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

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

September 30, 2023

Abstract

Your abstract.

1 Introduction

In 2020 64.2 Zettabytes of data are generated and saved in databases. Till 2025 this number increases up to 181 Zettabytes. Internet usage, social-media, research and financial market are some of the branches creating the most digital data. They create big databases with millions of data sets. Google is as a well known search-engine highly frequented. Nearly 100.000 search-requests per second are inquired alone on this platform. Therefore a huge amount of data is needed to be moved through the system. The storage and computation of this huge amount of data is energy intensive. The moving of data in the system is often much more power consumptive than the calculations itself, especially the movement and filtering of databases is energy expensive.

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 on the data inside storage would have an positive effect on the energy consumption. We could increase the amount of data that could be stored while saving throughput on the communication channels. This means we can keep the channels busy with filtered data and don't need to move raw data.

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

This paper deals with question which (de-)compression techniques is the best method to compress and decompress databases. The methods are investigate with the respect of the capability to be implemented on FPGAs for delivering high throughput requirements. Therefore we analyze existing compression techniques. We investigate the compression ratio as well as the compression and decompression time needed for a small

example database. This shall provide us a better understanding of (de-)compression requirements of different workloads and the FPGA implementation cost performance-tradeoffs for different tailored compression techniques.

We used two test database. The databases containing different data-types and multiple tables. For compression, we have limited ourselves to the data: float, integer, boolean, date, datetime and text. The first database is created by TPCH. That database represents a warehouse and is filled with random generated data. This test-database is a size of 1GB. The second database corresponds to the 'stackoverflow' database. In this database the data is closer to real data. By changing and adding some data-types we increase the variety of data-types and can build up a more complex database. This database is with 411MB smaller than the first one.

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

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

2 (De-)Compression Algorithms

Target of data compression is to reduce the needed bits to represent a symbol or number. By reducing the bits per symbol/number, more data can be stored at in the memory without increasing the memory capacity. Furthermore, more data can be transferred between systems without increasing the existing bandwidth. The usage of compression and decompression requires computational calculations. The compression and decompression of data can be pretty computational expensive. Depending on the used algorithms the costs differ. While some methods are expensive in compression, they can be very cheap in decompression. This is useful to keep static data in a compressed status in the memory and using the fast decompression algorithms, if that data is required. Since the channels between memory and computation are often the bottleneck in the calculation part it is desirable to keep the memory as close as possible to the computation part. So we can receive the best bandwidth usage and the maximum utilisation of the system.

We are storing the data in the memory of an FPGA in a compressed way. Using simple decompression algorithms implemented on the FPGA we want to reach the maximal bandwidth usage and keep the channels busy with valuable data. This decompressed data can than be used by a CPU-subsystem for further calculations.

In this work we decided to use simple and well known lossless compression techniques. Where the runlength compression reduces the size of the database by grouping symbols together and thus reducing the number of symbols needed to represent the database. Delta with Bitpacking and Huffman are compressing-algorithms which are

reducing the number of bits used to represent a symbol. It is also possible to combine multiple compression-algorithms. But with increasing number of used compression-algorithms the compression-time increases too. Since our focus is on a fast decompression on an FPGA, the compression time is not important. Depending on the used compression-algorithm the encoding and decoding can differ. Some algorithms are quite slow in the compression but fast in decompression.

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

- compression ratio: This is a metric used to evaluate the effectiveness of a data compression algorithm. It is defined as the ratio of the original data size to the compressed data size. The compression ratio indicates how much the data has been reduced or compressed by the algorithm. The ratio is a number between 0 and 1. A small ratio indicates a effective compression. The ratio is calculated like in the formular below.

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

- decompression time: The time needed to decompress the data and receive a high throughput. The compression time is not investigated, since the focus is on the fast decompression. Therefore the compression checks for several possible options to find the best fitting compression method. The decompression is done on the FPGA and the time needed to decompress is analysed.

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

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

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

2.1 Run-length

Run-length encoding (RLE) is a lossless data compression technique used to reduce the size of data by replacing consecutive repeated occurrences of the same value with a single representation of that value followed by the count of its occurrences. The combination of value and occurrence is called tuple. A tuple is in the form (t,l), where t is the value or symbol and l the count of the repeating occurrence of that value. The fundamental idea behind RLE is to exploit the redundancy in sequential data, especially when there are long sequences of the same value or pattern.

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

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

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

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

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

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

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

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

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

2.1.1 Run-length for Boolean

Boolean values can be effectively compressed using a specialized version of run-length encoding tailored to their binary nature. As boolean values can only take two states, true or false, we can simplify the compression approach. In this variant, we exploit 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 since it is either true or false. In this variant we count the occurrences in a consecutive sequence of trues or false. The counts are directly added to the list

of compressed data. With each new number in the list the values switches from the previous state to the opposite state.

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

2.2 Delta-Encoding

Delta-Encoding is another simple and lossless compression algorithm used to reduce the size of data by encoding the difference between consecutive elements in a sequence rather than encoding each element independently. That method calculates and represents the changes (deltas) between adjacent elements in a sequence. Instead of storing each element in its original form, delta encoding stores the differences or changes between neighboring elements.

As an example, let's consider the sequence of integer values $[10, 12, 15, 20, 22, 25]$. We can compress this sequence using delta encoding, which involves calculating the differences (deltas) between consecutive integers: $deltas = [2, 3, 5, 2, 3]$.

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

However, the largest delta in the sequence is 5, 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, 10, requires 4 bits to represent. Therefore, the compressed data, including the 5 deltas and the initial value, can be represented with a total of 19 bits.

$$compression\ ratio\ \eta = \frac{19bits}{30bits} = 0,63$$

The compression ratio is approximately 0.63, meaning that the compressed data is about 2/3 of the original data.

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

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

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

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

2.3 Bitpacking

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

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.

For example, if we have a dataset with values ranging from 0 to 15, we could represent each value using only 4 bits ($2^4 = 16$). With such a sequence of values, bitpacking allows to store several values within a single byte (8 bits), efficiently using the available space and reducing memory or transfer overhead.

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

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

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

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

2.4 Huffman

Huffman encoding achieves lossless data compression by assigning new codes to symbols. To accomplish this, a dictionary containing the symbols intended for compression is employed. Each symbol in this dictionary is associated with a distinct code, varying in

length. The symbols are organized within a binary tree. Symbols with higher frequency in the data being compressed are positioned higher in the tree, necessitating fewer bits for their representation. Conversely, symbols occurring less frequently are positioned in the lower levels of the tree, which might result in a greater number of bits needed for their representation compared to the original uncompressed data. As new symbols are added, the tree expands accordingly.

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 the binary tree is pre-generated and is used to compress all the data. Means the huffman tree is build up not knowing the data which is going to be compressed with the tree. The huge advantage of the static tree is that the tree don't needs to be included in the compressed data. Which reduces the size of compressed output.

2.4.2 Dynamic-Huffman

The dynamic huffman creates the binary tree while compressing the data. The algorithm counts the occurrence of each symbol in the uncompressed data, depending on that occurrence the symbol is ordered in the level of the binary tree. With the dynamic tree a better compression ratio can be achieved than with the static variant, since the binary is optimized to the data. 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. They should represent all possible cases in databases to get compression/decompression results for all characteristics. To test and validate the compression and decompression algorithms and find the best combination of methods to receive the highest data throughput on FPGA, three data-sets are used. Each data-set is freely available and consists of multiple tables of different sizes. The tables in a data-set having multiple columns with different data-types per column. The data types are limited to the following: Integer, Float, Boolean, Strings, Date and Datetime.

The data-sets were optimized before compressing. Due to data protection, some columns in the public database were left blank. Since this does not reflect the real database, these columns were either filled with randomly generated values in the optimization or the column was removed from the data-set.

The characteristics of the data in the tables differ also. In order to obtain results that cover a variety of different data and data characteristics, several tables and databases are used that differ in the data. To get an overview of the data characteristic in the tables of the databases the entropy is calculated:

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

H(x) is the entropy over a column in the database and p(xi) is the probability of each element in the column to occur. With higher probability the entropy is smaller. A small entropy indicates an ordered data set with a few different values. The greater the entropy, the greater the disorder in the data-sets. A high variety of values in a data-set create a small probability and a high entropy. This can be an indicator the 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. Because in the later compression the individual characters and not the entire content of the cell as one are compressed. The characteristics of the data-sets and some reference compressions with 7Zip are described in the following subsections.

3.1 Stackoverflow-Database

The stackoverflow database is a free part copy of the database from the website stackoverflow. What means that the database contains real data and real data-structures, what makes this a benchmark to test the algorithms on a real database. The database contains tables of votes, users, posts, postLinks, badges etc. Where we decided to use: badges, users, linkTypes, postLinks, postTypes, votes, voteTypes. All user contributed content is anonymized in that data-set, so no clear names, passwords or emails are contained. These leads to empty columns in the data-set. To receive a complete database without empty columns, the columns are filled with random values or are removed. The selected tables present a variety of different data-structures as well as different sizes of the tables and all data-types that are going to be investigated in the compression and decompression. Furthermore there are several sizes of the data-set available, this makes it possible to increase the data-set if necessary. In the following table the entropy of the database is shown. Each column of each table is given with its datatype and the entropy of the column.

Entropy of Stackoverflow 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:2.91
						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

For all seven used tables of the database is the entropy of each column and the average entropy of the table calculated. As the table shows, the database consists of different entropy's in a width range. The data-set includes columns with many repeating numbers as well as columns with many unique numbers. The same behavior counts for dates. The entropy of text-columns is in the most cases low. This is because in text-columns the maximal variety of different elements can be 128 in the ascii representation. After reaching the maximal number of different elements the total probability remains the same, just the internal probability of the single elements changes. Leading to a small entropy, since less different elements repeating more often with increasing text length.

The first three tables a small tables with a less data. They reach a small entropy and so a small information density. The remaining tables including a lot of data and reach a higher entropy, leading to a higher information density. The data-set contains many ordered and unordered data. Thus, many data characteristics can be covered.

The stackoverflow database can be opened in a SQL environment. From there we exported some of the provided tables and data to CSV-Files which are used to be compressed. We are using CSV since the data-structure of the database is not being lost with this datatype as well as the access to the file in python is quite easy. The exported data in the CSV files has a size of 394MB. Whereas the database size is with 232MB smaller.

Reference compressions of Stackoverflow								
	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

The table shows the size of the original data (database-size) and the memory needed per table. The original size is calculated by the default amount of bytes that is used to represent the corresponding datatype in a database multiplied by the number of entries per column. Furthermore the table contains the compression of the single tables and the complete database with 7Zip. The compression is fulfilled with two different compression processes on the exported CSV-file. Therefore the size of the exported CSV-file is also given. In the BZIP2 compression a sliding-window size of 900KB used. This is the highest value that can be selected in the compression. The second compression algorithm used in 7Zip is LZMA. This method achieves a better compression result than BZIP2. Here a sliding-window size of 16MB is used. But for the smaller tables like LinkTypes or postTypes the compression result is just slightly better. However, with both methods the compressed files of the small tables are larger than the originally uncompressed files. The best compression ratio can be achieved in the compression of the vote-table.

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

3.2 TPCH-Databse

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

Entropy of TPCH 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:13.2
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:6.9

Like the entropy table above shows the data-characteristics differs like in the stack-overflow. The database contains many unorganized data as well as organized once. But especially in the text they differ. The stackoverflow database contains real words like a real database would it containing. Whereas the TPCH database mostly contains randomly generated strings without a relation to real data-sets. This benchmark also contains except of the datetime all data-types.

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

The table shows the size of the database and its tables. The TPCH database consists of multiple tables with different sizes. The largest table is LineItem which accounts for three quarters of the entire database. Like before the table gives the size of the database in the db-size, as well as the size of the exported tbl files. But in this case the sizes of the both uncompressed databases do not differ. The compression with 7zip is also executed with the BZIP2 and LZMA compression methods. In nearly all cases the compression can reduce the size of the table. The BZIP2 compression achieves a better compression result over the complete database than LZMA.

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

3.3 Weather-Database

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

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

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

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

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

4 Database Compression

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

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

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

not taken into account. On the opposite data with a high entropy can result in good compression ratio if the data is sorted, like consecutive increasing IDs.

4.1 Integer-Compression

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

The main target is to reduce the number of bits used to represent the data. Therefore the number of bits used to represent a single integer needs to be reduced. On FPGAs the operations for the decoding are a fulfilled in parallel on hardware. Having the same bitlength for each number improves the hardware efficiency, as less logic is needed. To decoded all values with the same bitlength, the largest number after delta or runlength compression is used. The minimal needed bits to represent the value of that number is calculated and an additional bit for the sign is added. All values of the compression result are encoded with that amount of bits. This method entails a variable bit length between the individual columns. Again leading to a higher hardware usage and more complex logic on the FPGA. Another option would be the usage of multiple quantization steps. By defining multiple fixed bitlengths, the hardware on the FPGA could be kept simple. Therefore multiple decoders of different bitlengths would need to be implemented. Such quantization steps would also result in a worse compression ratio since the overhead of leading zeros would increase. In our tests we use the calculated bitlength, since the compression ratio is better.

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

This is a balanced tree with the maximal depth of 4 bits. With this tree each character 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 characters, including the sign, can be compression by huffman and reaches a compression result smaller than the original data. However for numbers larger than 8 characters the huffman compression creates worse compression ratio. Furthermore small values the huffman compression can also generate worse compression ratios. Since with huffman compression a dynamic bitlength per number is created, an additional separator is necessary. This separator marks the end of a number and the start of a new number.

To find the best compression algorithm for integers we analysed the compression with delta and huffman, as well as the combination of them and an additional runlength 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 delta can be reached, resulting in less needed bits.

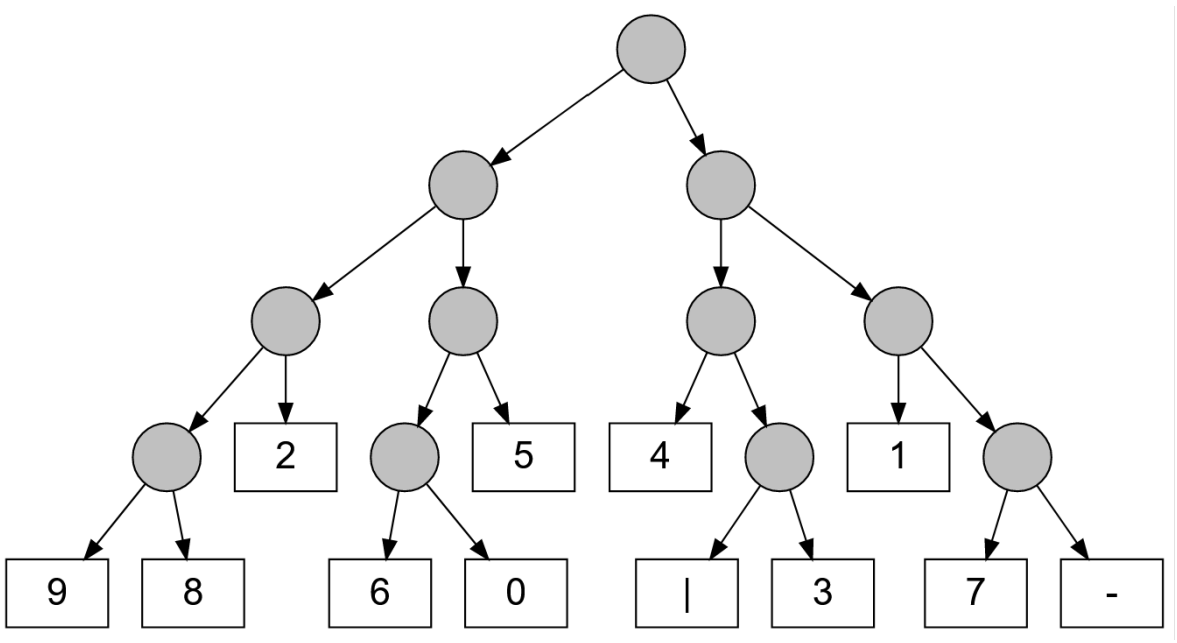
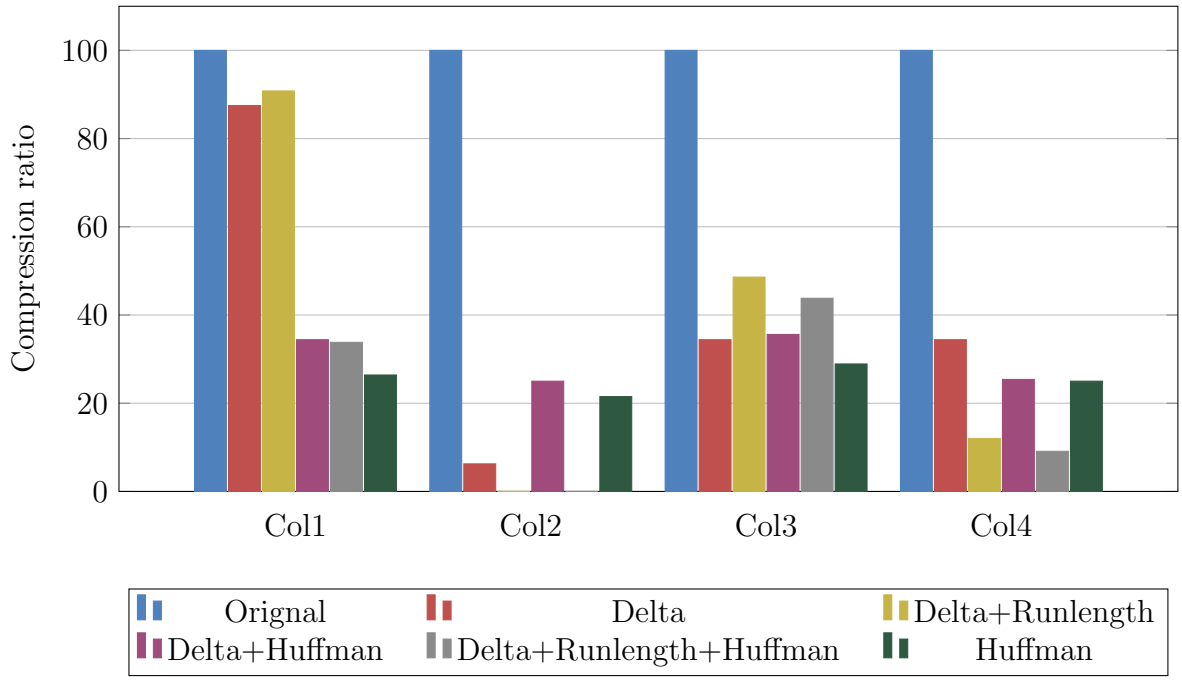


Figure 1: Static Huffman-Tree for compression of integers

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 runlength should achieve the best result.



In Col1 the normalized compression ratio of VoteTypeId of the Votes table in the stackoverflow database is given. This column is having a small entropy of 3.91. The most values are varying in a range of 1 to 3, with a few outliers. Furthermore the data is structured in lists with repeating values. Contrary to the hypothesis, a poor result is

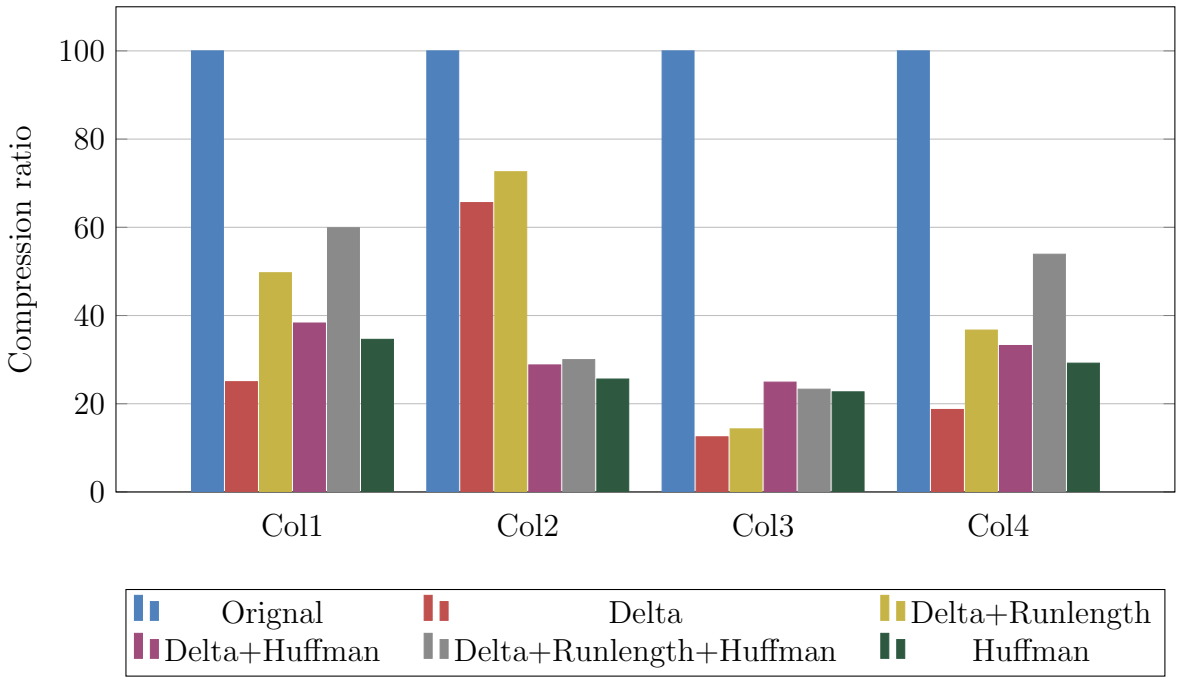
achieved with delta and runlength. Due to the large outlier the maximal number in the compressed data after delta is quite hight, resulting in many bits needed to represent all the values. Since in runlength the token and the length are both represented with the same amount of bits, the number of bits is doubled per tuple. Because the repeating sequences are to short and the values changes to often, the compression ratio gets worse. The break-point, at which a runlength compression achieves better ratios is reached when the average value change is smaller than 0,5. However, this is dependent on the value of the length not being greater than the value of the token. With the more dynamic huffman compression the size can be reduce quite well, since the most values are small and only needs 3 to 4 bits to be represented.

Col2 contains the compression ratio of the column LinkTypeId in the postLinks table of the stackoverflow 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 good result, since the delta is for all values except the first one zero. With runlength at the compression ratio can be improved further. The amount of numbers in the compressed value is reduced to 4 values. 2 values for the initial value and 2 values for the token and the length. The huffman compression on top of that can't improve that compression further. Specially the delta+huffman combination receive worse results, since for each value 3 to 4 bits are needed even its just a array of zeros.

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

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

When dealing with unstructured data but low entropy the compression still should achieve a good compression ratio. But in difference to the case before the compression with delta should reach a better compression ratio than the the combination of delta and runlength since there are too many changes between the values.



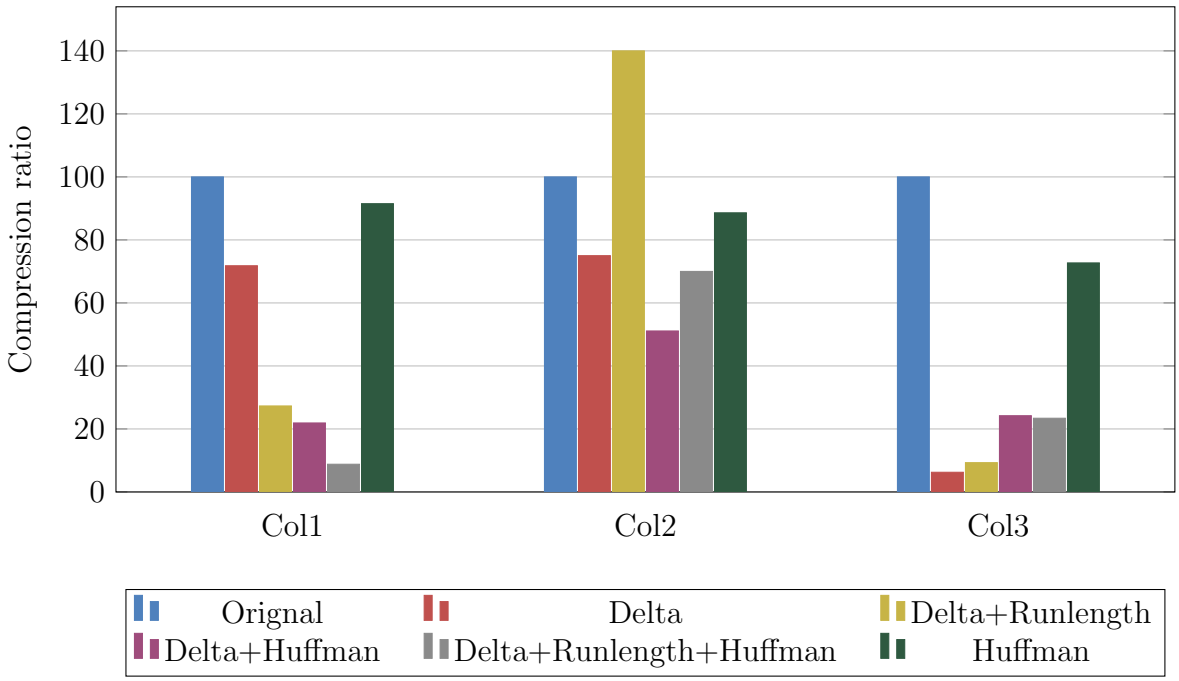
In Col1 the normalized compression ratio of the Age column in the users table of the stackoverflow database is shown. The column contains values between 6 to 100. This small amount of different values leads to the entropy of 6.57. Because the maximal delta can be ±96, the number of bits needed per value drops 8 bits with sign. This results in a compressed size of 25% compared to the uncompressed size. Through the disorder in the data-set the runlength makes the ratio worse. There are to many changes in the values, leading to many tuples representing only one value. With huffman the worst case needs 12 bits to represent the number and the additional 4 bits for the separator, resulting in a worse compression ratio.

The second column represents the ratio of Downvotes of the users table. Since the values in that column are quite high and the varying much, the deltas also keep high. This results in a worse compression ratio. However as expected a further compression with runlength is leading to an even worse ratio.

The last to column in the chart shows the ratio of the fourth column of lineitem table and the fourth column of the suppliers table in the tpch database. They are having an entropy of 2.61 and 4.64 respectively. Like in the cases before the delta compression achieves a better ratio then the combination with runlength. Since the values are small the delta compression achieves the best result of all methods.

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

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



Col1 is the compression ratio of the column Id in the votes table of the stackoverflow database with an entropy of 23.2. The Ids are a list of consecutive numbers with mostly a delta of one. 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 runlength improves the ratio, since the most deltas build up a long sequence of the same value. Since the outliers require many bits, the number of bits per number can be further reduced with huffman. The huffman compression itself gains a bad ratio, because the numbers are increasing up to 33.551.687. This means in the worst case $8 * 4 = 32$ bits for that word are needed plus additional 4 bits for the separator.

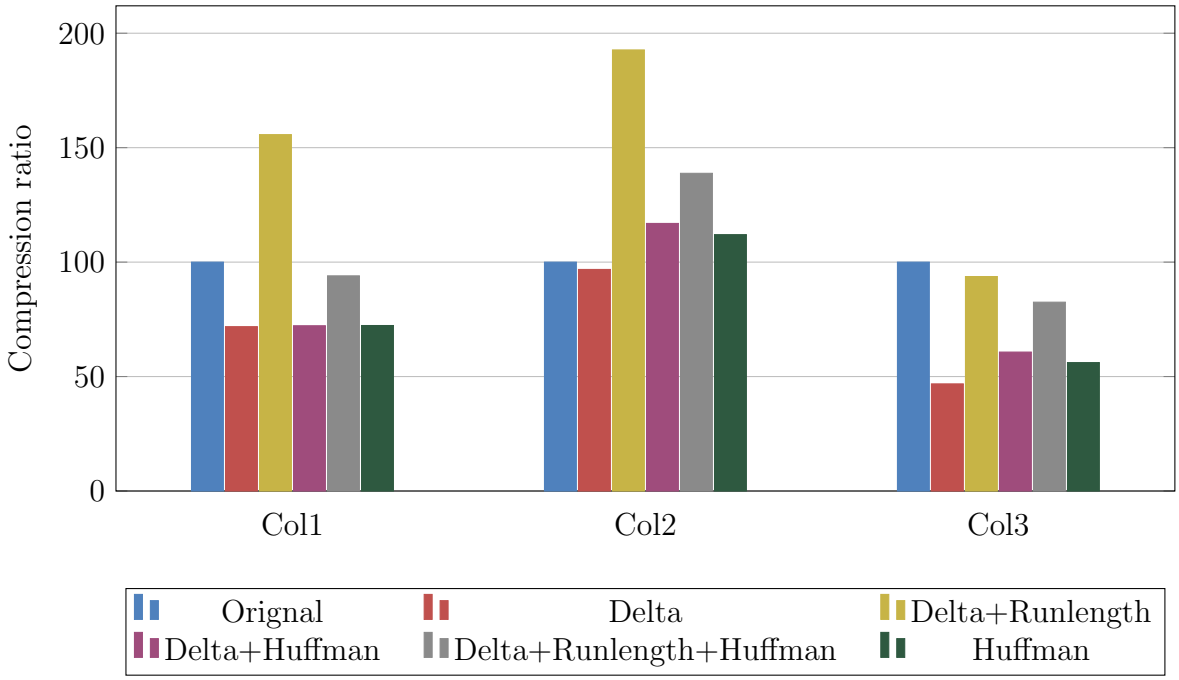
The second column represents the PostId (entropy 16.8) in PostLinks table of the stackoverflow database. As in the Col1 the numbers are in a consecutive order. But in contrast the deltas between the values are larger and varying in their value. Like before due to some large deltas the delta compression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

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

case 3 bits + separator are needed in the worst 4 bits + separator, when using huffman in combination with delta.

Compression of structured data with a high entropy can achieve good results with delta compression, if the outliers are keeps small. A further compression with runlength can make sense if the deltas are not varying more often than 50% of data-set. Also the the length of the longest sequence should be smaller than the largest token. The pure huffman compression does not work, because most numbers are too large and therefore more bits are needed for the representation than after the delta compression. The same counts for small deltas which needs less than 3 bits per value.

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



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

In Col2 the EmailHash of the User table is shown. It is having an entropy of 18.1 and contains a similar random structure as Col1 do. The difference is, that the numbers in

that data-set are larger. This results in a similar behavior of the compression methods like before. But the compression ratios are even quite worse, since the larger numbers needs more bits to be represented. Nearly all methods generating larger compressed files than the original one.

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

Overall the best compression of integers depends on the entropy of the data as well as on the structure. If the data contains small outliers and small deltas between the values. Than the delta compression achieves a quite good compression ratio. The runlength compression only makes sense if the number changes is less than 50% of the total amount of numbers in the data-set. However the length of the sequence also shouldn't be larger than the token. A further compression with huffman often results in worse ratios, since more bits are needed to represent the values. The compression of unstructured data with a large entropy is a challenging compression case. The delta compression performance therefore the best.

4.2 Float-Compression

4.3 Boolean-Compression

4.4 Date/Datetime-Compression

4.5 Text-Compression

4.6 Compression-Header

Was beinhaltet der, wie werden die Daten in die Bit array geschrieben (Delta, Runlength mit token und length), Huffman

4.7 Fail-Safe

5 Database Decompression

Wie aus Bitarray die Daten gezogen werden. Hier anhand der compressionsraten auch bestimmen welche kombination am ende den besten throughput schafft, in berücksichtigung auf den decompressionsaufwand