences. This data set has an entropy of 9.64

ratio. Due to the static amount of bits per value, the Delta compression is not reducing the data-size. The more dynamic Huffman compression reduces the data-size, if the limit values described earlier, are complied with.

The compression results of datetime compression is presented in Figure 13. In the first column of the chart, 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 second column represents the Date column in the badges table. Here, the data is sorted and has small deltas between the values. The entropy of both columns is high, since most values are not repeating each other.

In difference to the compression of the date format, the compression of datetime achieves bit savings with the conversion to the Unix timestamp and Delta compression. After the conversion to the timestamp, the maximal needed bits per value are 41, which is two 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, as explained in the date compression part of this section. Furthermore 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. This could be reduced by using a marker to encode the token and the length of a run with different numbers of bits. However, this would still result in a poor

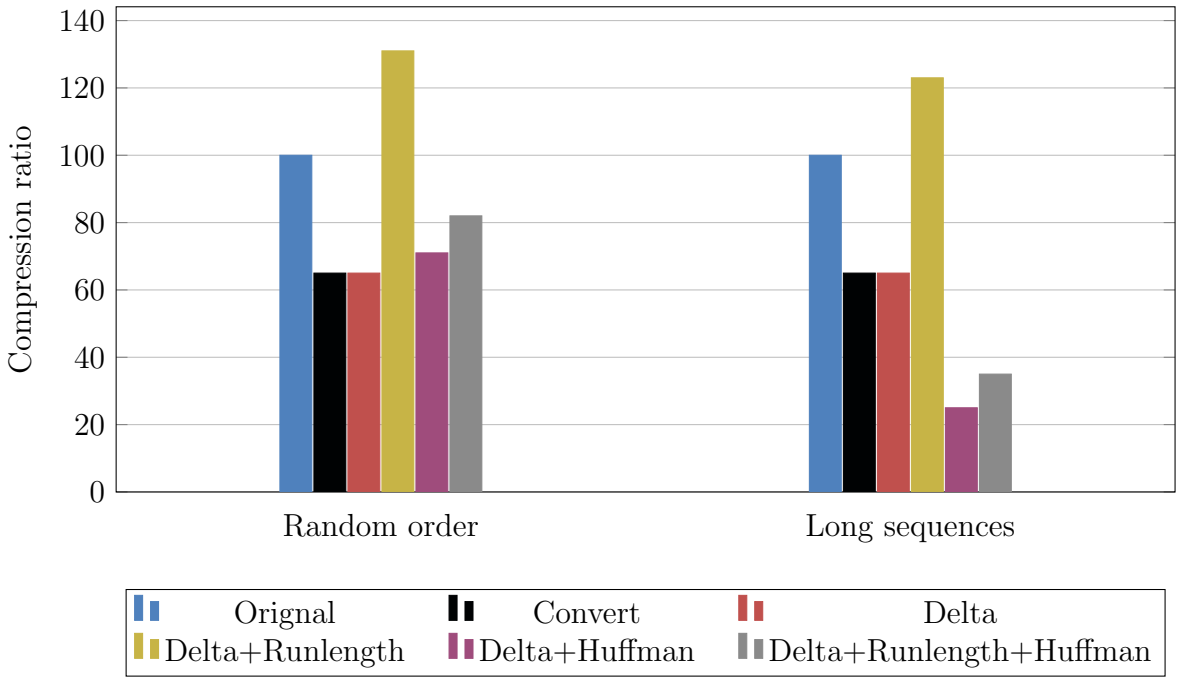

Figure 13: This chart shows the compression of datetime datatype. The compression ratios of the conversion to the Unix timestamp (convert) as well as the compression ratios of the compression methods are shown. The compression methods were applied to two data-sets. The first data-set consists of random unsorted dates with large deltas between the values. The second data-set consists of datetime values and small deltas between the values. The data-set has longer sequences of the same value.

compression, which requires more bits than the uncompressed data. Two values are still needed to represent a date and bits can only be saved when encoding the length of the run.

As mentioned before, the compression with Delta and Huffman achieves a slightly worse compression result than just delta. However 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 by 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 is 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, which makes a Run-length compression inefficient [19]. 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 that appear in text [20]. Assuming we look at the ascii table with only 94 characters, 7 bits are needed to encode all the values of the table. This means that in the worst case, the delta has

a value of 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 compression 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 (see Figure 1), this leads to a code word of a length between 3-4 bits. In addition however, a separator is needed between the individual encoded words, which also consists of 4 bits. The separator marks the end of the encoding of a value for further decompression with delta. For example, the delta values $2, 4, 7, 45, 3$ must be separated by a separator in the coding with Huffman. This is necessary because during decoding it is not known how many digits the next value consists of. Without a separator character, the sequence would be compressed with Huffman as follows: 247453. During decompression, the delta value 2 or 24 could therefore be read out, which would lead to incorrect decompression. Thus, in the best case with Delta+Huffman compression, the same amount of bits per value are required as in the uncompressed data.

This leads to the conclusion that only with a Huffman compression alone a profitable compression can be achieved. The Huffman tree for the compression of text is shown in Figure 14.

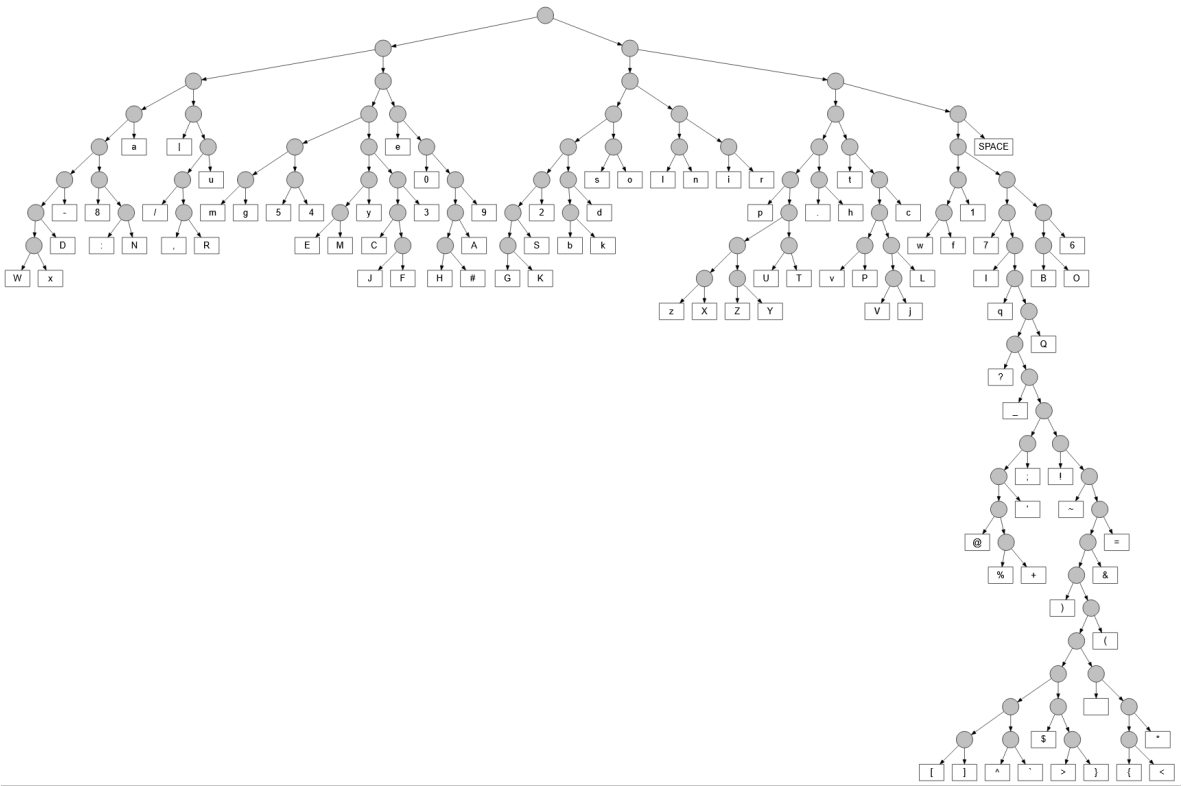

Figure 14: Static Huffman-Tree for compression of ascii values.

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 an 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 frequently and have less influence on the size. The goal is to use fewer bits on average than the original file.

In the database the number of characters per cell differs and the compression result with Huffman also generates a dynamic compressed size. Since Huffman also encodes prefix free, it is not known when the coding of the values in a cell ends and a new cell begins during decompression. To solve this, a separator is added to the coding. This marks the end of a database cell and at the same time the start of a new cell. This separator is not only used when encoding text but also when encoding numbers. This has already been described in the previous sections.

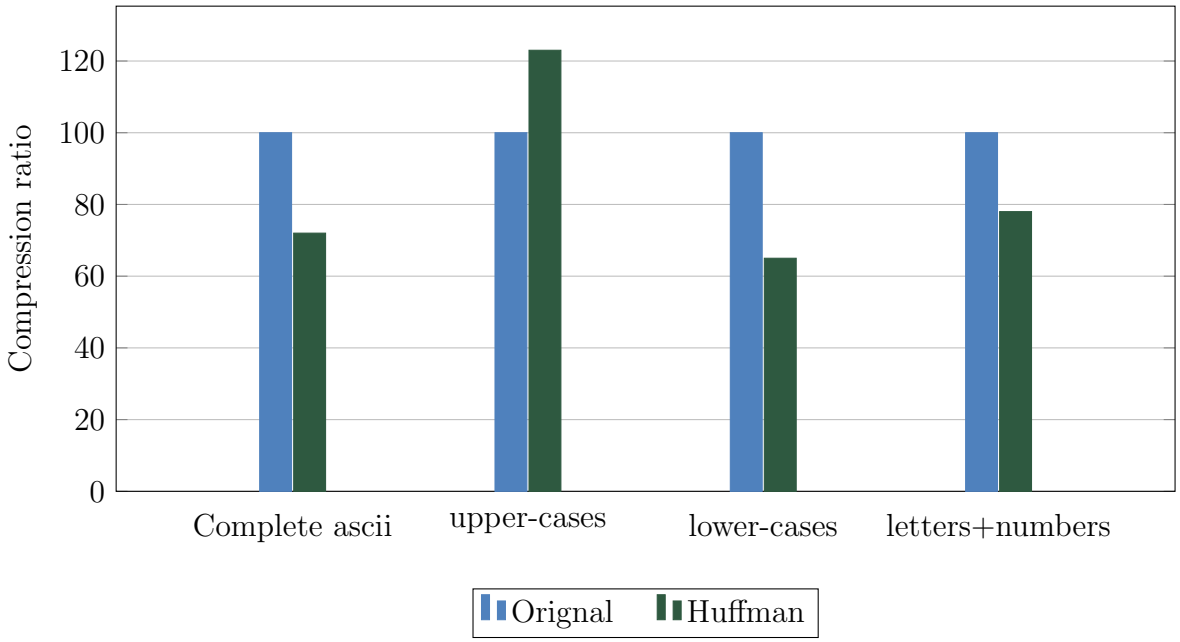

Figure 15: This chart shows the compression of text with Huffman compression. The compression method is applied to four data-sets. The first data-set consists of random unsorted values, where all 94 printable ascii characters are contained. The second data-set consists of only 2 characters per cell, where all characters are upper case letters. The third data-set contains only lower case letters. The last data-set contains the combination of letters with many numbers. All data-sets having an entropy of 4 to 5.

The Figure 15 shows the compression results form a Huffman compression with the static tree from above (see Figure 14). The compression was applied to four different data sets to investigate the behaviour of Huffman compression on different data. The first data-set represents the Location column in the user table of the Stack Overflow database. The values in that column are in a random order and consist of all different ascii symbols. The second data-set also is part of the user table and shows the result of the CountryCode column compression. Here, only 2 characters per cell are contained and these characters are all upper case letters. The third data-set represents the second

column of the part table in TPC-H database. The data only consists of letters of the lower case. The last data-set is the DailyAverageRelativeHumidity column form 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, which means that the characters cannot vary too much.

The compression results show that the compression with Huffman achieves an efficient compression in the most cases, where the compressed data size is between 60% and 80% of the of the original data. The compression of the second data-set achieves an inferior 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 achieved by using a dynamic Huffman tree, where the symbols are sorted and optimised to each column of the database specifically. However, 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 will be needed.

Overall, the compression with Huffman achieves a good compression result if the data consists not only of upper case or special characters. The best compression result, with the static tree used, is achieved when there are many lower case letters in the database cells. Since most of the lower case letters in the tree are at the higher levels, few bits are needed for coding.

4.6 Compression-Results

Overall, there is not one compression method that works best for all data types, but there is a pattern that is the best variant for the compression of numbers (int, float). It has been shown that the best results are achieved with the Delta compression as the first compression step. Further compression's with the Run-length or Huffman compression 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/datetime. After converting to the Unix timestamp, a compression with Delta+Huffman achieves the best compression result overall. For the compression of text, Huffman is the best 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.

The tables illustrates, that 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. Overall, we can reduce the size of the individual databases to

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
BZIP2	510.2%	142.5%	97.9%	25.2%	34.5%	50.7%	25.1%	33.7%
LZMA	423.0%	127.0%	89.8%	24.5%	26.9%	37.6%	26.3%	27.9%
Ours	30.7%	36.4%	35.2%	43.4%	58.4%	49.7%	40.2%	52.5%

Table 7: The size of the original uncompressed data (Stack Overflow benchmark) is given. The compression results of the individual techniques are given in percent. The percentage indicates the size of the compressed data compared to the uncompressed data(DB-Size).

Compressions of TPC-H									
	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
BZIP2	22.5%	17.8%	50.0%	15.8%	10.4%	14.8%	126.6%	26.6%	17.4%
LZMA	24.2%	20.1%	50.0%	18.2%	13.4%	17.5%	117.0%	29.0%	19.8%
Ours	53.3%	47.9%	65.0%	49.3%	47.6%	54.8%	52.9%	55.2%	50.0%

Table 8: The size of the original uncompressed data (TPC-H benchmark) is given. The compression results of the individual techniques are given in percent. The percentage indicates the size of the compressed data compared to the uncompressed data(DB-Size)

Compressions of weather database						
	Bristol	FAYETTEVILLE	Fort-Smith	Nashville	Norfolk	Total
DB-Size	6.2MB	7.1MB	9MB	8.2MB	8.3MB	38.8MB
BZIP2	5.3%	4.9%	4.8%	5.4%	5.0%	5.0%
LZMA	6.5%	5.9%	5.5%	6.3%	6.0%	5.8%
Ours	46.7%	47.8%	48.8%	48.7%	46.9%	48.1%

Table 9: The size of the original uncompressed data (Weather benchmark) is given. The compression results of the individual techniques are given in percent. The percentage indicates the size of the compressed data compared to the uncompressed data(DB-Size)

around half their original size by compressing them using the methods explained above.

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 this 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 only needs a few bytes.

5 Database Decompression

The 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 analysed in decompression. Methods 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 on the FPGA to keep the data as close as possible to the processing unit. From there, the values are read out in 32-bit words and entered into the decompression part. The decompressed data is then written to another BRAM, also in 32-Bit or 8-Bit words.

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

All the decompression algorithms are implemented modular, so they can easily 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 a 'next_one_ready' signal as input and a 'ready' signal 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. The previous step holds the current value for that time. This means that no data is lost between the steps. In addition, each decompression step signals to the next one whether the currently available data is valid. This prevents a potentially faster decompression step from decompressing the same data multiple times, which would lead to incorrect decompression results. The individual decompression steps only decompress if valid data is available. 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 for numbers or 8-Bit 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. The maximum frequency at which decompression can be run on the FPGA is calculated by the following formula:

$$Max\ Frequency\ f = \frac{1}{T - WNS}$$

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 the maximal frequency the decompression can run, given in MHz.

Using the determined maximal frequency, the worst-case throughput as well as the best-case throughput is calculated. Therefore, the output, 32-Bit or 8-Bit, is taken as

size of the decompression result. Depending on the number of clock cycles required to decompress a value, the throughput is calculated. The number of clock cycles per value can not only vary between the individual decompression chains, but also 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 clock cycles to decompress one value, b is the size of the output. As a result, the throughput is given in M words/s.

5.1 Delta-Decoding

For the decompression 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. Furthermore, there is no need to make any losses in the compression ratio due to quantization steps in compression, by using just few Bitpacking decoders with fixed sizes. The bits used to represent the numbers within a column are 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 the compression small. This value is then used as input for decompression during Bitpacking and configures the sliding window size.

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 applied to this vector. The remaining bits are the MSBs and are completed with missing bits from the new value. This means that encoded values that have a length that cannot be divided by two bases can also be decoded (see Figure 16).

The Bitpacking decoder can emit a new decoded word with each clock cycle, which then can 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 17. After the 'Input_Data' value has changed, the output becomes invalid for 3 clock cycles. In this time, the new value is loaded onto the Bitpacking decoder and a new value is built for the following decompression.

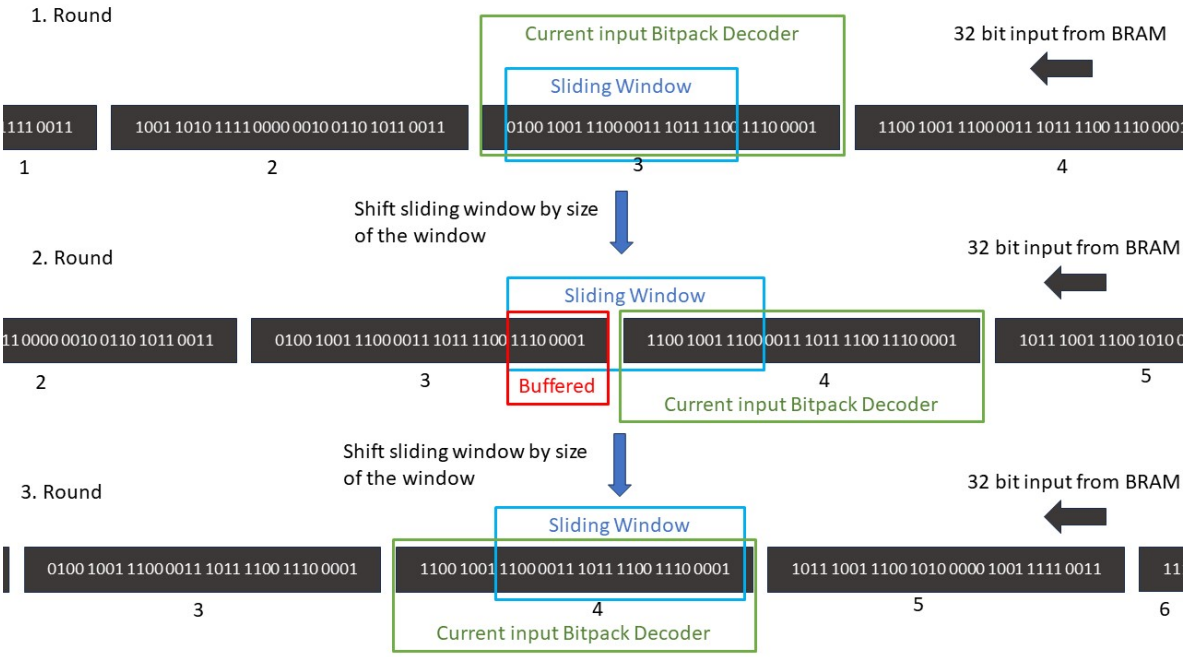

Figure 16: Bitpacking decoder with a sliding window size of 24 bits. The sliding window is iterating over 32 bit inputs from the BRAM and extracting 24 bits. After each extraction the sliding window shifts by its size. The Bitpacking decoder buffers the remaining bits when the current input does not contain enough bits to fill the sliding window. It then loads a new input.

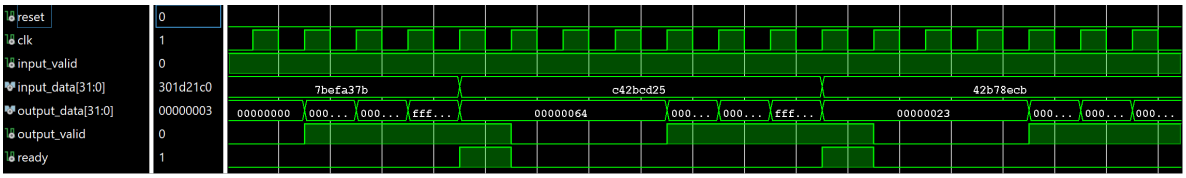

Figure 17: Bitpack-Delta Decompression with a small sliding window size.

Figure 17 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.

The decoding with bitpacking is followed by decompression with delta. The implementation of the delta decoder is simple. If the current input is valid, then that input number is going to be added to the previous number with each clock cycle. At the beginning of a decompression run, 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 decoder calculates the twos complement. By this, the delta decompression only needs to add up the output of the Bitpacking decoding.

As Table 10 shows, with this decompression chain, a throughput of around 800MB/s can be achieved in the best case. This result is possible if the sliding window's size is 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 clock 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 32 bits that throughput is 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. Table 10 also shows the

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 results on FPGA.

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. However, 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. 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 floating point 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 number must be stored in the compression header for each column of type floating point. 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 many clock cycles are needed, only bad decompression throughput is achieved. Therefore, a back conversion from integer to float on the FPGA is inefficient and should be done on the remaining CPU-subsystem.

5.2 Delta-Runlength-Decoding

The decompression with Delta and Run-length is based on the previously explained Delta decompression. Between the Bitpacking decoder and the Delta decoder, the Run-length decoder is added. However, Run-length 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 from the compressed data 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

the Bitpacking decoder. The Run-length 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 emitted for the corresponding number of clocks cycles. In this way, the Run-length decompression can emit one 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 Run-length. Therefore, more than 1 clock cycle per value is needed. This can be seen in Figure 18. The change of the input value in the Bitpacking as well as the loading of two new inputs to the Run-length 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 18 are related to the further decompression with delta. This also leads to the gap between the first decompressed value and the following once, even if the value is not changing in between.

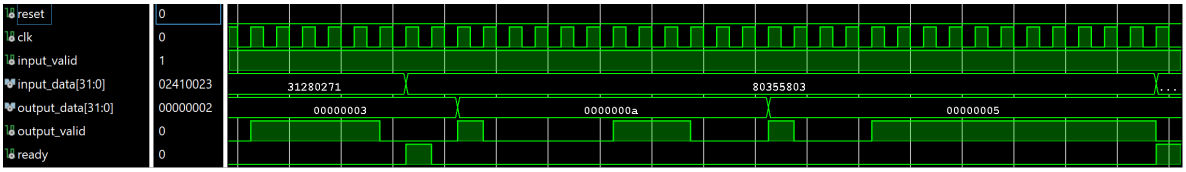

Figure 18: Bitpack-Delta-Runlength with a small sliding window on FPGA.

As the Table 11 shows, the worst-case and the best-case decompression throughput are worse than just the decompression with Bitpacking and Delta (see Section 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, values are decompressed with a frequency of 40MHz. This results in a throughput of 160MB/s for 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 clock cycles 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 Bitpacking. 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 Run-length 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 algorithm, more clock cycles are needed for the decompression, leading to a worse result.

The additional decompression method is also reflected in the consumption of FPGA resources. Energy consumption has slightly increased. Furthermore, around 50 more LUTs are used compared with just the Delta decompression. The number of flip flops

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 results on FPGA

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 not different to the Delta decompression.

As with the delta decompression before, the decompression of floating point 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, as mentioned earlier, the converting back to float is more efficient on the remaining CPU-subsystem.

5.3 Huffman-Decoding

The Bitpacking decoder is not needed for the compression with Huffman. Huffman is always the last step in the compression and the words are encoded as already described in the compression section. 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 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 the decoder Tom Wada described in his work [21]. 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 as 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 encoded words, the sliding window cannot always be moved by the size of the sliding window, as it is the case with Bitpacking. The sliding window can contain several words, which would then be lost. The moving of 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 therefore can be used to decode all columns in the database. Given the advance that the static tree 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). One 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 bits) are the encoding of the next word. The bits in the sliding window are then given as input to the first level LUT. The LUTs then check 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, the search is carried out in the next lower level. Then the 3 MSBs are considered. 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 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 emitted every clock cycle with Huffman decompression. This can be seen in Figure 19. As with Bitpacking, 3 clock cycles 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 due to the smaller sliding window.

In Figure 19 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 14) is 19-bit deep, which leads to frequent loading of new values from the BRAM.

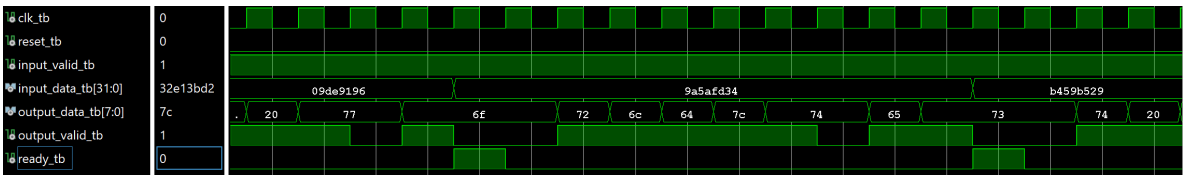

Figure 19: Huffman decoding of text on the FPGA

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 decoded word. This takes 3 clock cycles, which means that values can only be decoded every 3 clock cycles.

As can be seen from Table 12, the Huffman decoder on the Zynq UltraScale+ ZCU106 Evaluation Platform can be clocked faster than the decompressions based on Bitpacking. 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 with Huffman is achieved. Since the text is emitted in ASCII encoding, only 8 bits are used for the encoding output. This results in the most inferior throughput of 140MB/s.

In opposite, if only the encoded values from the top level appear in the same tree, the best case throughput is achieved. The top level characters are encoded 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 emitted at frequency of 300MHz. Having a 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 that 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 results on FPGA.

The resources required for the Huffman decoder are also lower. This means that slightly less energy is required than with Bitpacking. 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 were also reduced. Since the decompression does not require any input for the size of the sliding window, 6 bits of input can 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 a 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 after reaching 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 6 possible values left to code functions without expanding the tree further. But only 2 functions are needed for decoding. 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 as connecting each digit requires 1 clock cycle. This means that as the length of the number increases, the required clock cycles to output a decompressed value increases and the throughput decreases.

Figure 20 shows the decompression of numbers of different lengths. It can be seen that, 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 by further decompression steps such as Delta or Run-length.

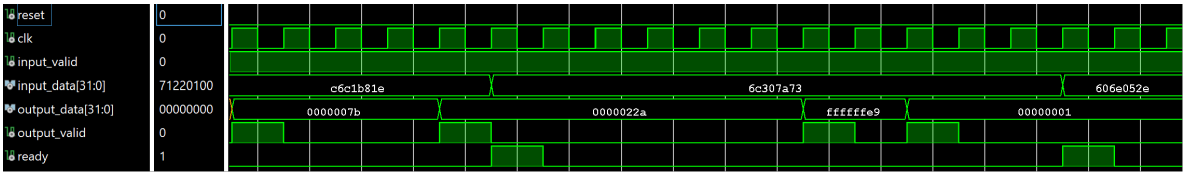

Figure 20: Huffman-Numbers decompression on the FPGA.

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 clock cycles are needed to decode the number and to put it together. In addition, 1 clock cycle must be added to process the separator and emit 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, leading to an output frequency of 28MHz for decompressed values. 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 emitted every two clocks or with a frequency of 210MHz. If the value is emitted with an output size of 32-bit, the best case throughput reaches 840MB/s. However, integer values that are only in the range 0-9 are rather rare in the benchmark databases.

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 method 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 decoding. 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 emit 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 decoding does not require an additional clock cycle, because while the number concatenation takes at least two clock cycles to assemble a new number, the Delta decoder can decompress the current value during this time. This works because the Delta decoding only requires 1 clock cycle. This can be seen in Figure 21. 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.

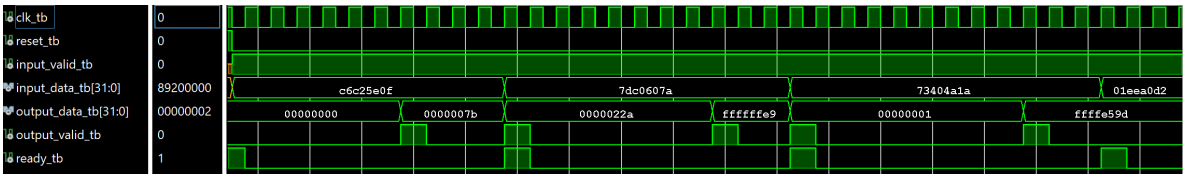

Figure 21: Huffman delta decompression of numbers on FPGA.

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 it is 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. By using Delta Huffman, the compression rate can be increased without requiring a long decompression time.

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. Booleans 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 (explained in Section: 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 also acts as 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 specialised 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 decoder. The first input value for the decoder is either '1' or '0' and indicates whether the initial value is True or False. Following the run length of the first value is the input for the Run-length decoder. Since boolean can only assume two states, we don't have to specify the token for decompression, only the run length. This means that the compressed file does not consist of tuples with two numbers but only a list of run length numbers. Therefore, the special variant of the Run-length only needs one input and only one value has to be extracted with Bitpacking to start the decoding. 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 clock cycles that were entered as input. This can be seen in Figure 22. This also means the longer the run length, the higher the throughput, since new values have to be read from the BRAM with Bitpacking less frequently or new values have to be extracted from the 32-bit word.

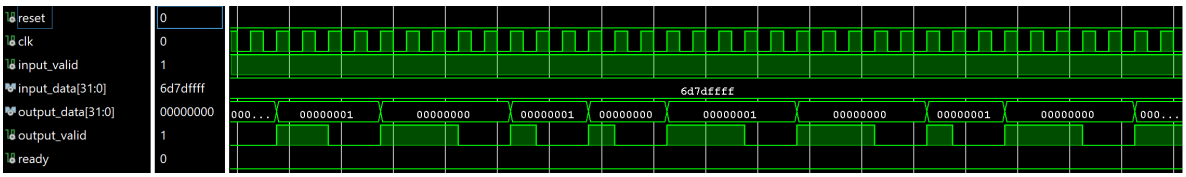

Figure 22: Bitpack-Runlength-Bool decompression on the FPGA.

The change between two values takes 2 clock cycles. During this time, the new value is read and entered into the Run-length decoder. In the worst case, only a run length of 1 is decompressed. Decompressing this run length then requires one clock cycle. This together with the 2 clocks from changing, results in 3 clock cycles to decompress a value. The decompression with Bitpacking and Run-length can be carried out on the board at 300MHz. So in the worst case, decompressed values are emitted 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 runs, 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 Run-length 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, each of the benchmark databases was able to be compressed to around half of their original size. However, the baseline compression with 7Zip could not be outperformed. The time necessary 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. Because 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 one hand, the decompression with a Bitpacking decoder and on the other hand the Huffman decoder. The two models differ mainly in the maximum possible frequency 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 compressions achieve a similar ideal throughput when it comes to the datatype they are optimal for. 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. However, when decompression numbers with Huffman, 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 Run-length 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 encoding the words by using the Bitpacking decoder. For decompressions based on Bitpacking, in the worst case scenario, 3 clock cycles are required for Delta decompression or 6 clock cycles for Delta Run-length decompression. So with Huffman, the word to be compressed can not be longer than 3 or 6 characters in order to achieve a higher performance. If the numbers have more than this number of digits on average, the Huffman achieves a poorer throughput because putting the numbers together takes longer. This also includes the minus sign and the separator of the table cell. 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 for 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 Run-length. However, the reverse conversion from integer to float on the FPGA is very time expensive in the calculation. So it is more efficient to do this on the remaining CPU subsystem. Compressing floats with Huffman achieves an inferior 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 one hand pure Bitpacking and on the other Bitpacking by specialised Run-length. Both methods achieved similarly good compression ratios on the benchmarks. The Bitpacking method achieved a constant throughput on the FPGA during decompression. The values in Bitpacking 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 clock cycles 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 Run-length 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 emits decompressed values with a clock rate of 1.0625. Decompression with Run-length only achieves this value in the best case if the Run-length corresponds to the largest number that can be represented with 32-bit. For all other cases this decompression method achieves worse ratios.

When compressing dates (YYYY-MM-DD) using Huffman, at least 8 characters must be encoded if the year, month and day are compressed separately. As each value is encoded with 3-4 bits, the encoded representation requires 24-32 bits. This corresponds to the uncompressed size. Datetime (YYYY-MM-DD hh:mm:ss.nnn) consists of longer strings of numbers and characters. If the fields are considered individually, 17 characters must be compressed, which leads to a compression result of 51 - 68 bits with Huffman. It therefore depends on the distribution of the characters whether Huffman achieves

a compression effect. For the compression with Delta, the values were converted into integer timestamps. Where the conversion of Datetime reduces the size of the data to 66% of the original size. Further compression with Delta does not bring any further improvement as the timestamps have a similar size and the initial value therefore already specifies a large number of bits with which the values are encoded. If the initial value is not considered in the compression and the deltas between the values are small, a good compression ratio can be achieved. Moreover, compression gains can be attained when encountering sequences of identical dates/datetimes within the data-set, enabling compression into fewer values through Run-length encoding. The utilization of both Delta and Huffman compression can further enhance the compression ratio. In scenarios where deltas mainly consist of a few digits, and outliers contribute to increased bit requirements in delta compression, Huffman compression helps mitigate this overhead, resulting in a significantly improved compression ratio.

The decompression of date and datetime on the FPGA is then analogous to integer decompression. Decompressions 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. But with many repeating values the deltas are small and also provide a better compression ratio with Delta-Huffman. With a few large deltas, Huffman decompression can also achieve a good throughput. 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 compression was considered. With the static Huffman tree used, different compression rates could be achieved depending on the characteristics of the data. The 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 located 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 rarely occur, so frequent values can be higher up in the tree and have shorter encodings than in the balanced tree.

If resources need to be saved, good throughput can be achieved with Huffman-based decompression. However, in most cases the compression with Huffman 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 decompression chain.

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

7 Conclusion

In this work, five basic methods for compressing databases were analysed. The aim was to find out which compression method can be used to obtain the highest possible compression ratio on the benchmarks. Furthermore, the decompression of the data should then generate a maximum throughput on an FPGA with as few resources as possible in order to supply the remaining CPU subsystem with valuable data. The compression time is neglected.

With the basic compression methods presented, we were able to compress the benchmark databases to half their original size. However, it did not achieve the compression ratio of 7Zip. However, a high throughput can be achieved when decompressing the data on the FPGA. Bitpacking and Delta decoding achieve a throughput of up to 800MB/s in the best case, while the worst case still achieves a throughput of 320MB/s. With around 1500 LUTS and 320FF, the decoder required less than 1% of the available resources of the test board. With further decompression with Run-length, the throughput drops to 160MB/s - 560MB/s. However, this requires hardly any more resources than delta decompression. This method has proven to be the most efficient decompression of numerical values. It is also used to decompress the converted date and float values. However, converting these values back is difficult to implement on the FPGA and requires several clock cycles, which reduces throughput significantly. Therefore, re-conversion on the CPU system is recommended.

For text, the Huffman decoding has the highest throughput. A throughput of 140MB/s to 300MB/s is achieved for a decoding output of 8 bits per value. In an ideal scenario, Huffman and Delta decoding for numbers even achieve higher throughput than Bitpacking and Delta. However, this is only achieved in the best case, when the number only consists of one digit, which rarely occurs in databases. Huffman decompression is also significantly more resource-efficient than Bitpacking-based decompression. It can be implemented with a few hundred LUTs and FFs. Therefore, our Huffman algorithm can also be used on smaller FPGA boards.

We have shown that databases can be reduced to about half their original size using simple compression methods and then decompressed on an FPGA with a high throughput of up to 800MB/s.

As an outlook, further compression methods such as LZSS could be considered to improve the compression ratio. Furthermore, a fail-safe can be developed for compression so not the entire file is lost in the event of compression errors. These fail-safes could also be used as a quick entrance for filtering, optimising the performance of database access.

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] B. L. Malcolm Singh, "Introduction to the ibm netezza warehouse appliance," *CASCON '11: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011.

[4] P. Francisco, "Ibm puredata system for analytics architecture, a platform for high performance data warehousing and analytics," *IBM Redbooks*, 2016.

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

[6] 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.

[7] S. K. N. Pralhadrao V Shantagiri, "Pixel size reduction loss-less image compression algorithm," *International Journal of Computer Science Information Technology (IJCSIT) Vol 5, No 2,*, 2013.

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

[9] 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.

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

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

[12] 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).

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

[14] 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).

[15] "Ieee standard 754 floating point numbers," Available at https://upload.wikimedia.org/wikipedia/commons/e/e8/IEEE_754_Single_Floating_Point_Format.svg (2023/11/10).

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

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

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

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

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

[21] 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).