UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di Laurea Triennale in Tecnologie Web e Multimediali

Tesi di Laurea

COMPRESSION IN TIME SERIES DATABASES

Relatore:
Prof. NICOLA VITACOLONNA

Laureando:
UMBERTO D'OVIDIO

ANNO ACCADEMICO 2018-2019

Contents

Chapter 1

Introduction to Time Series Data

1.1 What is time series data?

Time series data can be thought of as a series of data points, measuring a certain variable over time, stored in time order. Time series datasets most commonly share three characteristics [1] :

- Data points are almost always stored as a new record
- Data typically arrive in time order
- Time is the most important feature of time series data and is a key attribute that distinguishes it from other data

For example, let us consider the S&P 500, a stock market index based on the market capitalization of the largest 500 companies in the USA. We are interested in collecting the high, low and closing price on a daily basis. The resulting data set is illustrated in Figure 1.1.

The date column is the time index, while the other columns represent measurements of some specific metrics. This is a typical example of time series data, although not every dataset containing date/time information is necessarily a time series data set.

The rule of thumb is that with time series data sets changes over time are tracked by creating new records, rather than updating existing ones. If you have a record with a timestamp, and whenever something changes in the measurement you create a new record instead of updating the previous one, then you have a time series dataset.

Date	Open	High	Low	Close	Adj Close	Volume
2018-12-31	2498.939941	2509.239990	2482.820068	2506.850098	2506.850098	3442870000
2019-01-02	2476.959961	2519.489990	2467.469971	2510.030029	2510.030029	3733160000
2019-01-03	2491.919922	2493.139893	2443.959961	2447.889893	2447.889893	3822860000
2019-01-04	2474.330078	2538.070068	2474.330078	2531.939941	2531.939941	4213410000
2019-01-07	2535.610107	2566.159912	2524.560059	2549.689941	2549.689941	4104710000
2019-01-08	2568.110107	2579.820068	2547.560059	2574.409912	2574.409912	4083030000
2019-01-09	2580.000000	2595.320068	2568.889893	2584.959961	2584.959961	4052480000
2019-01-10	2573.510010	2597.820068	2562.020020	2596.639893	2596.639893	3704500000
2019-01-11	2588.110107	2596.270020	2577.399902	2596.260010	2596.260010	3434490000

Figure 1.1: Time series values for the S&P 500 aggregated on a daily basis

1.2 What is the importance of time series data?

Time series data is becoming increasingly important in our world, as one can infer from the growing popularity of time series databases in the period 2017-2019 (Figure 1.2).

Many different businesses rely on the collection and the analysis of this type of data. DevOps monitoring, scientific measurements, financial markets are just a few business sectors heavily using time series data. In most of the applications relying on time series data, it is vital to collect values as often as possible, to create faster and more accurate monitoring systems, better models and simulations, more accurate forecasting predictions. As an example, consider a system that monitors the overall CPU usage of a distributed system. Whenever the CPU usage exceeds a certain threshold, a remediation action is triggered (for example spawning an additional node to reduce the load on the system). It is apparent that the usefulness of the system depends on how often the CPU usage values are collected (the more frequently values are collected, the faster the detection of some anomaly and the faster the remediation action can take place).

The need for finer temporal resolutions requires automatically storing larger quantities of data, up to the point where traditional relational database systems are not able to handle the scale. This is because relational database systems store data as a collection of fixed-size pages on disk. Data is usually indexed by data structures that minimize disk access, most commonly B-trees. When data sets become large to the point it is not possible to store indexes in memory anymore, inserting or updating a row might require swapping in-

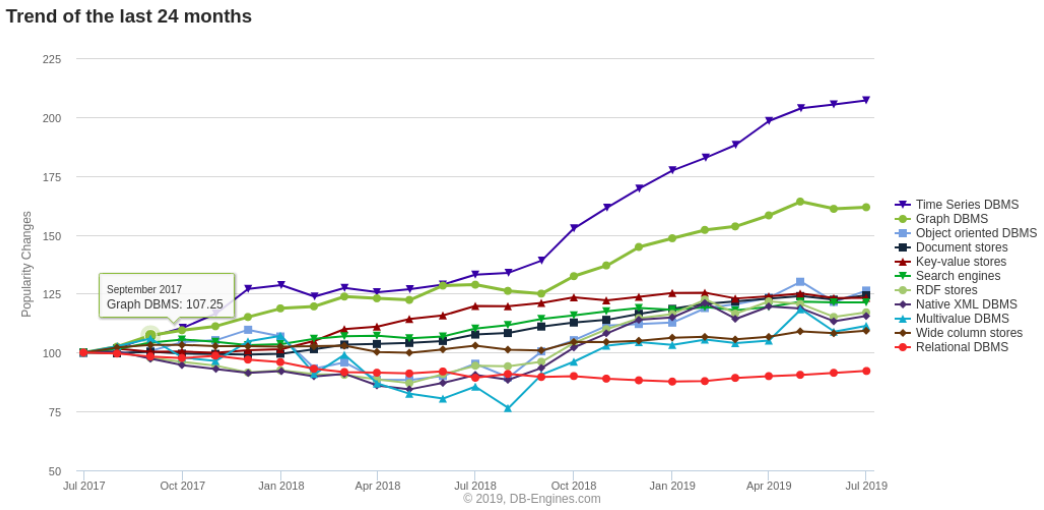

Figure 1.2: Database popularity in the period 2017-2019 according to `https://db-engines.com`, July 2019

dex pages from memory to disk, which drastically decreases performance [2]. For this reason, new databases that specifically target time series data have been created. These databases have higher ingestion rates, faster queries at scale and better data compression as compared to traditional RDBMS. In this dissertation, I will analyze the problem of data compression in time series data.

1.3 Why compressing time series?

Finding a good compression algorithm to store time series can have multiple benefits depending on the adopted technique and the use case. Intuitively, reducing the storage needs of an application allows the system to increase the temporal resolution at which the data is recorded. The storage requirements may be reduced up to the point the whole data set can fit in memory, reducing writing and reading latency by an order of magnitude or more. If the application is network bound, compressing data means reducing the amount of data to be transferred from server to client or the other way around. In some IoT applications, the compression can be done at the device level, so that less data need to be transferred over the network. This is usually more power efficient, as the energy required to compress the data is much less compared to the energy required to transmit it.

1.4 Four approaches for time series compression

We can identify four different approaches for compressing time series data.

1. **Storing aggregates**. In most applications, we are mostly interested in recent data as compared to older data. Thus, older data can be "aged out" , that is, stored in aggregates over a longer temporal interval. Consider the previous example of the S&P Index stock price. Stock prices of the current year can be aggregated over a one-hour interval, while older data can be aggregated over days, weeks, and even months. One problem with this approach is that aggregates remove fluctuations and outliers that are sometimes important when analyzing historical data.

2. **Lossless compression**. Time series data is characterized by some properties that can be exploited to obtain good compression ratios. For example, most of the time data comes at regular intervals, so it does not make sense to store the whole timestamp value for each record, but rather go with a delta encoding approach, that is storing only the difference between sequential timestamps.

3. **Lossy compression.** Many applications are concerned with the trends that can be extracted from time series data, and do not require a high level of accuracy. For this type of applications, lossy compression can be considered, which usually results in much better compression rates as compared to lossless compression.

4. **General-purpose algorithms**. Some time series databases do not offer compression out of the box. In these cases, one can consider running the databases on ZFS or other file systems that offer data compression [3]. Other benchmarks [4] have indicated that general-purpose algorithms such as LZ4, achieve good compression ratios and may be more desirable to custom solutions given their highly efficient implementations.

Chapter 2

Compression techniques for time series data

Before venturing into modern time series compression algorithms, it is useful to have a brief introduction into the basic techniques that can be used to achieve compression. In this chapter, we will discuss the principles of delta-encoding, variable-length encoding, run-length encoding, dictionary-based compression, and finally introduce Huffman encoding, an algorithm used to create variable-length codes.

2.1 Delta encoding

Delta encoding is a technique in which the difference between sequential values is stored instead of their original values. For example, suppose we have the following values:

$$V = 60, 61, 62, 60, 65$$

We can transform the values as differences plus the initial reference value:

$$V' = 60, 1, 1, -3, 5$$

Delta of delta encoding is a variation of delta encoding whereby we first transform a sequence with deltas and we then re-apply the delta encoding. This can be achieved with the following formula: $D = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$, where t_n is the n-th value in the time series sequence. This variation is useful when the deltas between values are big but almost always the same, for example, if we record data every minute the delta between the timestamps measured in seconds will always be 60, and by re-applying delta encoding we will transform the values to 0. As an example, in the first column of Table 2.1, we have 6 Unix

Timestamp	Delta	Delta of Delta
1567330058	-	-
1567330118	60	-
1567330177	59	-1
1567330238	61	2
1567330298	60	1
1567330358	60	0

Table 2.1: A comparison between delta and delta of delta compression.

time timestamps in seconds. In the second column, we have the delta between these timestamps, while in the third column we have the delta of delta. Since the timestamps were collected roughly every minute, the delta is always around 60 seconds, while the delta of the delta is always around 0. By storing the delta of deltas together with an appropriate variable length code (described in the following section), one can reduce the amount of storage needed as compared to storing the timestamps. This technique is used in Gorilla [5], a time series database created by Facebook to monitor their infrastructure, which we will analyze in chapter 3.

2.2 Variable-length encoding

Variable-length encoding refers to the technique whereby, given a dataset made up of symbols drawn by an alphabet, we replace symbols that occur more often with short codes and symbols that occur less frequently with longer codes. The result is a long string of bits with no separators between the codes of consecutive symbols. The decoder should be able to read this string unambiguously into individual codes. Such codes are called "prefix codes" in Information Theory. One example of a prefix code is Elias' Gamma Code [6]. This code, designed for positive integers, prefix each integer with a representation of its order of magnitude. Given a positive integer n, proceed as follows:

- Denote by M the length of the binary representation of n.
- Prepend $M - 1$ zeroes to n.

Table 2.2 shows some sample numbers encoded using the gamma code. How can variable-length encoding help when dealing with time series data? In many time series applications, the values we record are strongly correlated. For example, if we are recording the temperature of a certain location every

Number	Gamma Encoded Number
1	1
8	0001000
12	0001100
18	000010010

Table 2.2: Example of some Gamma encoded numbers.

minute, chances are that the value we record in a given minute is similar to its predecessor. To exploit this fact, we may use delta encoding to store only the differences between each value, and encode such value with a variable length code for integers, for example, the Rice code [7]. As a practical example, let us consider the following values:

$$V = 80, 82, 83, 81, 81$$

To compress these values, we first need to apply delta encoding:

$$V = 80, 2, 1, -2, 0$$

Now it is time to apply the Rice code. The Rice code depends on the choice of a base k and is computed in three steps. Given an integer number n, whose k-base representation we call N, proceed as follows:

- Separate the sign bit of N from the rest of the number
- Separate the k least significant bits of N. They become the least significant bits of the rice code
- Code the $j = \lfloor \frac{|n|}{2^k} \rfloor$ remaining bits as j 1's followed by a 0
- If n is non-negative, prepend 0, else prepend 1

By applying these rules with $k = 2$, we obtain the codes in Table 2.3 We can store the whole sequence in 40 bits, which is a great improvement over storing each value as a 32-bit integer. Most audio compression algorithms adopt this technique [8].

2.3 Run-length encoding

For certain types of data adjacent symbols are often correlated, and there may be runs of an identical symbol repeated that could be exploited for compression purposes. In general, the way this is exploited is by replacing a long

Initial number	Number after compression
80	0|11111111111111111110|00
2	0|0|10
1	0|0|01
-2	1|0|10
0	0|0|00

Table 2.3: On the first column, the original time series values. On the second column, the binary representation of these numbers after applying delta and rice encoding.

run with a pair (length, symbol). For example, $(1, 10), (0, 5), (1, 10)$ is the short form the following sequence of values: 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1. In many applications where changes in data are rare this leads to good compression ratios; on the contrary, if the data changes frequently using RLE results in expansion. In the following chapter, we will talk about Sprintz [9], an algorithm designed to compress sensor-generated time series data, which uses, among other techniques, run-length encoding.

2.3.1 Dictionary-based compression

In text data, we can observe that words are often repeated. We can exploit this fact by building a dictionary of words we encounter in a text and storing a reference to a dictionary entry instead of storing the words again. The same idea can be extended to non-text data, for example, we consider a sequence of numbers as an entry in our dictionary. Every time the sequence is repeated, we can store a reference to it. The entire field of dictionary-based compression was initiated with the algorithm published in a seminal paper by Ziv and Lempel in 1977, commonly referred to as LZ77 [10]. LZ77 is based on the concept of a sliding window. The window is divided into a search buffer and a look-ahead buffer. The search buffer contains the dictionary and the data that has been recently processed. The input that has not been read and encoded yet resides in the look-ahead buffer. The algorithm can be described as follows. Let $S[0...j-1]$ be the search buffer, $L[0...k-1]$ be the look-ahead buffer (starting at position j) and $W(0...j+k-1)$ be the entire window. Let $C(x)$ denote the default code for symbol x. Every symbol will be represented by a pointer structured like the following $< off, len, C(x) >$, where off is the distance from the current symbol to the position in the search buffer and len is the length of the matched string.

1. let $w[p...q]$, $p < j$ be the longest string that has a match beginning at

$L[0]$

2. if no match encode $< 0, 0, C(L[0]) >$
3. else encode $< p, q - p + 1, C(L[q - p + 1])$
4. advance search and look-ahead buffers by $len + 1$ symbols, where len is the length of the matched string

As a practical example, let us try to encode the following sequence:

$$\text{Seq} = \text{c a b r a c a d a b r a r r a r r a d}$$

For this example we will use $j = 7$ and $k = 6$. The sequence encoding is described in Table 2.4.

Search buffer	**Look-ahead buffer**	**Longest string matching $L[0...k]$ for some k**	**Encoded symbol**
cabrace	dabrar	/	$< 0, 0, d >$
abracad	abrarr	abra	$< 0, 4, r >$
adabrar	rarrad	rarra	$< 4, 5, d >$

Table 2.4: Example of dictionary-based encoding using a sliding window.

The encoded sequence will look like this: cabraca$< 0, 0, d >< 0, 4, r >< 4, 5, d >$. Over the years, many different variations of this algorithm have been proposed, but the basic principles are to be found in every general-purpose compression algorithm.

2.4 Huffman encoding

Huffman encoding is a lossless compression algorithm that produces a variable-length code, whereby each symbol is associated with a given prefix which length is proportional to the number of occurrences of the symbol. The algorithm uses a binary tree of nodes. Each node can be a leaf node or an internal node and contains a value (a character) and a frequency. To build up the tree, apply the following steps:

1. create a leaf node for each character and add them to a priority queue, where the priority is the inverse of the frequency in which they occur
2. while there is more than one node in the queue

   (a) remove the two nodes of highest priority (lowest frequency from the queue)

   (b) add a new internal node with the two nodes as children and with a frequency equals to the sum of the frequency of the two nodes. Add this node to the priority queue

3. the remaining node is the root node and the tree is complete

The leaf of the tree contains all the characters we wanted to encode. The build the prefix code for a given character, we traverse the tree until the node containing the character is found. Every left turn is encoded as 0 while the right turn is encoded as 1. As a practical example, consider the characters A, B, C, D with the associated frequency 3, 5, 7, 2. After having executed step 1, we are entering the loop of step 2. We remove the node A and D since they have the lowest frequency and we add a new node (let us call it E) to the tree with frequency 5 having A as a left child and D as a right child. We also add this node to the priority queue. The next nodes to be removed from the priority queue are B and E. We create a new node F, with frequency 10, that has B as a left child and E as a right child. We add F to the priority queue. We have two nodes left in the priority queue, C and F. We remove them, and we create a new node G, with frequency 17 that has C as a left child and F as right child. We then add this node to the priority queue. At this point, the loop ends as our tree is already completed. The resulting codes are shown in Table 2.5, while the Huffman tree is depicted in Figure 2.1.

Character	**Encoding**
A	110
B	111
C	0
D	10

Table 2.5: Resulting codes for the characters A, B, C, D with frequency 3, 5, 7, 2.

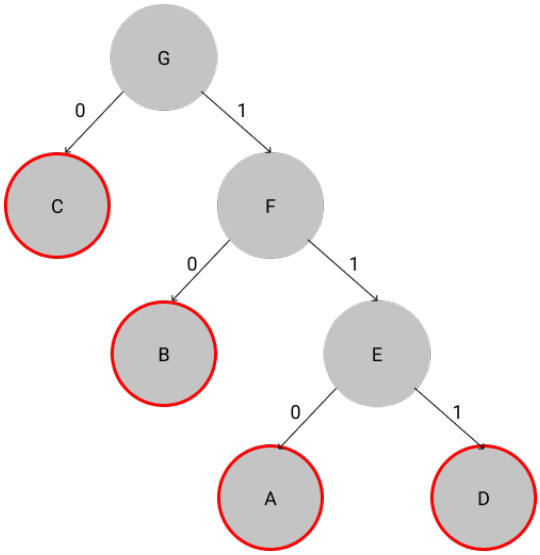

Figure 2.1: Huffman tree for characters A, B, C, D with frequency 3, 5, 7, 2.

Chapter 3

Approaches to Time Series compression

3.1 Storing aggregates

Storing aggregates is probably the most common approach to deal with large time series. The idea is to reduce the temporal resolution of data by storing aggregates of values within some predefined time buckets. Once values have been aggregated to a lower temporal resolution, they can be deleted, provided that the initial temporal resolution is not needed anymore. As a practical example, let us consider an application in which we record the time delays accumulated by trains. The following series represent the daily delay of a specific train on a certain day in minutes.

Day 1	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**
20	20	10	50	40	40	30

Table 3.1: Daily delay of a specific train of a certain day in minutes.

After each week passes, we want to aggregate the previous week's daily data. With reference to our example above, we can delete the daily values and replace them by their sum (210 minutes). By also storing the count, we can compute the average daily during that week (210 / 7 = 30 minutes). This technique, often called downsampling, is so simple and effective that most time series databases offer it out of the box. For example, OpenTSDB, an open-source time series database, offers the possibility to query downsampled data [11]. This is useful when one wants to plot data keeping the density at a level that can be easily understandable by users. Admittedly, this feature does not hold any savings in terms of storage, as it is performed only at query

time, but it is still interesting for network bound applications. Since version 2.4 OpenTSDB offers the possibility to store the result of the downsampler [12]. In this way, when querying for a long time span, there is no need to wait for the downsampled data to be computed. InfluxDB, another popular open-source database, offers the possibility to downsample old data using continuous queries and retention policy [13]. Continuous queries are queries that are run automatically and periodically within a database. Retention policy defines for how long InfluxDB keeps the data. By using these concepts together, one can use a continuous query to aggregate old data into a lower temporal resolution and delete the original data automatically by setting an appropriate retention policy.

3.2 Lossless compression

Lossless compression is a method of data compression that allows to reconstruct the original data from the compressed data [14]. When applied to time series, it is used when we are interested in retaining the original data but we want to reduce storage space. In this section, we will present two different algorithms that have been devised to solve different problems. The first algorithm was created at Facebook to face the huge amount of data gathered by monitoring their systems. This algorithm has allowed them to store all data in memory, thus improving considerably reading and writing performance. The second algorithm we present, called Sprintz, was created to limit the energy consumption of IoT devices. By sending compressed data, the network interface use is reduced, thus obtaining considerable power savings.

3.2.1 Gorilla

Gorilla is an in-memory time series database used by Facebook for monitoring the health of their systems. It was initially presented in a 2015 paper, and it is now available as open-source software under the name of Beringei [15]. Gorilla schema is a simple tuple composed of three values: the key of the time series stored as a string, the timestamp of the measurement stored as an integer, and the value stored as a double. The compression is applied both to the timestamp and the actual double values. For the timestamp the delta-of-delta technique is used (which we discussed in chapter 2), while for the double values a special kind of XOR compression is applied.

Timestamp compression

Timestamp compression is achieved by storing the delta of deltas (see previous chapter) together with a variable-length code. The algorithm can be described as follows:

1. in the block header, we store the initial timestamp $t-1$ which is aligned to a two-hour window. The first timestamp t_0 is stored as a 14-bit delta from $t-1$.
2. For the subsequent timestamp t_n:
   (a) calculate the delta of delta $D = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$
   (b) if delta is 0, store '0'
   (c) if delta is between [-64, 64], store '10' following the 7-bit value of the delta
   (d) if delta is between [-256, 256], store '110' following with the 9-bit value of the delta
   (e) if delta is between [-2048, 2048], store '1110' following the 12-bit value of the delta
   (f) otherwise, store '1111' followed by the 32-bit value

This type of compression applied to Facebook experimental data was able to achieve a 96% compression ratio. Figure 3.1 shows the result of the timestamp compression achieved by Gorilla with Facebook's data. It is interesting to note that 96% of the time timestamps were compressed in one bit.

Compressing values

Gorilla restricts the value element in its tuple to a double floating-point type. To compress the values, XOR compression is used together with a variable-length code. Since values that are close together do not differ by a large amount, the sign, exponent and few bits of the mantissa are usually the same. Gorilla exploits this by computing the XOR of the current and previous values. We XOR'd values are then encoded using the following encoding scheme:

1. the first value is stored with no compression
2. if the XOR with the previous value is 0, then store '0'
3. when XOR is non-zero, calculate the number of leading and trailing zeros in the XOR, store bit '1' followed by either a) or b):

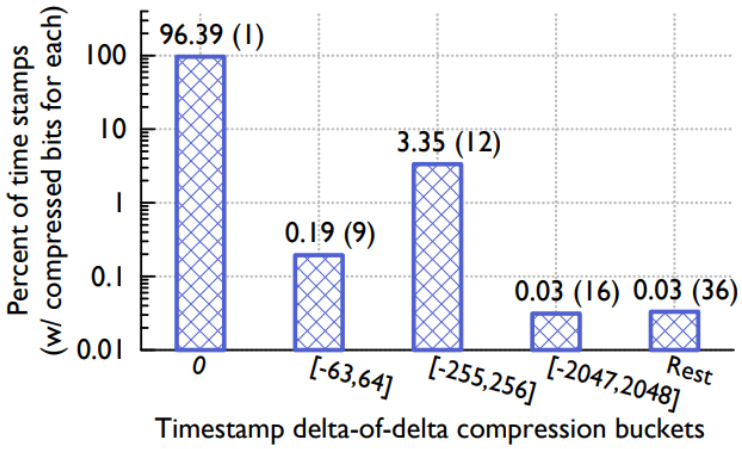

Figure 3.1: Distribution of timestamp compression across different ranged buckets. Taken from a sample of 440,000 real timestamps in Gorilla [5]

(a) (control bit '0') If the block of meaningful bits falls within the block of previous meaningful bits, i.e., there are at least as many leading zeros and as many trailing zeros as with the previous value, use that information for the block position and just store the meaningful XORed value

(b) (control bit '1') Store the length of the number of leading zeros in the next 5 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally, store the meaningful bits of the XORed value

We show an example of this coding in Table 3.2.

Using their data set, the authors of Gorilla noted that 51% of the time subsequent values do not change at all so they can be stored with one bit (case 2), while 30% of the time they have as much leading and trailing zeroes as the previous XOR'd value (case 3a), while the remaining 19% are stored with the '11' prefix (case 3b), with an average size of 36.9 bits due to the 13 bits extra overhead.

3.2.2 Sprintz

Sprintz is a compression algorithm that is specifically tailored for Internet of Things (IoT) devices [9]. These devices have low power consumption and computational limits. Sprintz reduces the amount of data sent over the network

Decimal	Double representation	XOR with previous	Encoded value
20.5	0x4034800000000000	-	0b010000000011010 01000000000000000 00000000000000000 000000000000000
18	0x4032000000000000	0x0006800000000000	0b110001100000010 1000100
21.5	0x4035800000000000	0x0007800000000000	0b101001110
21	0x4035000000000000	0x0000800000000000	0b101000
21.25	0x4035400000000000	0x0000C00000000000	0b101100
21.25	0x4035400000000000	0x0000000000000000	0b0

Table 3.2: An example of Gorilla's value compression.

while still being efficient enough to be run on low powered devices. The design requirements of Sprintz were the following:

- **Small block size**. IoT devices usually do not have much memory, therefore it is not possible to buffer large quantities of data. Moreover using large buffers introduces high latency which is not favorable for many real-time applications.

- **High decompression speed**. Decompression speed is paramount for all those applications which are read-heavy. Compression speed, on the other hand, must only be fast enough to keep up with the ingestion rate.

- **Lossless**. Instead of assuming that some level of downsampling is appropriate for all data, the approach used by Sprintz is to keep the original data quality and leave to the developer the choice to downsample data in the preprocessing stage.

Sprintz is also targeted to multivariate time series. In a multivariate time series, each record consists of multiple values. Each value is associated with a dimension. This contrasts with univariate time series, whereby there is only one dimension.

Sprintz components

Sprintz can be described as a bit packing-based predictive coder [9]. It consists of four components:

- An online forecaster, that predict the next value in the time series. Sprintz saves the error between the predicted value and the real value. This value is supposed to be 0 most of the time

- A bit packer, that packs the error as a payload and adds sufficient information to the header to invert the bit packing

- Run-length encoding: Sprintz stores the number of 0 blocks instead of an empty payload

- Entropy coding: Sprintz uses Huffman coding for the header and the payload

Encoding

The basic algorithm is outlined in the following pseudo-code listing:

In line 3-7 the forecaster is used to predict the value, get the error of the forecaster, and update the forecaster with the current parameters. Then in line 8-11, the maximum number of bits needed for representing the max value is computed for each column. If all the dimensions had a 0 error, new blocks are read until an error is found. Run-length encoding is then used to encode the 0 errors (line 17-19). The header is written as zeros repeated for the number of dimensions. The payload consists of the number of zero blocks found. Header and payload are then further compressed using Huffman code. If non-zeroes errors were found, then the header will include a header containing the bit width of each dimension. The values are then packed in the payload. Header and payload are then encoded using Huffman encoding.

Decoding

The decoding is simpler than the encoding. The first step is to decode the Huffman-bitstream into a header and a payload. Once decoded, the header is examined. If it contains all 0, it means that the payload was run-length encoded. The number of run-length encoded blocks is read, and each value can be reconstructed by predicting its value with the forecaster. If the header does not contain all zeros, then after predicting the value we also need to add the error value, which is present in the payload. The value will be the predicted value plus the error value. The forecaster is then updated with the current parameters.

Algorithm 1 Sprintz encoding algorithm [9]

1: **procedure** ENCODEBLOCK($\{x_1, \ldots, x_b\}$, forecaster)
2: Let buff be a temporary buffer
3: **for** $i \leftarrow 1, \ldots, B$ **do** ▷ for each sample
4: $\hat{x_i} \leftarrow$ forecaster.predict(x_{i-1})
5: $err_i \leftarrow x_i - \hat{x_i}$
6: forecaster.train(x_{i-1}, x_i, err_i)
7: **end for**
8: **for** $j \leftarrow 1, \ldots, D$ **do** ▷ for each column
9: $nbits_j \leftarrow max_i\{\text{requiredNumBits}(err_{ij})\}$
10: $packed_j \leftarrow$ bitPack($\{err_{1j}, \ldots, err_{Bj}\}, nbits_j$)
11: **end for**
12: // Run-length encode if all errors are zero
13: **if** $nbits_j == 0$, $1 \leq j \leq D$ **then**
14: **repeat** ▷ Scan until end of run
15: Read in another block and run lines 3-11
16: **until** $\exists_j[nbits_j \neq 0]$
17: Write D 0s as headers into buff
18: Write number of all-zero blocks as payload into buff
19: Output huffmanCode(buff)
20: **end if**
21: Write $nbits_j$, $j = 1, \ldots, D$ as headers into buff
22: Write $packed_j$, $j = 1, \ldots, D$ as payload into buff
23: Output huffmanCode(buff)
24: **end procedure**

Algorithm 2 Sprintz decoding algorithm [9]

```
1:  procedure DECODEBLOCK(bytes, B, D, forecaster)
2:      nbits, payload ← huffmanDecode(bytes, B, D)
3:      if nbits_j == 0 ∀j then
4:          numblocks ← readRunLength
5:          for i ← 1, ..., (B × numblocks) do
6:              x_i ← forecaster.predict(x_{i−1})
7:              Output x_i
8:          end for
9:      else
10:         for i ← 1, ..., B do
11:             x_i ← forecaster.predict(x_{i−1})
12:             err_i ← unpackErrorVector(i, nbits, payload)
13:             x_i ← err_j + x̂_i
14:             Output x_i
15:             forecaster.train(x_{i−1}, x_i, err_i)
16:         end for
17:     end if
18: end procedure
```

The forecaster

Sprintz uses a novel forecaster algorithm called FIRE (Fast Integer REgression). It is slightly slower than delta encoding but yields a better compression ratio most of the time [9]. Each value is thought of as a linear combination of a fixed amount of previous values plus some error. The details of the forecaster are outside the scope of this thesis, but the interested reader can find more in the original paper [9].

Bit packing

Explaining bit packing will be easier by using a practical example. Suppose we want to encode the following time series:

X	**Y**	**Z**
5	2	104
10	2	103
15	2	102
20	2	101

We first need to compute the delta. Supposing that the data point imme-

diately preceding this series was time = -3, value 1 = 2, value 2 = 104, then the delta will look like the following table:

dX	dY	dZ
8	0	0
5	0	-1
5	0	-1
5	0	-1

The next step is to apply the Zigzag encoding. Zigzag encoding is used to transform all values in nonnegative by representing each integer by twice its absolute value, or twice its absolute value minus one for negative integers so that positive integers are encoded as even numbers and negative integers as odd numbers, effectively using the least significant bit to represent the sign.

dX	dY	dZ
16	0	0
10	0	1
10	0	1
10	0	1

The next step is to find the bit width for each column. To do so, for each value we need to compute the current bit width minus the number of leading zeros. For each column, we take the maximum of this value, which we will call w. For example, assuming we are 8-bit integers, 16 is represented in binary as 00010000, so we will have $8 - 3 = 5$ bits necessary to represent this number. 10 is represented as 00001010 and requires only 4 bits to be represented. Therefore we take the max of the two values, 5. The reason why Zigzag encoding was used is that computing the number of leading zeros can be done in a single assembly instruction (on systems that support it, such instruction is usually called `clz`). In the following table we list the maximum number of bits required to represent each number in each column.

dX	dY	dZ
5	0	1

The final step will be to write the header and bit pack the deltas in the payload. The header will consist of D unsigned integers, one for each column, storing the bit widths. Each integer in the header is stored in $\log_2 w$ bits. In the payload, values are stored contiguously. In the following table, we use different colors for different columns. Orange will be used for the X column, green for the Y column and blue for the Z column.

Header	Align	Payload
101 000 001	0000000	10000 01010 01010 01010 0111

When there are many columns, the bits for each sample are stored contiguously, adding up to 7 bits padding when the sample does not span a multiple of a byte. The reason why this is not done when there are few columns is that having a 7-bit padding can reduce drastically the compression ratio, and in the case of univariate data it leads to expansion.

Entropy coding

For entropy coding, a Huffman coder is used. This step is optional, but the authors have observed increased compression ratios using it. Huffman coding is applied after bit packing for two reasons:

1. Reducing the input to the Huffman coding also reduces its latency. This is important as Huffman coding is slower than the rest of Sprintz

2. A better compression ratio is achieved as compared to the case in which Huffman coding is applied first.

3.3 Lossy compression algorithms

This family of algorithms uses inexact approximations to represent a data series. In contrast with lossless compression, these algorithms do not allow to compress data and keep the full resolution of it. In this section, we will present some algorithms that use this approach. A feature of the algorithms that we present is that they allow for quality guarantees to be defined, that is, users can define how much the compressed values can differ from the original value. In the following sections, we will analyze Piecewise Constant Approximation, Piecewise Linear Approximation, and ModelarDB, a time series database that makes use of multiple algorithms to optimize the compression ratio based on the user-defined quality guarantees.

3.3.1 Piecewise Constant Approximation

Piecewise constant approximation represents a time series S as a sequence of K segments [16][17]:

$$PCA(S^n) = \langle (c_1, e_1), (c_2, e_2), \ldots, (c_k, e_k)) \rangle$$

where e_k is the end-point of a segment and c_k is a constant value for time in $[e_{k-1}+1, e_k]$ or for time in $[1, e_1]$ for the first segment. If each segment corresponds to many samples of the series, then high compression ratios can be achieved. In [17], a simple algorithm name Poor Man's Compression - Midrange (PMC-MR) was proposed to construct a PCA representation of a series with $O(1)$ space requirements and $O(n)$ time complexity, where n is the number of samples in a series. This algorithm does also take into consideration quality guarantees, that is, users can decide the maximum acceptable quality loss the algorithm will produce. Algorithm 3 lists the procedure taken from [17].

Algorithm 3 Poor Man's Compression Midrange [17]

```
1: procedure PCR(series, tolerance)
2:     PCA(series) ← ⟨⟩
3:     n ← 1
4:     m ← series[n]
5:     M ← series[n]
6:     while series.hasMoreSamples() do
7:         if max{M, series[n]} - min{m, series[n]} > 2 x tolerance then
8:             append(\frac{M+m}{2}, n − 1) to PCA(series)
9:             m ← series[n]
10:            M ← series[n]
11:        else
12:            m ← min{m, series[n]}
13:            M ← max{M, series[n]}
14:        end if
15:        n ← n + 1
16:    end while
17:    append(\frac{M+m}{2}, n − 1) to PCA(series)
18:    return PCA(series)
19: end procedure
```

The procedure uses n to represent the endpoint of the segment, and represents the minimum and maximum values found in the segment not yet pushed to the PCA with m and M. In each cycle, we evaluate whether the range we are considering is no more than twice the quality boundary. If that is the case, we update m and N. Once the current range exceeds this quality boundary, we push $\frac{m+M}{2}$ up until $n-1$ and we reset m and M to the current value (which was not included in the current segment). As a practical example, suppose we

have the following series:

$$S = 22, 24, 31, 32, 33, 37$$

If our quality guarantee is 3 (that is, the absolute value of the difference between the compressed value and the original value should never exceed 3), the series will be compressed to the following PCA:

$$PCA = \langle (23, 2), (34, 6) \rangle$$

In Figure 3.2 we have a visual representation of how the series has changed.

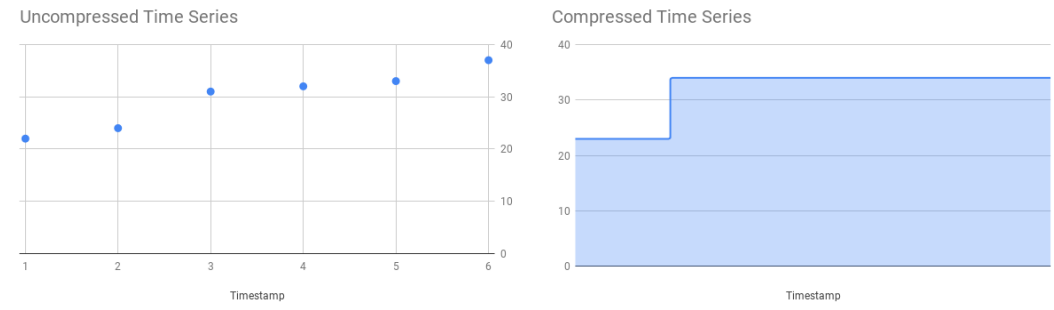

Figure 3.2: A time series before and after applying the PMC-MR algorithm.

3.3.2 Piecewise Linear Approximation

Piecewise Linear Approximation (PLA) is the natural evolution of PCA. Instead of approximating multiple values with a single constant value, we approximate a time series S with k straight lines. Algorithm 4 is one popular algorithm that produces a PLA.

This algorithm resembles PMC-MR, in that it grows a segment until it can fit the user-defined max error. The algorithm keeps track of two variables: anchor, which define the start index of a potential segment, and i, which defines the end of the segment. It appends the generated segments to PLA, a variable referencing a list. In line 6-7 it checks whether the segment from anchor to anchor + i is able to represents the original values within the tolerance value defined. When this condition is satisfied, the index i is increased. If that is not the case, then the segment from anchor to anchor + $i - 1$ is appended to the PLA(series) and the anchor variable is updated. The approximation can be improved by using other variations. Most of them are batch algorithms [18][19] [20], but some online alternatives have been proposed as well [21].

Algorithm 4 Online Piecewise Linear Approximation [17]

```
1: procedure PLA(series, tolerance)
2:     PLA(series) ← ⟨⟩
3:     anchor ← 1
4:     while not finished segmenting time series do
5:         i ← 2
6:         while calculateError(series[anchor:anchor +i]) < tolerance do
7:             anchor ← anchor + 1
8:         end while
9:         append createSegment(series[anchor:anchor+i]) to PLA(series)
10:    end while
11:    return PLA(series)
12: end procedure
```

3.3.3 ModelarDB

ModelarDB is a time series database management system that was designed to store and query massive amounts of high-quality sensor data ingested in real-time from many sensors [22]. To tackle these problems, model-based compression was used. Model-based compression is a lossy compression technique that stores a representation that can recreate data within a known error bound. For example, PLA is a type of model-based compression, using linear equations to represent data. ModelarDB is model agnostic, this means that it uses multiple models depending on the one with the best fit. It also supports user-defined models. Users can extend the system with their own compression models.

Model-Agnostic Compression Algorithm

ModelarDB uses a model agnostic compression algorithm so that users can extend the set of compression models. Data is stored as segments, where each segment stores multiple data points and can use a different model. In order to satisfy latency constraints, two types of segments are used, namely a temporary segment (ST) and a finalized segment (SF). The STs are emitted based on a user-defined latency in terms of data points not yet emitted to the stream, while SFs are emitted when a new data point cannot be represented by the set of models used. We have an example of how the algorithm works in Figure 3.3. For this example, the latency is three data points, and a single linear model is used. At t_3 we have three points, and an ST is emitted since the number of represented points equals the latency. As the model M might

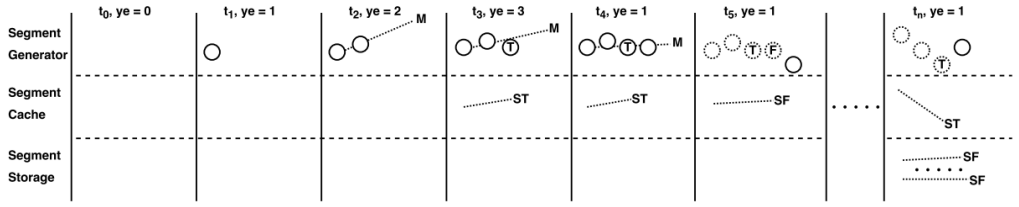

Figure 3.3: A depiction of the ModelarDB compression algorithm, taken from [22]

be able to fit more data points, the points are kept in a buffer and the next data point is added at t_4. At t_5, M cannot fit all data points within the user-specified error bound, so it emits then an SF with the 4 fittable values and a new segment is created from datapoint 5. After a user-defined bulk write size is reached, the segments are flushed to disk. The pseudo-code is listed in Algorithm 5. After initializing variables, we iterate through all the values of the time series (line 12). In case a value was lost, we flush the current buffer (lines 14-16). The data point is then added to the buffer, and we set previous to the current data point. The data point is then added to the model. If the model can still fit the data point, the yetEmitted variable is increased. If this variable has reached the user-defined latency, then a temporary segment is emitted and the variable is set to 0. If the model cannot accept another data point without exceeding the user-defined error, then a new model is initialized with the current buffer (lines 25-27). When the list of models becomes empty, an SF containing the model with the highest compression ratio is emitted in line 29. We then initialize the first model with the remaining points in the buffer (lines 30-31). In line 32 we update yetEmitted to reflect the decrease in the number of emitted points. Finally, we flush the remaining data points in the buffer in line 35.

3.4 General-purpose algorithms

In this section, we will analyze some general-purpose algorithms that have been used in time series databases. We start with LZ4, an extremely fast compression algorithm inspired by LZ77. We continue with Deflate, an algorithm that mixes dictionary compression and entropy compression. We will finish with ZStandard, an open-source Deflate alternative.

Algorithm 5 Online model-agnostic compression algorithm [22]

```
1: procedure MODELARDB(series, tolerance)
2:     Let series be the time series of data points
3:     Let models be the list of models to select from
4:     Let tolerance be the user defined error bound
5:     Let limit be the limit on the length of each segment
6:     Let latency be the latency in non emitted data points
7:     Let interval be the sampling interval of the time series
8:     model ← models.head()
9:     buffer ← createList()
10:    yetEmitted ← 0
11:    previous ← nil
12:    while series.hasMoreSamples() do
13:        dataPoint = series.getNextDataPoint()
14:        if timeDifference(previous, dataPoint) > interval then
15:            flushBuffer(buffer)
16:        end if
17:        appendDataPointToBuffer(dataPoint, buffer)
18:        previous ← dataPoint
19:        if appendDataPointToBuffer(
            dataPoint, buffer, tolerance, limit) then
20:            yetEmitted ← yetEmitted + 1
21:            if yetEmitted = latency then
22:                emitTemporarySegment(model, buffer)
23:                yetEmitted ← 0
24:            end if
25:        else if models.hasMoreModels() then
26:            model ← models.getNextModel()
27:            initialize(model, buffer)
28:        else
29:            emitFinalizedSegment(models, buffer)
30:            model ← models.head()
31:            initialize(model, buffer)
32:            yetEmitted ← min(yetEmitted, buffer.length())
33:        end if
34:    end while
35:    flushBuffer(buffer)
36: end procedure
```

3.4.1 LZ4

LZ4 is a lossless compression algorithm focused on compression and decompression speed based on LZ77 [10][23]. As LZ77, LZ4 uses a sliding windows consisting of a search buffer and a look-ahead buffer. Repetitive data is replaced with an index to the search buffer. An LZ4 compressed block is composed of sequences. A sequence consists of a token, literal length, offset and matched length, as shown in the following table.

Token	**Literals Length**	**Literals**	**Offset**	**Match Length**
1 byte	$0 - n$ bytes	$0 - \|Literals\|$ bytes	2 bytes	$0 - n$ bytes

The token is a 1-byte value separated in two 4 bits fields. The four most significant bits, which can range from 0 to 15, represent the length of the literals that will follow. Since the literals can be of any length, bytes are dynamically added if they cannot fit in the 4 bits. For example, suppose we want to represent the value 470. Then the first part of the token will be set to 15, and we would need to read a byte from the literal length part. This byte will be 255. When the byte is 255, we need to read a new byte. This byte will be 200. When the byte is not 255, we know that this was the last byte in the literal length part. So in total we will have 15 + 255 + 200 = 470 bits. This mechanism scales linearly with the length of literals, but this usually is not a problem since most of the time the length of the literal is small. The four least significant bits of the token indicates the match length. As with the length of the literal, this number has no limitations and can be expanded by adding new bytes to the match length section of the block. Following the token and the length of the optional literal, we have the literals part of the block. It is possible that this part is skipped (this happens when the word is entirely matched in the dictionary). The offset is composed of two bytes, and it represents the offset from the position where the match begins. Let us make a practical example. Suppose that during the LZ4 compression we have the following binary string in our search buffer

$$S = 00001111000011110101$$

We want to compress the following string by refering the search buffer:

$$S' = 001111000011110101011111$$

We notice that the first 18 bits are identical to the last 18 bits in the search buffer. Thus we will only need to include the last 4 bits in the literals section of the block. We already know how the token byte will look like: 0100|1111.

The first 4 bits indicate that we have a 4 bits literal length. The last 4 bits indicate that we have a match bigger than 15 bits, therefore we need to check for additional bytes in the match length part of the block. Since the literals length is < 15, we do not need any literals length additional bytes. Next we can encode the literals as 1111. We then need to encode the offset, which is 18. So far we have the following bits: 01001111111100000000000010010. The last part is to include an additional match length byte, to obtain 18 bits of matched length. This last byte will encode 3 (we must sum the 15 bits of the token to obtain 18). The final block encoding S' thus looks like the following:

Token	**Literals Length**	**Literals**	**Offset**	**Match Length**
01001111	/	1111	00000000 00010010	00000011

3.4.2 Deflate

Deflate uses both LZ77 and Huffman coding to achieve compression [24]. A compressed data set is composed of blocks. Each block can use a different "mode" of compression. There are three available modes:

1. No compression. Useful when data has already been compressed
2. Compression, with LZ77 and Huffman coding in sequence. The trees used are the ones in the specification.
3. Compression, with LZ77 and Huffman coding. The compressor creates the trees and store them with the data.

Each block consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part, and a compressed data part. The compressed data parts consist of literals and pointers to the duplicated strings, represented as $< length, distance >$ pairs. Each type of value (literals, distances and lengths) in the compressed data is represented using Huffman code, using one code tree for literals and lengths and a separate one for distances. When using compression mode 2, the Huffman codes are not explicitly represented in the data, as they can be generated by the decompressor. This is because mode 2 uses a Huffman tree with two further restrictions as compared to the standard Huffman tree:

- elements with shorter codes are placed left to element with longer codes
- elements with the same length are ordered lexicographically

The reason why these two restrictions are placed upon trees, is that with these restrictions for a given set of elements and their respective length there is only one possible tree. When using mode 3, the Huffman codes for the two alphabets must be included. Deflate has many implementations that differ in some minor details. The most popular are zip, gzip, and zlib.

3.4.3 ZStandard

ZStandard was built to be a replacement for deflate for faster and more effective compression [25]. It uses Finite State Entropy (FSE) [26] and it is optimized for modern CPUs. Finite State Entropy can be seen as a replacement for Huffman encoding: it allows for the same compression ratio while improving on the speed [26]. The details of FSE are outside the scope of this thesis, but the interested reader can find more information in [26] [27]. ZStandard is a versatile compression algorithm, suitable for usage in widely different circumstances. It is faster both in compression and decompression speed as compared to zlib, while achieving similar compression ratios [25]. ZStandard offers also a specific training mod, which can be used to tune the algorithm for a selected type of data. The result of the training is stored in a dictionary which will be loaded before compression and decompression. This can be an interesting use case for compressing JSON data returned by web services [25].

Chapter 4

Empirical evaluation of compression algorithms

In this chapter, we compare different compression algorithms in terms of compression ratio using both real and generated time series data. The compression algorithms that we have chosen are Gorilla, LZ4, Deflate and Zstandard. By doing this we want to evaluate whether a compression algorithms specifically targeted to time series is better than general-purpose algorithms.

4.1 Datasets

For these benchmarks, we have used three different data sets. The first dataset was generated from the Time Series Benchmark Suite, a command-line tool written by TimescaleDB [28] that makes easy to generate realistic time series data to benchmark different time series databases. The second dataset was created by using Dynatrace. Dynatrace is a Software Intelligence platform with a strong emphasis on application performance monitoring. The datasets consist of metrics gathered from different hosts running a wide range of applications under different loads. The third dataset was provided by the New York Taxi and Limousine Commission (TLC), and it consists of records representing taxi trips, including information such as duration of the trip, amount charged, tip amount [29].

4.1.1 Generating devops data

As we have mentioned above, we have used the TimeSeries Benchmark Suite to generate fake DevOps data. This utility allows to specify which use case to simulate data for, which time interval between each data points to

use, the starting timestamp and the ending timestamp, how many hosts to simulate. We have decided to generate data points every minute, for a single host and for 5 different time intervals: 2 hours, 4 hours, 8 hours, 16 hours and 48 hours. For each of these intervals, we generate 50 different time series.

4.1.2 Collecting Dynatrace Data

Dynatrace provides a REST API to retrieve metrics it collects [30]. We have used this API to retrieve data of several hosts which were monitored in one of the demo environment Dynatrace uses for testing purposes. The metrics we have chosen to collect are the following:

- Host CPU Usage: Percentage of overall CPU usage
- Host CPU System: Percentage of CPU time used by the kernel
- Host CPU User: Percentage of CPU time used by userspace processes
- Host CPU IO Wait: Percentage of CPU time spent waiting for input/output operations
- Host Memory Used: Percentage of memory used
- Host Memory Usage: Memory usage in bytes
- Host Disk Read Time: Disk read time in milliseconds
- Host Disk Write Time: Disk write time in milliseconds

4.1.3 Taxi Data

The taxi data set was provided by the New York Taxi and Limousine Commission. For each month of the year, the TLC provide a data set for the two different taxi companies running in New York (green and yellow) and for limousines. We have decided to take into account only yellow taxi data for one month. Moreover, to make in-memory compression feasible, we have dropped the last 3 million records from the dataset. Additionally, all the columns containing string data were removed since they are not handled by the Gorilla algorithm.

4.2 Compressing data

The metric which we are interested in these benchmarks is the compression ratio, while we do not evaluate compression speed as this is dependent on the specifics algorithms implementations. We have created a Java program that reads data from the input stream and returns the compression ratio achieved by the algorithm selected. We decided to use Java because we could easily find implementations for all the algorithms listed above. The program requires to specify the format of the dataset provided in the standard input and the algorithm to use for compression. For example, if we want to benchmark gorilla with the dynatrace dataset, we would execute the following command:

```
$ java -jar
TimeseriesCompressionBenchmarks-all.jar dynatrace gorilla
```

Gorilla compression does not work on raw byte data. It needs instead pairs of values and timestamps. For this reason, we parse each time series file to obtain a TimeSeries object. This object contains a key and a list of value-timestamp pairs. We pass this TimeSeries object to a compressor, which is an interface responsible for returning a compressed version of the time series. In the case of Gorilla compression, we have implemented this interface by iterating through all the data points, pushing those to the Gorilla compression library, and returning the compressed version of the time series, which is a long array in the case of the Gorilla Compression library we are using. With the other compression algorithms, we could serialize the TimeSeries object to a byte array, which was then compressed to a new byte array. The way we compute the compression ratio is by dividing the number of bytes representing the serialized time series object, divided by the number of bytes of the serialized compressed time series. The code is available on github [31]. For gorilla compression, we have used an open source java implementation of the algorithm [32]. Deflate, on the other hand, is already present in the the package *java.util.zip*, using the zlib library under the hood [33]. For LZ4 and ZStandard we have used open source implementations [34][35].

4.3 Results

4.3.1 Generated data

Figure 4.1 illustrates the result of applying different compression algorithms with generated data. It clearly shows how Gorilla (mean = 13.35, standard deviation = 1.60), is outperforming the other lossless compression algorithms. It is also interesting to notice how ZStandard (mean = 3.58, standard deviation

= 0.32) and Deflate (mean = 3.22, standard deviation = 0.24) performs similarly, while LZ4 (mean = 2.36, standard deviation = 0.14) has consistently the worst compression ratio, as one might have predicted given that it was engineered to optimize compression speed rather than compression ratio.

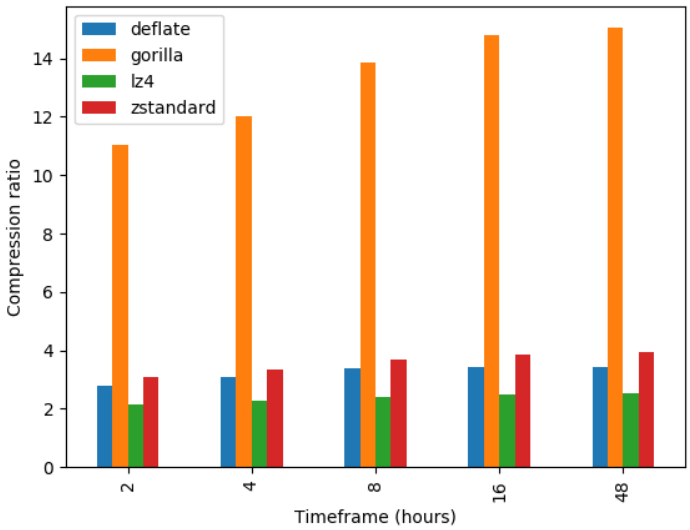

Figure 4.1: Generated data compressed with Gorilla, Deflate, ZStandard, LZ4. Higher values indicates higher compression ratios.

4.3.2 Dynatrace Data

Results of the different compression algorithms applied to real-word series data are shown in Figure 4.2. The results are similar to what we have seen for generated data, with Gorilla (mean = 3.40, standard deviation = 0.10) outperforming all the other general purposes algorithm across all metrics. As we also saw in the previous benchmark, deflate (mean = 2.16, standard deviation = 0.34) and zstandard (mean = 2.15, standard deviation = 0.42) have a similar performance, which is significantly better than LZ4 (mean = 1.77, standard deviation = 0.19). It is interesting to notice that although Gorilla provides the best compression here, it is much lower as compared to the compression achieved with generated data. By having a closer look at both sources of data, we hypothesized that this is because the generated data contains repeated consecutive values much more often as compared to real-world data. Gorilla is optimized to reduce the storage need for this particular case, and this alone

would explain the big difference. Indeed a closer inspection reveals that the percentage of repeated values in the case of generated data is 41%, while in the case of real-world data is 0.76%. This happens because most values were generated using a random walk distribution which results in a high correlation between the values. Moreover, most of the generated values are integers. One interesting thing we have also noted is that the time series benchmark tool is that it tends to create more values when the timeframe is larger. When it comes to data gathered by Dynatrace, all the values are represented as floating-point values. Although the delta between the values tend to be small most of the time, the chances of getting the same value in a sequence are much smaller with high-resolution real data.

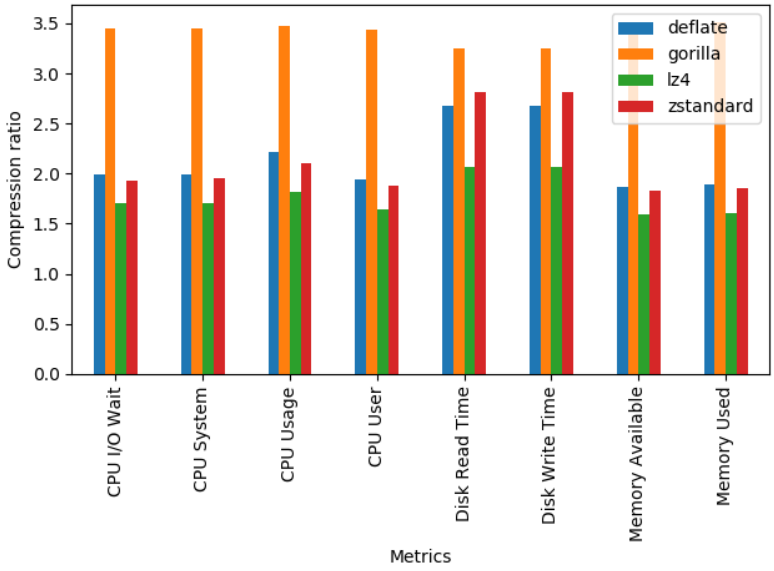

Figure 4.2: Real world data collected by Dynatrace compressed with Gorilla, Deflate, ZStandard, LZ4. Higher values indicates higher compression ratios.

4.3.3 Taxi Data

For taxi data, we have decided to transform the data in a column format. Since each record contains 15 columns, this means that we have transformed data in 15 time series, ordered by pick up time. Each time series data point represent a specific value for a specific taxi trip. The time interval between each data point is not regular as in the previous data sets, for this reason

Gorilla delta of delta timestamp compression is likely to not yield good compression ratios. Moreover since there is no correlation between taxi trips, XOR compression of float will not be effective. Having a look at the results in Figure 4.3, we can see that indeed Gorilla compression does not perform well as in the other benchmarks. The best compression algorithms in this case are deflate (achieving a compression ratio of 6.78) and zstandard (achieving a compression ratio of 6.70). Gorilla performs slightly better than LZ4 (4.80 vs 4.11).

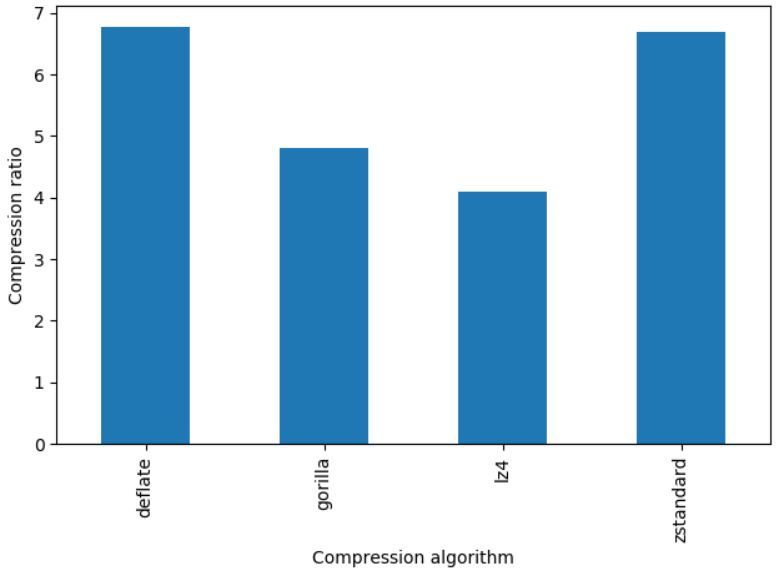

Figure 4.3: Taxi Data compressed with Gorilla, Deflate, ZStandard, LZ4. Higher values indicates higher compression ratios.

4.3.4 Discussion

From our limited results, it is hard to say which compression algorithm is the best for time series data. Although Gorilla seems to allow for greater compression ratios for DevOps data, this is not the case when time series data has no correlation, and when the interval between data points is irregular. Deflate and ZStandard offers good compression ratios and they have the advantage to have many stable and optimized implementations. LZ4 performed consistently worse than the other algorithms in each of the benchmarks, but might be chosen for those applications where compression and decompression speed are important.

Conclusions

This thesis aimed to identify the principal approaches and algorithms for time series data compression. We have identified four main approaches that are used in real-world applications and databases. We then focused on lossless compression algorithms by creating benchmarks to evaluate how Gorilla, an algorithm specifically tailored for infrastructure monitoring time series data, performs against state of the art general purposes algorithms. These benchmark clearly illustrates how Gorilla can outperform the general purposes algorithm, but it also shows how this is highly dependent on the correlation between values in the time series. Based on these results, we suggest to first analyze which kind of data one is dealing with before choosing a compression algorithm. Our benchmarks showed that the best results are obtained with time series data sets whereby the same values are repeated in neighboring data points.

We should note that Gorilla compression deals only with univariate time series, and this can limit its usefulness. In the monitoring, we usually have different metrics collected for the same entity at regular intervals. As an example, we could measure disk space, CPU usage and memory usage for and store them as a multivariate time series. Gorilla would require us to split this time series into three different univariate time series, thus causing the same timestamps to be repeated three times. Adding some sort of bit packing, as Sprintz does, would allow us to use Gorilla also for multivariate time series. One limit of our benchmarks is that we did not consider compression and compression speed This might be interesting especially when dealing with database queries and inserts. On the one hand, data compression increases the information density of transferred data, thus improving disk bandwidth. On the other hand, the decompression overhead might outweigh the potential benefits of compression. One recent work has found that query times were improved considerably after using compression [36].

In conclusion, this thesis provides an overview of the problem of time series compression and offers some insights into the compression performance of Gorilla and other general purposes lossless algorithms. Future research should

expand on these findings by including other compression algorithms and by taking account compression and decompression speed.

Bibliography

[1] A. Kulkarni, *What the heck is time-series data (and why do I need a time-series database)?*, 2018. [Online]. Available: https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/ (visited on 09/21/2019).

[2] M. Freedman, *Time-series data: Why (and how) to use a relational database instead of NoSQL*, 2017. [Online]. Available: https://www.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/ (visited on 09/21/2019).

[3] *Timeseries storage and data compression*, 2016. [Online]. Available: https://docs.timescale.com/v1.3/faq#compression (visited on 09/21/2019).

[4] J. Danjou, *Timeseries storage and data compression*, 2018. [Online]. Available: https://julien.danjou.info/gnocchi-carbonara-timeseries-compression/.

[5] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database", *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.

[6] P. Elias, "Universal codeword sets and representations of the integers", *IEEE transactions on information theory*, vol. 21, no. 2, pp. 194–203, 1975.

[7] R. F. Rice, J. P. L. U. I. S. Division, U. S. N. Aeronautics, and S. Administration, *Some practical universal noiseless coding techniques*. National Aeronautics And Space Administration, Jet Propulsion Laboratory, California Institute Of Technology, 1979.

[8] W. Contributors, *Golomb coding*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Golomb_coding (visited on 09/21/2019).

[9] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time Series Compression for the Internet of Things", *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, p. 93, 2018.

[10] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression", *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[11] *Downsample — OpenTSDB 3.0 documentation*, 2019. [Online]. Available: `http://opentsdb.net/docs/3x/build/html/user_guide/querynodes/downsample.html` (visited on 09/21/2019).

[12] *Rollup And Pre-Aggregates — OpenTSDB 2.3 documentation*, 2019. [Online]. Available: `http://opentsdb.net/docs/build/html/user_guide/rollups.html` (visited on 09/21/2019).

[13] *Downsampling and data retention — InfluxData Documentation*, 2016. [Online]. Available: `https://docs.influxdata.com/influxdb/v1.7/guides/downsampling_and_retention/` (visited on 09/21/2019).

[14] W. Contributors, *Data compression*, 2019. [Online]. Available: `https://en.wikipedia.org/wiki/Data_compression` (visited on 09/21/2019).

[15] facebookarchive, *facebookarchive/beringei*, 2018. [Online]. Available: `https://github.com/facebookarchive/beringei` (visited on 09/21/2019).

[16] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Locally adaptive dimensionality reduction for indexing large time series databases", in *ACM Sigmod Record*, ACM, vol. 30, 2001, pp. 151–162.

[17] I. Lazaridis and S. Mehrotra, "Capturing sensor-generated time series with quality guarantees", in *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, IEEE, 2003, pp. 429–440.

[18] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature", *Cartographica: the international journal for geographic information and geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[19] S. Park, D. Lee, and W. W. Chu, "Fast retrieval of similar subsequences in long sequence databases", in *Proceedings 1999 Workshop on Knowledge and Data Engineering Exchange (KDEX'99)(Cat. No. PR00453)*, IEEE, 1999, pp. 60–67.

[20] E. J. Keogh and M. J. Pazzani, "An Enhanced Representation of Time Series Which Allows Fast and Accurate Classification, Clustering and Relevance Feedback.", in *Kdd*, vol. 98, 1998, pp. 239–243.

[21] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series", in *Proceedings 2001 IEEE International Conference on Data Mining*, IEEE, 2001, pp. 289–296.

[22] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "ModelarDB: modular model-based time series management with spark and cassandra", *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1688–1701, 2018.

[23] lz4, *lz4/lz4*, 2019. [Online]. Available: `https://github.com/lz4/lz4` (visited on 09/21/2019).

[24] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3", no. 1951, 1996, `http://www.rfc-editor.org/rfc/rfc1951.txt`.

[25] C. Y., *Smaller and faster data compression with Zstandard - Facebook Engineering*, 2018. [Online]. Available: `https://engineering.fb.com/core-data/smaller-and-faster-data-compression-with-zstandard/` (visited on 09/21/2019).

[26] ——, *Finite State Entropy - A new breed of entropy coder*, 2013. [Online]. Available: `http://fastcompression.blogspot.com/2013/12/finite-state-entropy-new-breed-of.html` (visited on 09/21/2019).

[27] ——, *FSE decoding : how it works*, 2014. [Online]. Available: `http://fastcompression.blogspot.com/2014/01/fse-decoding-how-it-works.html` (visited on 09/21/2019).

[28] timescale, *timescale/tsbs*, Sep. 2019. [Online]. Available: `https://github.com/timescale/tsbs` (visited on 10/10/2019).

[29] *TLC Dataset*, 2018. [Online]. Available: `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`.

[30] *Metrics API - GET all metrics — Dynatrace Help*, 2013. [Online]. Available: `https://www.dynatrace.com/support/help/extend-dynatrace/dynatrace-api/environment-api/metric/get-all-metrics/`.

[31] dovidio, *dovidio/ts-compression-thesis*, Oct. 2019. [Online]. Available: `https://github.com/dovidio/ts-compression-thesis/tree/master/code/TimeseriesCompressionBenchmarks` (visited on 10/10/2019).

[32] burmanm, *burmanm/gorilla-tsc*, Mar. 2018. [Online]. Available: `https://github.com/burmanm/gorilla-tsc` (visited on 10/10/2019).

[33] *Deflater (Java Platform SE 8)*, Mar. 2019. [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/util/zip/Deflater.html` (visited on 10/10/2019).

[34] lz4, *lz4/lz4-java*, Sep. 2019. [Online]. Available: `https://github.com/lz4/lz4-java` (visited on 10/10/2019).

[35] luben, *luben/zstd-jni*, 2015. [Online]. Available: `https://github.com/luben/zstd-jni`.

[36] M. Burman *et al.*, "Implementing compression on distributed time series database", 2017.