Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Olav Kaada

Continuous lossless compression of streams of high-frequency multivariate financial market data

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

NTNU
Norwegian University of Science and Technology

Olav Kaada

Continuous lossless compression of streams of high-frequency multivariate financial market data

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Data is being produced at increasingly high frequencies. These magnitudes often leave current storage systems inadequate, calling for more efficient methods for handling streams of data. High-resolution sensor data enables monitoring and analysis of complex systems and processes. More efficient representations of such streams enable in-memory storage of data samples, opening a field of possibilities for real-time systems.

This thesis takes a grasp at lossless compression of streams of high-frequency multivariate time series, financial market data in particular. Market data is unique in its unpredictability. Compressing such data while maintaining its information entirely yields exciting challenges.

We argue that the compression of multivariate streams is essentially the same as compressing multiple univariate streams with a shared time dimension. By leveraging techniques from state-of-the-art compression schemes in this domain, we perform a novel study of compression of streams of high-frequency multivariate data.

Using techniques such as comparative encoding, we can continuously compress multivariate samples of high randomness losslessly, increasing the number of recent samples storable in-memory by a factor of 4.4 on average. Compression speeds one order of magnitude higher than the frequency of the most actively traded assets, and one-pass decompression with blocks of less than 550 kB, indicates that such techniques may be suitable for in-memory storage in data-intense real-time systems.

Sammendrag

Data blir produsert med stadig høyere frekvenser. Disse datamengdene fører i økende grad til at nåværende lagrings systemer blir utilstrekkelige. Det trengs derfor mer effektive metoder for å håndtere datastrømmer. Høyoppløselig sensordata muliggjør overvåking og analyse av komplekse systemer og prosesser. Mer effektive representasjoner av slike strømmer muliggjør lagring av datapunkter i minne, og åpner et hav av muligheter for sanntidssystemer.

Denne oppgaven ser på tapsfri komprimering av strømmer av høyfrekvente multivariate tidsserier, spesielt finansiell markedsdata. Markedsdata er unike i deres uforutsigbarhet. Tapsløs komprimering av slik data har interessante utfordringer.

Vi argumenterer for at komprimering av multivariate strømmer i hovedsak er det samme som å komprimere flere univariate strømmer med en delt tidsdimensjon. Ved å utnytte teknikker fra moderne komprimeringsmetoder i dette domenet, utfører vi et studie om komprimering av strømmer av høyfrekvente multivariate data.

Ved bruk av teknikker som komparativ koding, kan vi komprimere multivariate datapunkter med høy tilfeldighet kontinuerlig og tapsløst, og dermed øke antallet nylige datapunkter som kan lagres i minne med en faktor på 4,4 i gjennomsnitt. Komprimeringshastigheter på én størrelsesorden høyere enn frekvensen til de mest aktivt handlede aktiva, og et-pass dekomprimering med blokker på mindre enn 550 kB, indikerer at slike teknikker kan være egnet for lagring av datapunkter i minne i dataintensive sanntidssystemer.

Preface

I want to thank Svein Erik Bratsberg for his helpful guidance and support in supervising this work.

Huge thanks to my friends for the support during the process of writing this thesis. Thanks to Kristian Flatheim Jensen for questioning methods and incorrect information. Thanks to Håvard Pettersson for providing tools and assistance helpful in its writing, including thorough feedback on late drafts.

The code produced for this thesis will be made available on the author's GitHub (https://github.com/kaada/). Feel free to reach out.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The rapidly increasing rate at which data is generated yields both new possibilities and challenges [1]. High-resolution sensor data, often represented as time series, enables monitoring and analysis of complex systems and processes. However, the increasing magnitudes of data often leave current storage systems inadequate [2]–[5].

Recent work has shown that compression can increase the throughput and efficiency of real-time systems [2]. More efficient representations of sensor data increase the number of samples that can be stored in-memory while reducing bandwidth requirements. Fewer cache misses and more efficient access to main memory can dramatically increase query performance [6]. The improvements enable higher data loads, paving the way for, e.g., previously unheard-of real-time monitoring tools.

This thesis considers high-frequency multivariate financial market data: high-density time series with multiple variables connected to each timestamp. Market data is used, for instance, to model and monitor the order book (buy/sell interest) of an asset [7]. The sheer volume of data generated from such financial systems makes the optimization of their storage crucial [1].

High-frequency data enables accurate and precise models of complex systems; every piece of information is therefore valuable. It is essential that each data point is available fast and with no loss of information, recent data points in particular.

Existing schemes for lossless compression of multivariate data operate on blocks of samples [8], [9]. By not supporting continuous compression of streams of data, fewer *recent* data points can be held in memory, significantly reducing the availability of these critical data points [2].

1.1 Goals and contributions

Based on previous contributions in compression of streams of data, we study, implement, and benchmark an algorithm for lossless compression of streams of high-frequency multivariate data. The goal of the research is to provide an example of how existing methods for compression of univariate streams can be combined to compress multivariate streams. The contributions are expected to identify challenges and solutions to a problem whose results may increase performance in a broad domain of services, particularly caching and storage of real-time data in high-throughput services.

We focus on the compression of financial market data: sensor-like data with significantly high frequencies and unpredictability. These properties enable a general overview of the problem of compression of high-frequency multivariate data.

We investigate the following research questions:

- **RQ1**: How can existing techniques for compression of time series be leveraged to create a compression scheme for continuous lossless compression of multivariate data?
- **RQ2**: What structures and properties exist in financial market data enabling compression?
- **Main RQ**: How can we losslessly compress streams of high-frequency multivariate financial market data to enable more efficient in-memory storage for real-time systems?

1.2 Outline

- In Chapter 2, we present background material concerning financial market data, and techniques and algorithms for compression of streams.
- In Chapter 3, we take a closer look at the data, and go into detail on the specifics of our implementation of a compression algorithm for streams of high-frequency multivariate financial market data.
- In Chapter 4, we analyze and discuss the benchmark results.
- In Chapter 5, we present conclusions and finish with suggestions for future work.

Chapter 2

Background

This chapter introduces the necessary knowledge regarding high-frequency financial market data and compression. In Chapter 2.1, we discuss the background, structure, and properties of time series and market data in particular.

With this new-found understanding regarding time series, Chapter 2.2 takes a look at why compression is beneficial, exposing features important for further examination of methods for continuous compression of streams (Chapter 2.3).

2.1 High-frequency financial market data[1]

High-frequency financial market data is a type of time series. We first look at how time series are generated, including their characteristics. Using this knowledge, we study the domain of high-frequency data and how it differs from data of lower frequencies. We then consider high-frequency market data specifically.

2.1.1 Time series

A time series is a collection of data points sourced over a period of time. Even though the notion of measurements associated with a specific time is common in everyday life, the granularity and preciseness have recently increased drastically to satisfy the requirements of the information age. Weather information can be sampled almost continuously, stock prices are captured

[1] Adapted from the author's specialization project from fall 2019 at NTNU.

Table 2.1: Example of a univariate time series, sampling temperature with seconds granularity. Time in HH:MM:SS.

Time	Temp (°C)
08:00:00	20.0
08:00:01	20.1
08:00:02	20.0

in ticks—one logical unit of information [7]—and sensors from small and inexpensive IoT devices can sample their environments multiple times per second.

Format and properties

A data point in a time series consists of a set of values associated with a particular time. Each data point is called a *sample*, where the number of non-time values—D—denote the sample's dimensionality. If $D = 1$, the sample is called *univariate*, if $D > 1$ it is *multivariate* [9].

A time series is a sequence of N samples, each with dimensionality D. Each sample is atomic, and has no relationship to the other samples, except for shared dimensionality and structure. Although the delta of two adjacent samples normally decreases with increased granularity—particularly when sampling natural environments—it is more of a practical property than a feature.

Data is also assumed to be immutable; updates are rarely needed. Considering that samples often arrive in chronological order, adding new samples—for instance, in row-major order (Chapter 2.2.2)—is done in an append-only fashion.

Even though the data is usually sampled with a fixed frequency (e.g., once every second, as in Table 2.1), data can also be sampled irregularly given the occurrence of an event; a change of information (Table 2.2). The latter may, therefore, have an arbitrary number of different data points at a particular timestamp (depending on the granularity of the time).

2.1.2 High-frequency data

High-frequency data consists of values sampled at high rates. The significant granularity allows for capturing even the smallest changes in delicate and

Table 2.2: Example of Level 1 market data for Apple Inc., January 3rd, 2018. Only selected columns are shown. Time is on the format HHMMSSxxxxxxxxx, where the 9 x's represent a nine digit nanosecond value. Bid and offer sizes are the maximum number of shares available at the given price [12].

Time	Bid Price	Bid Size	Offer Price	Offer Size
80000189974662	172.26	2	172.55	2
80000190231252	172.26	2	172.55	1
80000190619305	172.26	2	172.63	1
80001307295190	172.52	3	172.63	1
80001388058920	172.52	3	172.62	1

sensitive systems. The higher the number of independent samples, the higher the degree of freedom. The increased granularity allows for more precise models; for instance, statistical models to examine the probability of extreme events [1].

High-frequency market data is used—amongst other things—as a source to monitor, identify, and trade on temporary market inefficiencies created by competing interests [10]. Being able to leverage vast amounts of data in a flash is paramount.

In contradiction to traditional long-term buy and hold strategies, high-frequency trading strategies require shorter evaluation periods to establish the strategy's credibility [10]. Increasing the amount of new data available at short notice is expected to increase both the speed and preciseness of such systems.

2.1.3 Financial market data

This thesis considers a stream of financial market data, more specifically, *Level 1* market data. The data reports, for instance, the current best prices for securities in a market. Even this simple form of market data is highly complex. We focus on a subset of its fields, representing the real-time highest bid and lowest ask for an asset, including the quantities—or volumes—available at those prices [11].

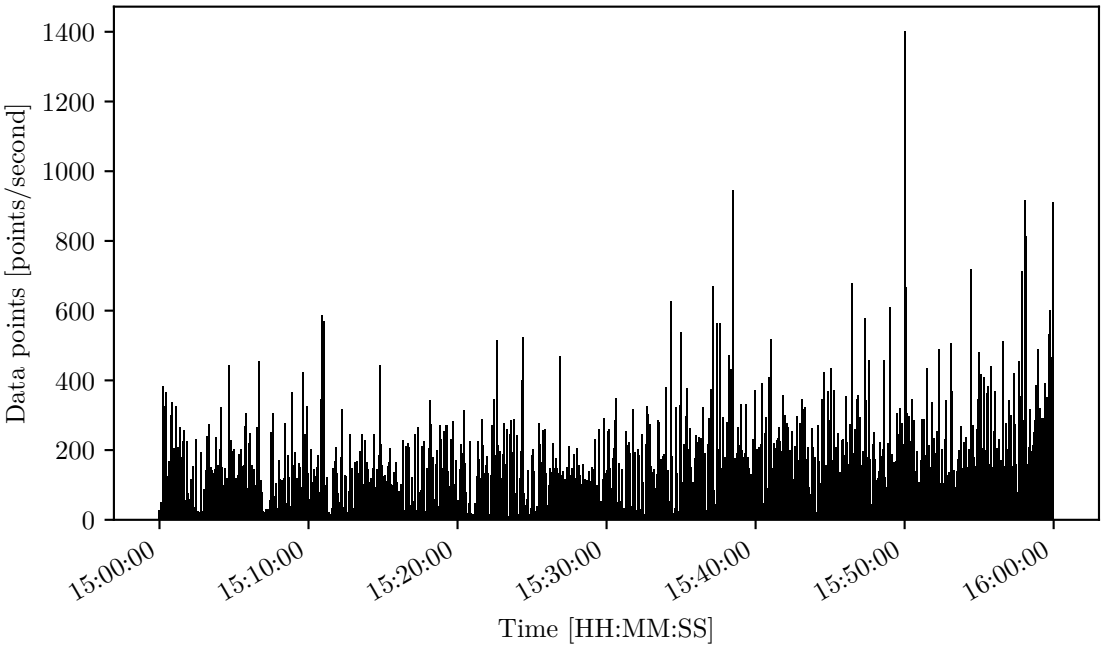


Figure 2.1: Distribution of Level 1 market data for Apple Inc., measured by data points per second from 15:00:00 to 16:00:00 on January 3rd, 2018 from the NYSE. The stock is the 11th most actively traded on this date according to the sample data set provided by the exchange [13]. $N = 273793$.

Format and properties

Level 1 market data are time series, where price and volume changes in a market are captured, as exemplified in Table 2.2. Whenever a new bid, offer, or trade is made—changing the pricing or volumes—a new data point is sampled. As a data point is stored given a change (tick) in the market, the sampling naturally occurs irregularly in time. The samples are captured with nanosecond precision, with a frequency depending on the asset, often a significant number per second [1]. The sampling frequency of *Apple Inc.* over one hour can be seen in Figure 2.1.

Table 2.2 shows Level 1 market data for Apple Inc. Only the aforementioned fields are shown. These fields contain critical values used, for instance, to calculate and monitor the bid-ask spread of an asset, or for measuring liquidity and market activity.

Financial markets are unique in their unpredictability. According to the Efficient Market Hypothesis, an asset's price reflects all available information, and price changes are therefore random, as stated by the Random Walk Hypothesis [7].

Normalizing monetary values

Prices in the data set are represented as floating-point values (exemplified in Table 2.2). The data type creates significant hurdles if left unconsidered. Although the price might seem correct for a single sample, a floating-point value is only an approximation of the underlying value. The approximated value can thus increasingly differ from the actual value when inaccuracies are combined, for instance, when values are aggregated [14].

The precision of a floating-point number generally increases with the number of bits used for its decimal value [15]. Higher precision, therefore, imply a large number of bits, unfortunate for a compression scheme.

These hurdles can be mitigated by representing monetary values in their lowest atomic form, e.g., in cents for USD. Thus, by storing these values as fixed-points—i.e., as integers—instead of floating-points, we can obtain accurate representations of prices, while using fewer bits.

2.2 Compression[1]

Compression of data is done to reduce the memory footprint in preparation for processing, transmission, or storage. The performance benefits a system gets from compression, comes from a variety of factors: lower miss-rate as a larger amount of keys are stored hot, lower bandwidth requirements in multiple transmission stages, and saved energy due to less energy-consuming memory needed for the same amount of data [16].

This section describes the different classes of compression schemes, their primary traits, and the importance of each in its design.

2.2.1 Classes

Compression algorithms are categorized as either lossy or lossless, with other customizable traits to make them perform well in the specified domain.

Lossy

Lossy compression algorithms approximate a sequence of data. The compressed representation errors in relation to the real representation, to save space. With time series, lossy compression can be done, for instance, by approximating the data as a sequence of polynomials [9].

Lossless

Lossless compression considers a more space-efficient representation of the data and conserves the information entirely. Lossless algorithms are used, e.g., in the field of audio compression [9]. Lossless compression of time series can be done, for instance, by storing the delta of subsequent values, instead of the actual values. This is known as delta encoding (Chapter 2.3.1).

Although universal lossless compression algorithms exist—many based on the widely used Lempel-Ziv-Welch (LZW) algorithm [8]—special-purpose compression algorithms yield the best result given that the data distribution and requirements of the system are known a priori [6]. General-purpose algorithms also operate on complete data sets, making them infeasible for processing streams of data in real-time [2].

2.2.2 Technical properties

Aside from the compression rate of a scheme, with the upper limit depending on the compressibility of the data set, the important properties of a lossless compression scheme are its speed, block size, and major order.

Speed

Compression is subject to the space-time complexity trade-off. While uncompressed data is ready instantly, increasing the compression ratio to reduce the storage footprint generally increases computing time needed for compression and decompression [16]. Considering this trade-off is paramount when constructing a compression scheme for a particular domain. Some devices might restrict computational resources in order to save battery. In other devices, data ingestion can be slow due to a lower sample rate, allowing computational resources to be focused on increasing the compression rate, and thus reducing network costs [9].

When time series are stored—either in memory or on disk, for instance, in a Time Series Database (TSDB)—the system is often read-heavy. Large blocks of data could be read and fed to machine learning models or visualization tools. Good decompression speed is essential. Compression simply needs to keep up with the ingestion rate [9].

Block size

A block is the physical set of bits or bytes amassed and compressed, either continuously or when the block is full. A block is thus the unit of data usually considered when compressing or decompressing data.

The block size is the number of bits stored in each block. Greater block sizes generally mean more redundancy, making higher compression rates possible [16]. However, larger blocks require more data to be decompressed to retrieve a single entry. Some devices have limited memory resources, unable to hold significant amounts of data before transmission or storage. In environments with lower sample rates, filling a block could take a significant amount of time, causing lags in data sent from the sensing device [9]. The most practical parameter is once again hugely dependent on the domain.

Major order

The major order of a storage system, such as RAM, refers to how multidimensional data is stored [14]. If each sample is multidimensional, the samples can be stored in either row- or column-major order.

Consider a time series of N samples, each having dimensionality $D > 0$. $A_{r,c}$ is the value in column c of row r. $A_{r,1}$ is then the timestamp of sample r, followed by its D variables in $A_{r,2}, ..., A_{r,D+1}$.

Multidimensional data can either be stored, linearly, in *row-major order*:

$$A_{1,1}, ..., A_{1,D+1}, ..., A_{N,1}, ..., A_{N,D+1}$$

or in *column-major order*:

$$A_{1,1}, ..., A_{N,1}, ..., A_{1,D+1}, ..., A_{N,D+1}$$

Row-major order is preferred if multiple variables are considered when a sample is fetched; in a location device, for instance, both longitude and latitude are needed to calculate the position. It is thus beneficial that they are stored sequentially in pairs, reducing lookup speeds.

In column-major order, the columns are laid out linearly. In a device sampling temperature and humidity, the temperature values could be stored sequentially, followed by the humidity values. This storage is beneficial if queries on the data usually consider the variables in isolation.

2.3 Lossless compression of streams

This section introduces comparative techniques and general techniques popular when losslessly compressing samples from streams.

2.3.1 Comparative techniques[1]

For lossless compression of streams of numeric data, three main types of predictive encoding are prevalent: *delta encoding*, *delta-of-delta encoding*, and *predictive filtering* [9]. These methods compress streams efficiently, as they reduce the entropy of a newly observed sample based on previous observations.

Delta encoding

A value is encoded as the difference from the previous observed value; the next value is predicted to be equal to the previous one.

$$\delta_i = x_i - x_{i-1}$$

Where the first value is stored in full.

$$[x_1, \delta_2 = x_2 - x_1, ..., \delta_N = x_N - x_{N-1}]$$

A value is decompressed by computing the prefix sums.

$$x_i = x_1 + \sum_{j=2}^{i} \delta_j$$

A value that is expected to be close to its preceding value is expected to have a delta significantly smaller than its original value. Storing this delta is thus expected to significantly reduce the number of bits needed to store its information [6].

Delta-of-delta encoding

A value is encoded as the difference from the previous difference.

$$\delta_i^2 = \delta_i - \delta_{i-1} = (x_i - x_{i-1}) - (x_{i-1} - x_{i-2})$$

Where the first value is stored as a delta from a reference value [2].

$$[\delta_1, \delta_2^2 = \delta_2 - \delta_1, ..., \delta_N^2 = \delta_N - \delta_{N-1}]$$

The value is decompressed by computing (from Appendix A):

$$x_i = x_{ref} + i\delta_1 + \sum_{j=2}^{i}(i - j + 1)\delta_j^2$$

This method is particularly efficient if the values are evenly spaced; timestamps sampled at regular intervals will have a delta-of-delta of zero.

Predictive filtering

Instead of simply predicting that the next value will have a naive correlation to the previous value(s), predictive filtering creates a model based on a linear combination of a fixed number of preceding samples. When this filter is learned from the data, it is called adaptive filtering.

$$x_i = ax_{i-1} + bx_{i-2} + ... + \epsilon$$

Though computationally more expensive, calculating the linear factors (e.g., a and b), and storing only the error, ϵ, from the prediction yields a higher compression rate when the samples do not follow a random walk [9].

2.3.2 General techniques

Compression algorithms use a variety of techniques and methods to optimize for speed and size. This section introduces bit packing and zigzag encoding, both essential in storing data compactly. Advanced hardware techniques, such as SIMD [6], [17], are also increasingly being used in modern compression algorithms—for significantly increased speed—but are out of scope for this work.

Table 2.3: Example of variable-length bit packing of a 32 bit integer with the value of 17226. The green bit indicates whether the *value* is non-zero. Yellow, the number of bits in the value. Red, the actual non-zero value.

Value	32 bit binary	Bit packed
17226	00000000000000000100001101001010	10111110000110100 1010

Bit packing

Bit packing [2], [6] is a technique for storing ("packing") values together, omitting unnecessary bits in each value. For instance, given values in range $[0 - 15]$, we can pack eight values in a single 32-bit integer—and thus in a single 32-bit word—as each value only needs $log_2(16) = 4$ bits of information [9].

The technique is also suited for variable-length encoding of values. In this case, a header can be used to specify the number of bits of entropy in the variable. For instance, values of a maximum of 32 bits, need $log_2(32) = 5$ bits specifying the size of the value (Table 2.3).

Further optimization of bit packing can be done for data sets with certain properties. For instance, data sets consisting of a large number of zeros can be optimized by storing a flag bit, indicating whether the value is zero or not. Thus if the value is zero, the value is stored directly using a single bit. If the value is non-zero, a 1 is stored, followed by the header bits, followed by the bits representing the actual value.

Using a flag bit to indicate a non-zero value, and variable-length encoding, the number of bits for a value $n > 0$ can be expressed as the sum of the bits needed:

$$1 + log_2(E) + \lceil log_2(n) \rceil = b,$$

where E is the number of bits of entropy in the variable. Bit packing is thus of less benefit when $b \to E$, as the size of the encoded representation approaches the size of the original representation of the value.

Bit packing reduces the number of bits needed to represent a value. When combined with techniques such as delta and delta-of-delta encoding (Chapter 2.3.1), the algorithm has elevated efficiency, as zeros are more prevalent [17].

Listing 2.1 shows a C++ implementation for bit packing a single integer into a bit vector. The vector of each value can then be stitched together into a single bit vector representing the multivariate sample.

Unpacking a bit packed sequence is essentially the reverse process of packing [17]. If the first bit is 0, we know that the full value is zero, and we continue to the next bit representing the next value. If the value is 1, we know that the next $log_2(E) = N$ header bits represent the following number of bits used to store the actual value.

Unlike the packing algorithm, the speed of unpacking is not directly dependent on the size of the value. The actual value can be accessed immediately, given the header size, resulting in a constant upper bound on the speed. The value zero, however, has a significantly lower speed constant, as the value is stored in the first bit, omitting the need for further parsing.

Listing 2.1: Example C++ code for bit packing a 64 bit unsigned integer into a bit vector using a specified number of header bits.

```
/*
 * Returns 0 if num == 0, else
 *
 *      1             x...x         n...n
 *      ^               ^             ^
 * control bit   n header bits   sig bits
 */
std::vector<bool> bitPack(uint64_t num, size_t nHeaderBits)
{
    std::vector<bool> bits;
    bits.reserve(nHeaderBits + sizeof(num) * 8);

    if (num == 0) {
        bits.push_back(0);
    } else {
        bits.push_back(1);

        auto nSignificantBits = sizeof(num) * 8
                              - __builtin_clzl(num);

        std::bitset<6> sigBits = nSignificantBits;
        for (int8_t i = nHeaderBits; i > 0; --i)
            bits.push_back(sigBits[i]);

        std::bitset<64> bs = num;
        for (int8_t i = nSignificantBits; i > 0; --i)
            bits.push_back(bs[i]);
    }

    return bits;
}
```

Table 2.4: Example of zigzag encoding the value −1.

Value	8 bit binary	Zigzagged value
−1	11111111	00000001

Zigzag encoding

To efficiently bit pack a value, the toggled bits (1's) should be stored in the least significant positions, allowing the 0's in the most significant positions to be packed away. Negative values have information in the most significant bits when using two's complement representation [18]. These values have all their bits flipped, and one added, compared to their inverse—positive—counterparts. The number -1 is thus all 1's for any arbitrary sized integer. Zigzag encoding [9], [19] is a trivial encoding that moves that sign bit to the least significant position, and flips the rest of the bits, as shown in Table 2.4. This transformation allows bit packing to treat the integer as a non-negative (unsigned) value.

Listings 2.2 and 2.3 show how such encoding and decoding can be done for a 64 bit signed integer. Decoding is the inverse process of encoding.

Listing 2.2: Example C++ code for zigzag encoding a 64 bit integer.

```
void zzEncode64(int64_t *x)
{
    *x = ( (uint64_t) *x << 1 ) ^ -( (uint64_t) *x >> 63 );
}
```

Listing 2.3: Example C++ code for zigzag decoding a 64 bit integer.

```
void zzDecode64(int64_t *y)
{
    *y = (int64_t) ( (*y >> 1) ^ -(*y & 0x1) );
}
```

2.3.3 Related work[1]

This section takes a look at some interesting state-of-the-art systems for lossless compression of streams of time series.

Gorilla

Gorilla is an in-memory time series database developed at Facebook [2]. It functions as a write-through cache storing the most recent sensor data from Facebook's systems. Their compression scheme and extensive hardware allow for compression of 700 million univariate data points per second. The database enables Facebook to have 26 hours of sensor data in cache, greatly reducing the query times on the cached data.

The scheme uses comparative compression techniques to compress streams of univariate time series. Timestamps and values are compressed separately. Facebook found that in their system, block sizes that extended 2 hours gave diminishing returns on the compression rate.

Timestamps are compressed using delta-of-delta encoding. A reference timestamp, t_{-1}, is stored in a block header. The first timestamp, t_0, is stored in the block as the delta from the reference timestamp. Subsequent timestamps, $t_1, ..., t_n$, are stored as delta-of-delta of the previous value. Facebook found that 96% of the timestamps could be compressed to a single bit, due to samples arriving at a fixed interval.

Gorilla is limited to compression of univariate time series; only a single, double-precision floating-point value is allowed per sample. The first float is stored in full. Subsequent values are stored as the XOR of the previous value. This technique assumes that neighboring values differ only slightly, which was observed to be the case with Facebook's operational data.

The system makes two strong assumptions of the data set: timestamps are mostly evenly spaced, and subsequent values have small deltas.

Sprintz

Sprintz is a time series compression algorithm aimed at IoT devices [9]. The algorithm has low memory requirements but adds a small latency to the system. Although designed for low-energy devices, the authors argue that it can also be used for larger systems for serving and querying data. The authors found the algorithm to significantly outperform delta encoding. It

uses a forecasting algorithm that is trained online and tuned to the data distribution.

Sprintz's advantage over existing solutions stems from its strong assumptions of the characteristics of the data. The paper argues that time series from natural environments have four unusual characteristics:

- *Lack of exact repeats*: The presence of natural features such as noise makes exact repeats of sequences of bytes less common. Typical compression algorithms, such as dictionary-based methods, are thus not suitable.
- *Multiple variables*: More than one variable is often needed when sampling a natural state, e.g., longitude and latitude for location.
- *Low bandwidth*: Devices normally use limited bandwidth when sampling environments. Integers of 32—or even 16—bits are sufficient.
- *Temporal correlation*: Successive samples tend to have similar values. The sample rate is often higher than the natural phenomenon recorded.

The algorithm makes assumptions about strong correlations between consecutive values. This assumption about temporal correlation is often the case for IoT devices; for instance, when sampling the location of a stationary device. It, therefore, performs poorly when the data distribution tends to jump between discrete states.

Differential-finite-context-method (DFCM)

DFCM makes predictions by looking for matches in previous difference patterns [20]. A table lookup is performed to retrieve the difference that followed the last time a similar sequence was observed. The difference is an XOR-based delta encoding on the predicted and actual value. The algorithm is essentially the same as in Gorilla, but with predictions based on context.

Chapter 3

Lossless compression of streams of high-frequency market data

In this chapter, we describe the development and implementation of an algorithm for lossless compression and decompression of streams of high-frequency market data.

In Chapter 3.1 we discuss technical requirements of an algorithm working on high-frequency multivariate market data. We then, in Chapter 3.2, take a more in-depth look at the structure and properties of the data. This information hints at methods and techniques appropriate for an efficient compression scheme in this domain. Lastly, in Chapter 3.3, we go into detail on the implementation of this compression scheme, and benchmarks created to confirm its viability.

3.1 Technical requirements[1]

A general trend in Chapter 2 is that the efficiency of a compression scheme increases the more it is fitted to the problem. Both Gorilla and Sprintz obtain their advantage over generalized schemes by forming explicit assumptions and requirements of the system.

We, therefore, form a set of preliminary technical requirements for compressing streams of market data losslessly.

3.1.1 Speed

We know from Chapter 2.2.2 that the speed requirements of the domain are clear: while the compression speed must keep up with the injection rate, decompression speed is of more benefit after that threshold is reached.

A complex stream of financial market data may consist of samples from multiple distinct stocks or assets. Luckily, although samples from different assets may be correlated, the samples are independent. The compression of each asset's samples can thus be distributed over multiple processor cores (or nodes). The *hard* speed requirement of the compression algorithm is, therefore, essentially the maximum frequency of the most actively traded asset.

From Figure 2.1, we see that the activity for an individual (actively traded) asset is several orders of magnitude lower than the obtained speeds of known efficient methods of compression of streams [6], [20]. These methods are, however, for streams of a single variable. Testing has to be done to measure how well such techniques perform when adapted for streams of multiple variables.

For a system handling the complete stream of samples, the compression speed requirement is the peak frequency of the full set of samples. The requirement is, therefore, dependent on the number of assets in question. For all US equity markets combined, the frequency peaks at roughly 5 million messages per second [5].

3.1.2 Block size

Pelkonen et al. [2] states that the increase in block size comes with diminishing returns. This statement is logically grounded, as the main benefit of having larger block sizes is to reduce the relative impact of the initial uncompressed values, taking up a significant number of bits. This impact decreases as the block size increases. Hence, blocks have an optimal size where an increase in size does not yield a noteworthy increase in the compression rate.

We want to keep the block size as small as possible, while still having the benefits of compression. Empirical data on the compression rates of different block sizes will show where this threshold is for a specific scheme and data set. For Facebook's Gorilla system, the optimal block size was two hours of data (Chapter 2.3.3). Considering that market data have variable frequencies, it can be expected that block size is better measured in the number of ticks.

3.1.3 Major order

Level 1 market data is multivariate. As explained in Chapter 2.2.2, we know that row-major order is preferred if values have an essential connection to each other, namely that they are used together. This connectivity is important for, e.g., monitoring tools.

Column-major order could be preferred if the data is used to monitor individual variables, for instance, the change in volume, or for analytical purposes where the asset's asking price is of sole importance.

3.2 Structure and properties of market data

In this section, we discuss the structure and distribution of Level 1 market data. Financial market data consists of multiple fields, essentially representing different univariate streams with identical timestamps. We take a closer look at each of the fields, discussing properties that may be exploited for efficient compression.

We have a closer look at the market data for Apple Inc., specifically. The stock is actively traded (Figure 2.1), providing a reference as to how the data from a single asset can be compressed.

We discuss the timestamps, the bid and offer prices, and the bid and offer sizes.

3.2.1 Timestamps

Chapter 2.1.3 states that market data has tick granularity; a sample is captured whenever a change of value occurs. The *Time* field, therefore, represents the time at which either of the Price or Size fields are modified. As the frequency depends on the rate of information, it reflects the asset's trading activity at a given point in time, visualized in Figure 2.1.

The unpredictability of the values in the Time field has negative consequences for its theoretical maximum compression rate. We know from Shannon [21] that higher entropy naturally yields lower possible compression rates. We can, however, reduce this entropy as much as possible.

Figure 3.1 shows the values of timestamps stored with previously discussed encodings for streams of values (Chapter 2.3.1). Both delta and delta-of-delta encodings reduce the magnitude of the timestamp values, reducing

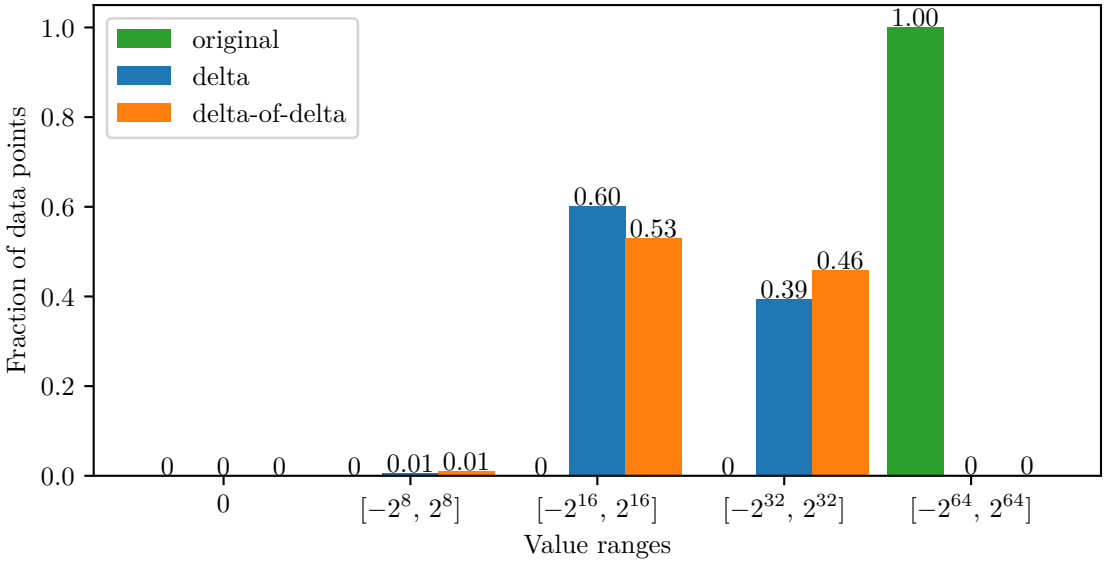


Figure 3.1: Distribution of values for timestamps for Apple Inc. on January 3rd, 2018. Delta and delta-of-delta values are computed as described in Chapter 2.3.1. $N = 1633158$.

the number of bits needed for their storage. Though each timestamp *originally* needed 64 bits, these encodings enable the values to be stored using less than half the amount of bits.

These results also indicate that delta encoding reduces the values slightly more than delta-of-delta encoding. We know from Chapter 2.3.1 that the latter is optimal with evenly spaced timestamps. This property is, however, unnatural for time series with tick granularity. Further testing and benchmarking will confirm whether this is the case for Level 1 market data in general.

3.2.2 Prices

Each sample in our market data contains two monetary values: the maximum price someone is willing to pay for an asset, the *Bid Price*; and the minimum price someone is willing to sell an asset for, the *Offer Price*. These values usually vary only slightly, in the so-called the bid-ask spread.

The tick level granularity of an asset's data stream implies that prices will not necessarily change between consecutive samples. A new data point

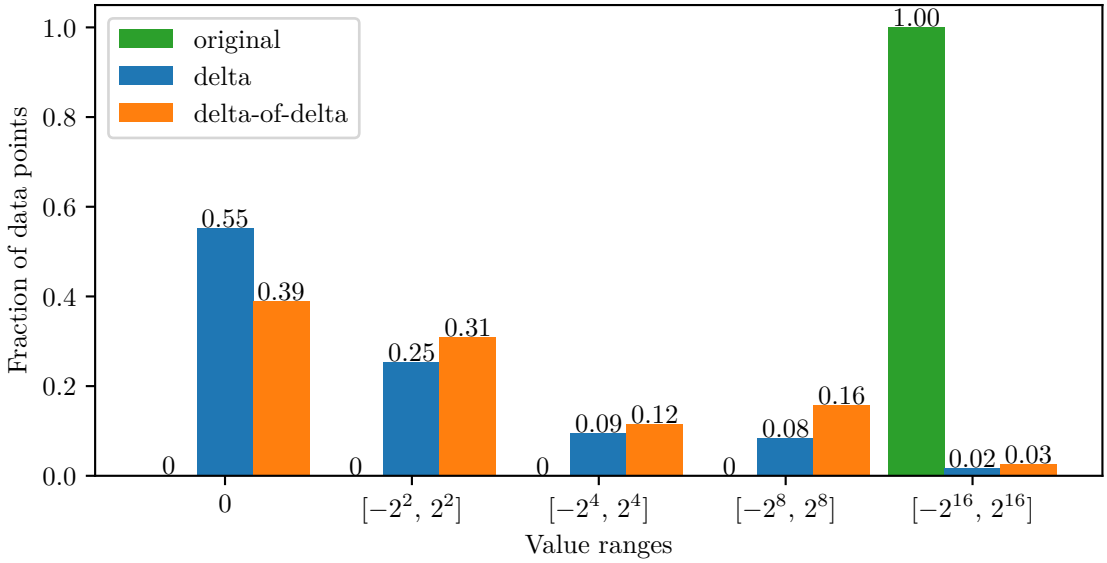


Figure 3.2: Distribution of values for bid and offer prices for Apple Inc. on January 3rd, 2018. Prices in cents. Delta and delta-of-delta values are computed as described in Chapter 2.3.1. $N = 3266316$.

could be sampled given a change in either the other Price field or one of the Size fields. It is, therefore, logically grounded that the values in the Price fields will often stay unchanged.

Figure 3.2 shows an example of how consecutive price values can differ. We see that delta and delta-of-delta encodings both produce a high number of zeros. We know from Chapter 2.3.2 that zeros are beneficial, as the compression algorithm can be configured to store the value as a single bit.

As with timestamps, delta encoding seems to yield lower entropy than delta-of-delta encoding for monetary values. These values will logically be close together, as the bid and offer prices are relatively constant over time. However, delta-of-delta encoding could be preferable if there is a general upwards or downward trend in the prices.

Even though Apple Inc.'s price fields seem to be using a maximum of 16 bits, this is not the case for the data sets studied from other stocks. The amount of information in the original price depends on the magnitude of its value. Empirical data shows that some data sets need more than 16 bits, so we support up to 32 bits in our implementations to support all cases.

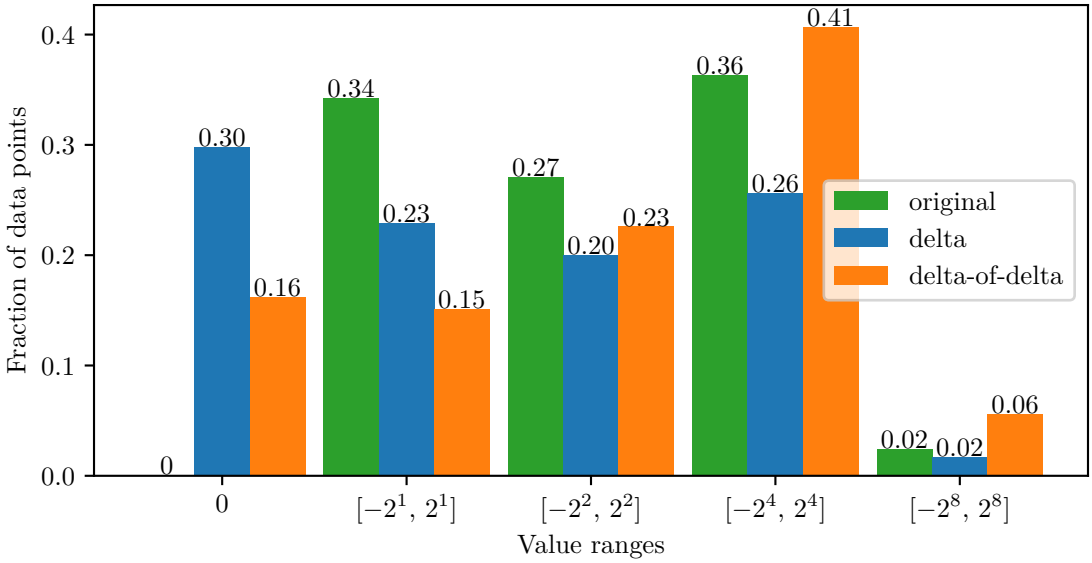


Figure 3.3: Distribution of values for bid and offer sizes for Apple Inc. on January 3rd, 2018. Delta and delta-of-delta values are computed as described in Chapter 2.3.1. $N = 3266316$.

3.2.3 Sizes

Bid and offer sizes quantify the volumes available at specific prices. As seen in Table 2.2 and Figure 3.3, these quantities are often small, which is logically grounded, as the positions with the highest bid and lowest offer prices are the least contested.

The insignificant magnitude of the values raises the question of whether compression of these fields is beneficial. Figure 3.3 shows that delta encoding produces slightly lower values. We know that values of zero are beneficial for a basic implementation of bit packing, significantly reducing the number of bits needed for their representation. The benefit of compressing these values is thus reduced to a trade-off between the speed overhead due to the comparative encoding, versus a slightly higher average number of bits per value. Benchmarking will show the actual impact of this trade-off.

As with the Price fields, bid and offer sizes need a full 32 bits representation. Some data sets have sizes of greater magnitude, with values exceeding 16 bits.

In closing, we see that the Size fields may have additional opportunities for

compression with increased customization. Bit packing needs a 5-bit header to represent up to 32 bits of information, but most uncompressed Size values can be represented with fewer than 5 bits, as seen in Figure 3.3. Using a smaller header, and storing outliers elsewhere, could benefit the compression rate.

3.3 Implemented algorithms

We have implemented and benchmarked several compression schemes to gather empirical data on the performance of different comparative techniques and storage methods. First, we benchmark different techniques on single fields, i.e., compression rates and speeds using various encodings on time-stamps, prices, and sizes. Afterward, we combine the implementations into compression schemes that work on multivariate financial market data.

In this section, we describe the implementation of the general compression algorithm. A subsample of ticks from Apple Inc. is used as an example.

Table 3.1: Example of initial structure of market data for Apple Inc. Each tick is first loaded into memory to closer simulate reading from a stream.

#	Sample
1	80000189974662 | 172.26 | 2 | 172.55 | 2
2	80000190231252 | 172.26 | 2 | 172.55 | 1
3	80000190619305 | 172.26 | 2 | 172.63 | 1
4	80001307295190 | 172.52 | 3 | 172.63 | 1
5	80001388058920 | 172.52 | 3 | 172.62 | 1

3.3.1 Compression

The outline of the compression algorithm is as follows:

1. Initialize
2. Process ticks
   (a) Comparative encoding (none, delta, or delta-of-delta)
   (b) Zigzag encoding
   (c) Bit packing
3. Finalize

Initialization

The data is initially stored on disk, where each sample from a specified asset is on the form:

$$Time|Bid\ Price|Bid\ Size|Offer\ Price|Offer\ Size,$$

as exemplified in Table 3.1. We load each sample into memory to remove the speed impact of reading from disk.

We then create a bit vector for storing the full compressed result and allocate memory, assuming a compression rate of 1:1 for the block. This preallocation is crucial as continuous reallocation of the resulting bit vector—an operation linear on the number of elements [22]—significantly reduces compression speed.

Table 3.2: Samples parsed from the data in Table 3.1.

#	Time	Bid Price (c)	Bid Size	Offer Price (c)	Offer Size
1	80000189974662	17226	2	17255	2
2	80000190231252	17226	2	17255	1
3	80000190619305	17226	2	17263	1
4	80001307295190	17252	3	17263	1
5	80001388058920	17252	3	17262	1

Table 3.3: Delta encoding of samples in Table 3.2.

#	Time	Bid Price (c)	Bid Size	Offer Price (c)	Offer Size
1	80000189974662	17226	2	17255	2
2	256590	0	0	0	-1
3	388053	0	0	8	0
4	1116675885	26	1	0	0
5	80763730	0	0	-1	0

Comparative encoding

For each sample, each field is then parsed, normalized, and stored in its appropriate data type, as shown in Table 3.2. The fields are then compressed using the specified encoding resulting in the sample values as exemplified in Table 3.3. In the case of "none", this step is skipped. For delta and delta-of-delta encoding, the first value is stored as-is, with consecutive values following as defined in Chapter 2.3.1.

Zigzag encoding

For variables that could be negative, zigzag encoding (Chapter 2.3.2) is performed in preparation for bit packing. Table 3.4 and Table 3.5 illustrates this transformation for the Offer Size value, which results in a long head of consecutive zeros, allowing for more efficient bit packing.

Bit packing

Each value in a sample is packed independently, as shown in Table 3.6. Bit packing is performed using $log_2(64) = 6$ header bits for the Time field, as

Table 3.4: Binary values of a delta encoded sample. The sample is the same as the corresponding sample in Table 3.3.

#	Field	Bits
2	Time	00000000000000000000000000000000000000000000000011111010100100111
	Bid Price	00000000000000000000000000000000
	Bid Size	00000000000000000000000000000000
	Offer Price	00000000000000000000000000000000
	Offer Size	11111111111111111111111111111111

Table 3.5: Zigzagged binary values of the sample in Table 3.4. Time is not zigzagged as the delta is always positive.

#	Field	Bits
2	Time	00000000000000000000000000000000000000000000000011111010100100110
	Bid Price	00000000000000000000000000000000
	Bid Size	00000000000000000000000000000000
	Offer Price	00000000000000000000000000000000
	Offer Size	00000000000000000000000000000001

its values need the full 64 bits when *no-comparative-encoding* ("none") is performed, and in some cases even for delta and delta-of-delta encoded values.

Prices and sizes need $log_2(32) = 5$ bits for the header. Empirical data show that these fields use only slightly over 16 bits of storage, and never exceeds the 32 bits a 5-bit header provides.

The bit packed values are then merged into a single vector representing the complete sample (Table 3.7). Finally, the sample's bit vector is appended to the block's bit vector.

Finalize

When N samples are compressed, where N is the block size, the block is full. The compression is then restarted with a new block, with sample $N + 1$ as the initial sample.

The result is that all samples are stored compressed in memory, except for each block's initial sample. A sample is available by decompressing its block.

Table 3.6: Bit packed binary values of the sample in Table 3.5. The green bit indicates whether the *value* is non-zero. Yellow, the number of bits in the value. Red, the actual non-zero value.

#	Field	Packed bits
	Time	10100101111101010010011 10
	Bid Price	0
2	Bid Size	0
	Offer Price	0
	Offer Size	1 000011

Table 3.7: Complete compressed representation of the corresponding sample in Table 3.3. The green bit indicates whether the *value* is non-zero. Yellow, the number of bits in the value. Red, the actual non-zero value.

#	Compressed sample
2	10100101111101010010011100001 000011

3.3.2 Decompression

Decompression is essentially the reverse process of compression. By using known header sizes, values can be extracted sequentially from the bit vector describing the $2, ..., N$ following samples in the block, N being the block size.

To decompress delta or delta-of-delta encoded fields, it is worth noting that we need to keep track of the aggregated delta. To compute the value of a sample, we need to compute the delta of each preceding value of that same field in the block.

3.4 Benchmarking

This section introduces the techniques, and the benchmarks run to evaluate their viability. Finally, we specify the hardware used for the benchmarking.

3.4.1 Techniques

We benchmark no-comparative-encoding, delta encoding, and delta-of-delta encoding on different data sets. Zigzagging is performed where appropriate, followed by bit packing.

For each technique, we assess the compression rate, compression speed, and decompression speed. The compression rate is measured relative to the number of bits used to store the original value, e.g., 64 bits for each Time value.

3.4.2 Benchmarks

We first benchmark the individual fields to assess the performance of the discussed techniques on each variable. Compressing the columns in order also reveals the performance of storing the data in column-major order.

We benchmark the algorithm on the Time field, followed by price and size. For price and size, we focus on the benchmarks for Bid Price and Bid Size, respectively. We know, in part from Figure 3.2 and Figure 3.3, that the values, distribution, and format for the two price and the two size fields are close to identical. We therefore only study one of each.

We then benchmark the algorithm on the full multivariate data, storing the compressed data in row-major order. We use the same encoding on each field, and a *hybrid* scheme with different encodings on different fields, where there is an indication of a better compression rate or speeds.

To evaluate the implementation, benchmarks are run on financial market data from different securities traded on the New York Stock Exchange (NYSE). We use freely available sample data from January 3rd, 2018 [13]. We look at data from the most actively traded stocks, as these have the most benefit from compression due to the significant number of data points. The top 100 most actively traded stocks available in the data set have, on average, $1.1M$ samples of trading data on this date. We consider the five relevant fields from each of the samples.

We first benchmark Apple Inc., following up on the analysis from Chapter 3.2. When considering a single stock, we run 100 benchmarks and average the results. We then benchmark the top 100 most active stocks, including Apple Inc., and average the results. The latter results are truncated to the size of the smallest data set.

3.4.3 Hardware

The implementation is benchmarked on a 2018 MacBook Pro with a 2.3GHz Quad-Core Intel Core i5 processor and 16GB of random access memory (RAM). Input and output data is stored in RAM.

Chapter 4

Results and Discussion

With the knowledge of structures and properties of financial market data that could enable compression (Chapter 3.2), we can test these assumptions using the discussed techniques on a broad range of data sets. This chapter introduces and dwells on the results of these benchmarks.

In Chapter 4.1, we present and analyze the results, before we—in Chapter 4.2—discuss the speed, block size, and major order for compression of the full multivariate data.

4.1 Benchmark results

This section presents the results of our benchmarks, as described in Chapter 3.4. We first look at the benchmark results from compressing individual fields. We then consider the results from the full multivariate data.

4.1.1 Individual fields

We benchmark individual fields to assess the performance of the discussed techniques on each column in isolation. In this section, we look at the benchmark results from compressing the Time field, a Price field, and a Size field.

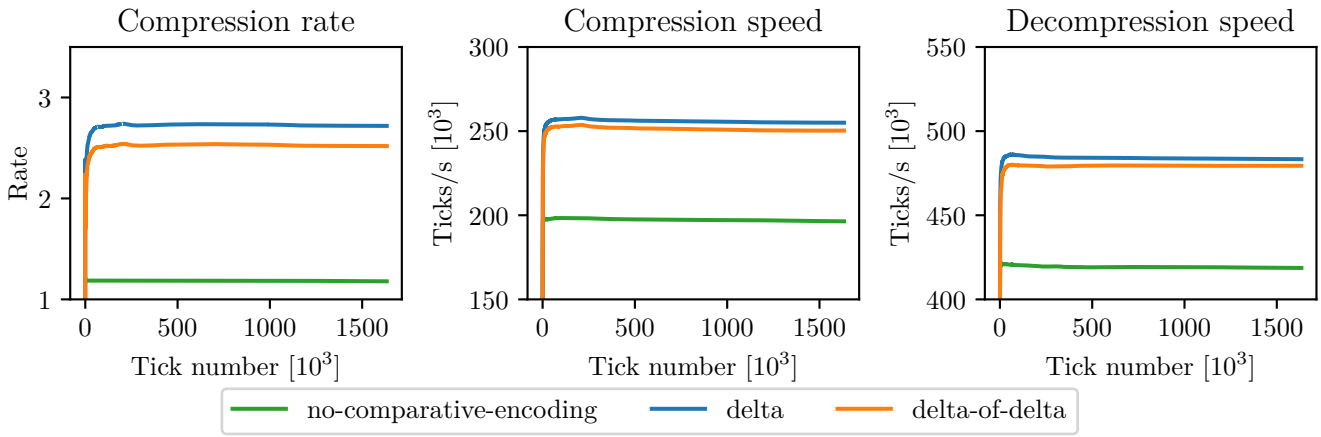


Figure 4.1: Benchmark results from compressing the Time field for Apple Inc.'s market data (January 3rd, 2018). Using delta encoding, values of 64 bits are reduced to $64/2.7 \approx 24$ bits.

Time

The results from benchmarking the Time field using the discussed techniques are displayed in Figure 4.1 and Figure 4.2.

In Chapter 3.2.1, we theorized that compressing randomly spaced timestamps yields only slightly lower entropy. Timestamps encoded with delta or delta-of-delta encoding are reduced to between 1/4 and 1/2 the number of bits. Figure 4.1 confirms that this is indeed the case for this data set. The observed compression rate is on the lower end (2.7), likely due to the additional $1 + 6 = 7$ bits needed for bit packing.

The compression rate mirrors the compression and decompression speeds. It can thus be assumed that the speed overhead of delta and delta-of-delta encoding is significantly less than that of bit packing. We know from Chapter 2.3.2 that bit packing is more efficient on smaller values. This statement supports a hypothesis that the smaller values resulting from comparative encoding have faster bit packing (and unpacking), leaving no-comparative-encoding with significantly lower speeds.

The results also reveal that we reach the maximum rate and speeds with high velocity. The overhead of storing the initial value without any form of compression—using a full 64 bits—is thus of limited impact. Using a small

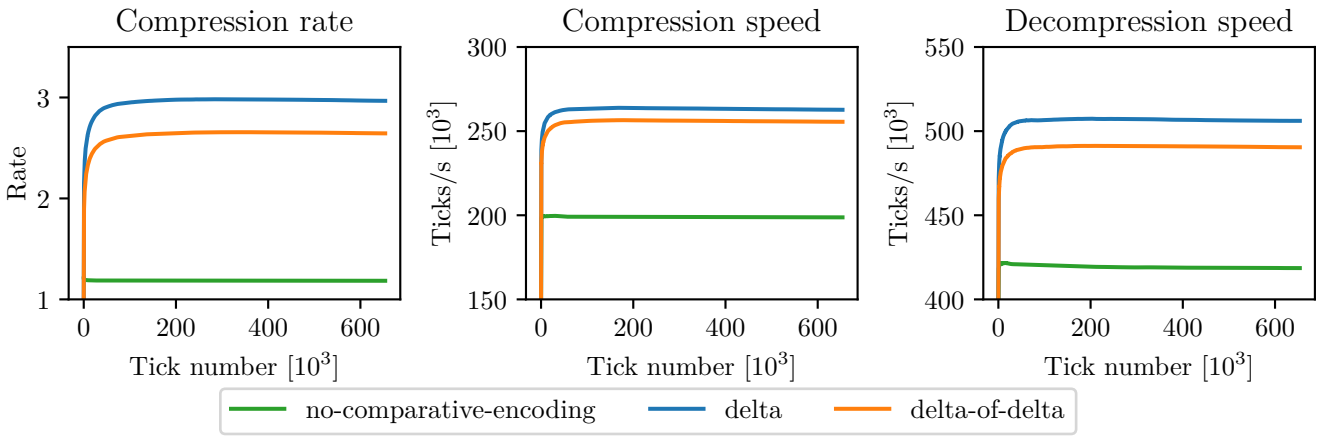


Figure 4.2: Average benchmark results from compressing the Time field of the market data from the top 100 most actively traded stocks (January 3rd, 2018). Using delta encoding, values of 64 bits are reduced to 64/3.0 ≈ 21 bits.

block size is therefore possible.

Lastly, we see that these statements are also valid for the general case, as displayed in Figure 4.2.

Price

The results from benchmarking the Bid Price field using the discussed techniques are displayed in Figure 4.3 and Figure 4.4.

We theorized in Chapter 3.2.2 that the Price fields have significant benefits from comparative encoding. Consecutive values are often identical, providing a large portion of zeros. We also have a general skew towards lower values. The results show that this is indeed the case, and we are able to reduce the average size from 32 bits to slightly less than 5 bits for the Apple Inc. data set.

Figure 4.4 shows that the compression rate is even higher for the general case. The magnitude of the price of a stock varies greatly. The average price for the top 100 most actively traded stocks is lower than that of the data in Figure 4.3. This is illustrated by the lower compression rate for Apple Inc. using no-comparative-encoding (1.5), versus the compression rate for

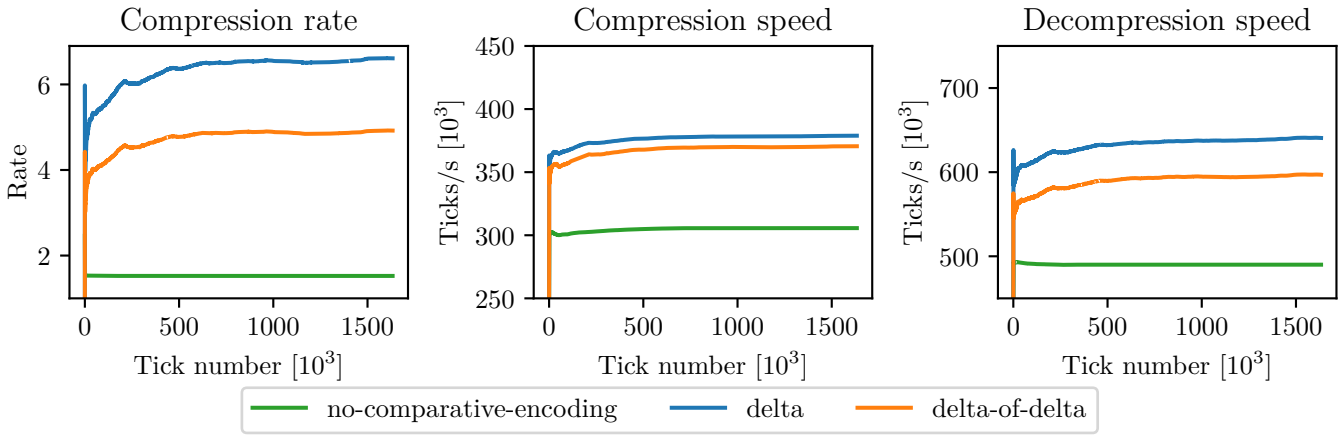


Figure 4.3: Benchmark results from compressing the Bid Price field of Apple Inc.'s market data (January 3rd, 2018). Using delta encoding, values of 32 bits are reduced to just over $32/6.6 \approx 5$ bits.

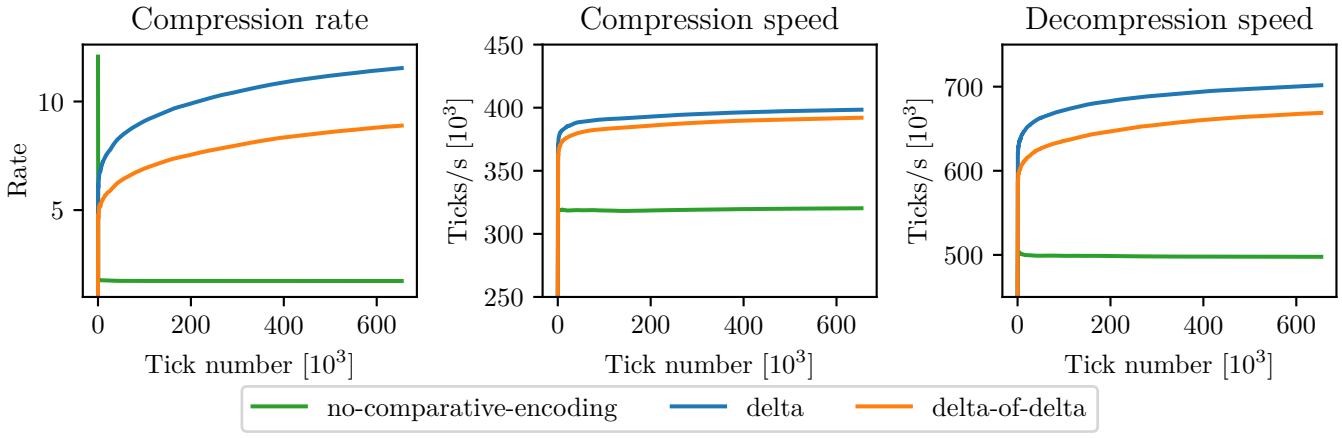


Figure 4.4: Average benchmark results from compressing the Bid Price field of the market data from the top 100 most actively traded stocks (January 3rd, 2018). Using delta encoding, values of 32 bits are reduced to $32/11.5 \approx 3$ bits.

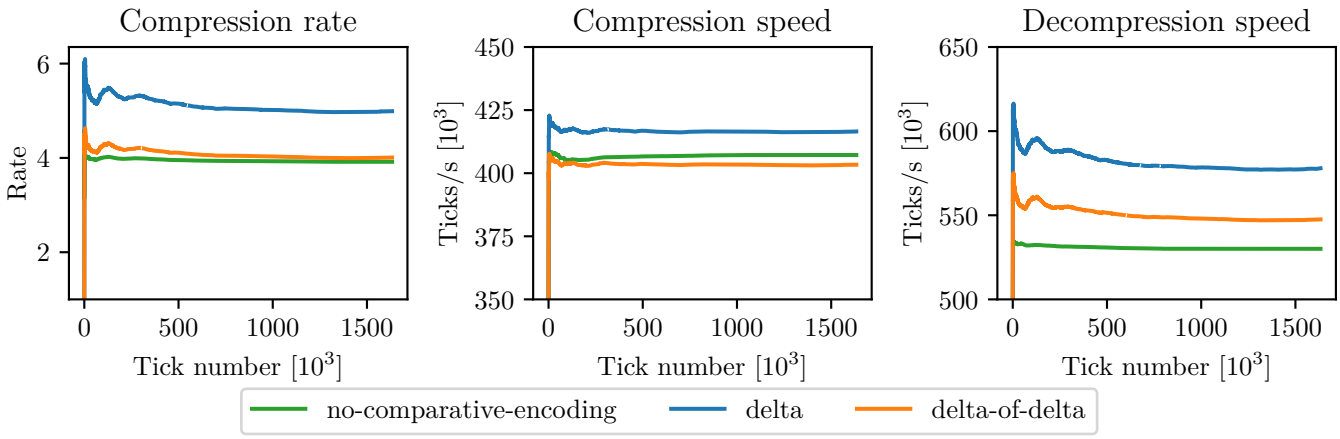


Figure 4.5: Benchmark results from compressing the Bid Size field of Apple Inc.'s market data (January 3rd, 2018). Using delta encoding, values of 32 bits converge towards $32/5.0 \approx 6$ bits.

the average case using the same technique (1.7). Higher compression rates are thus possible for the latter. These values can be compressed to about 3 bits each.

As with the Time field, a higher compression rate yields higher speeds.

Size

The results from benchmarking the Bid Size field using the discussed techniques are displayed in Figure 4.5 and Figure 4.6.

In Chapter 3.2.3, we hypothesized that the size values only have a slight benefit from comparative encoding. The results show that this is indeed the case. Compression rates spike for low tick numbers due to low values and entropy in early-morning trading volumes. The compression rate then flattens around 5 for Apple Inc. and 4 in the general case, when using delta encoding.

While delta encoding produces a slightly higher compression rate, both delta and delta-of-delta encoding have mediocre results compared to using no-comparative-encoding. Sizes are essentially random low-digit values, making the zeros produced be the only noteworthy benefit from comparative encoding.

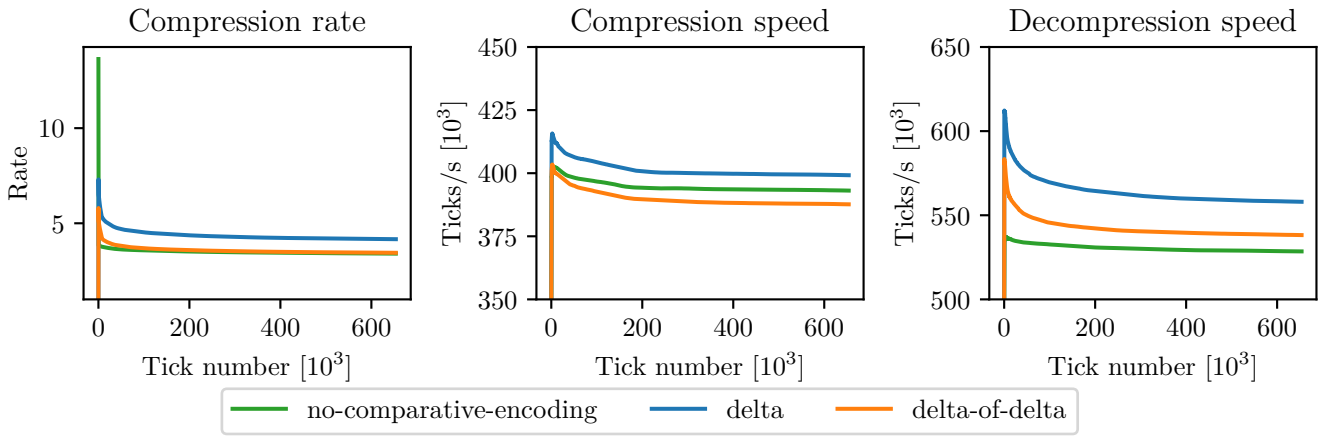


Figure 4.6: Average benchmark results from compressing the Bid Size field of the market data from the top 100 most actively traded stocks (January 3rd, 2018). Using delta encoding, values of 32 bits converge to $32/4.2 \approx 8$ bits.

The insignificant benefit of comparative encoding naturally raises the question of whether this type of encoding is necessary for the Size fields. However, as previously observed, there are indications that increased compression rates might be more critical for the speeds than the overhead of the encoding.

4.1.2 Multivariate data

Figure 4.7 and Figure 4.8 shows the results from benchmarking the full multivariate data for Apple Inc. and the average benchmark results from the top 100 most actively traded stocks, respectively. The data is processed and stored in row-major order (Chapter 4.2.3), where the labeled technique is used on all fields. For the hybrid scheme, delta encoding is used on the Time and Price fields, and no-comparative-encoding on the Size fields, as discussed in Chapter 3.4.2.

Unsurprisingly, we see that using the most efficient compression technique for each field also yields the best results for compressing the full multivariate data. We know that the benchmark results for the multivariate data are a weighted average of the benchmark results for each field. For instance, the

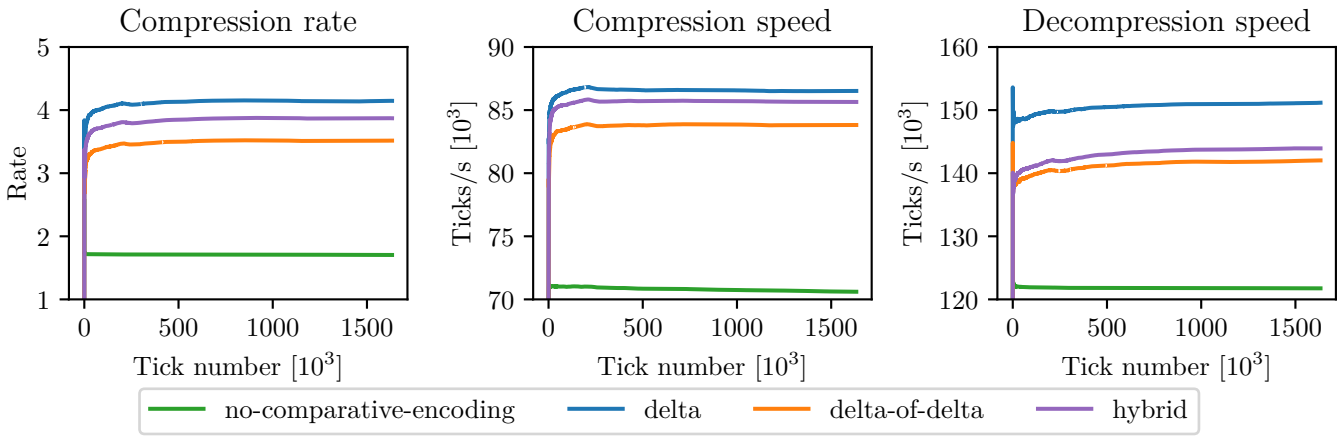


Figure 4.7: Benchmark results from compressing Apple Inc.'s market data (January 3rd, 2018). The labeled technique is used on all five fields. hybrid uses delta encoding for Time and prices, and no-comparative-encoding for sizes. The samples are stored in row-major order. Delta encoding reduces the size to $192/4.1 \approx 47$ bits.

64 bits in an original timestamp constitute a more substantial portion of the original sample than the other values, each represented by 32 bits. The compression rate of the Time field is thus of higher relative impact.

Further, we see that the compression rate and speed of the algorithm start to converge after only a few thousand ticks. The decompression speed increases slightly even after this point.

More surprisingly, we see that the hybrid scheme—i.e., compressing Time and Price fields with the optimal delta encoding, and Size fields with no-comparative-encoding—have worse results for all measurements. The lower compression rate seems to take a slightly higher toll on the compression speed than the benefits of removing the field's encoding. This observation is in line with the assumption from Chapter 4.1.1, confirming that the threshold for how much a comparative encoding has to compress the variable to be beneficial on speed is indeed small. The cost of storing the values in blocks—i.e., zigzagging and bit packing—is much higher than that of comparative encoding.

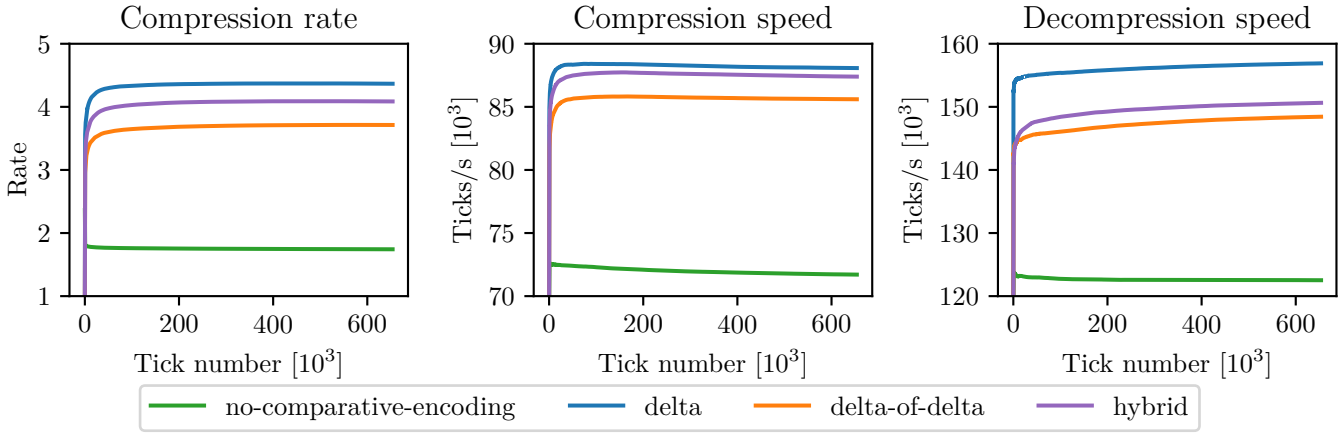


Figure 4.8: Average benchmark results from compressing the market data from the top 100 most actively traded stocks (January 3rd, 2018). The labeled technique is used on all five fields. hybrid uses delta encoding for Time and prices, and no-comparative-encoding for sizes. The samples are stored in row-major order. Delta encoding reduces the size to $192/4.4 \approx 44$ bits.

4.2 Discussion of technical properties

Having introduced the benchmark results from compressing the full multivariate data, we can discuss the important properties of the compression scheme. In this section, we discuss the speed, block size, and major order for the proposed scheme for continuous lossless compression of multivariate financial market data.

4.2.1 Speed

The benchmark results from the full multivariate data show that our simple implementation of the algorithm obtains compression speeds north of 85×10^3 ticks/s. We know from Chapter 4.2.1 that the compression speed needs to keep up with the injection rate of samples from each specific asset. While market data is highly random, and future frequencies are unpredictable, initial observations show that the highest frequencies of market data are in the thousands, exemplified for Apple Inc. in Figure 2.1. The peak frequencies of market data are thus expected to be one order of magnitude lower than the obtained compression speeds of this implementation.

It is expected that further optimization of the compressor, e.g., using SIMD techniques (Chapter 2.3.2) for parallel calculation of deltas in each row, can increase speeds significantly.

The results reveal a strong correlation between the compression rate and speed. Due to bit packing's speed being linear in the number of bits (Chapter 2.3.2), the compression speed is more dependent on this value, than the comparative encoding used. The speed of delta and delta-of-delta encoding on a multivariate sample is primarily dependent on the number of values—i.e., columns—in the sample, with delta-of-delta encoding having a slightly higher constant factor due to calculating two deltas.

Unlike compression speed, decompression speed is not directly dependent on the size of the values. We know from Chapter 2.3.2 that the speed of unpacking a value has a constant upper bound, where the deciding factor is whether the value is zero or not, with zero reducing the decompression speed slightly. This lack of dependency is illustrated in Figure 4.6, where the considerable reduction in bit size at early tick numbers yields a minimal increase in decompression speed when using no-comparative-encoding, as the values are still non-zero. Delta and delta-of-delta encoding, however, produce a large number of zeros, increasing the decompression speed significantly.

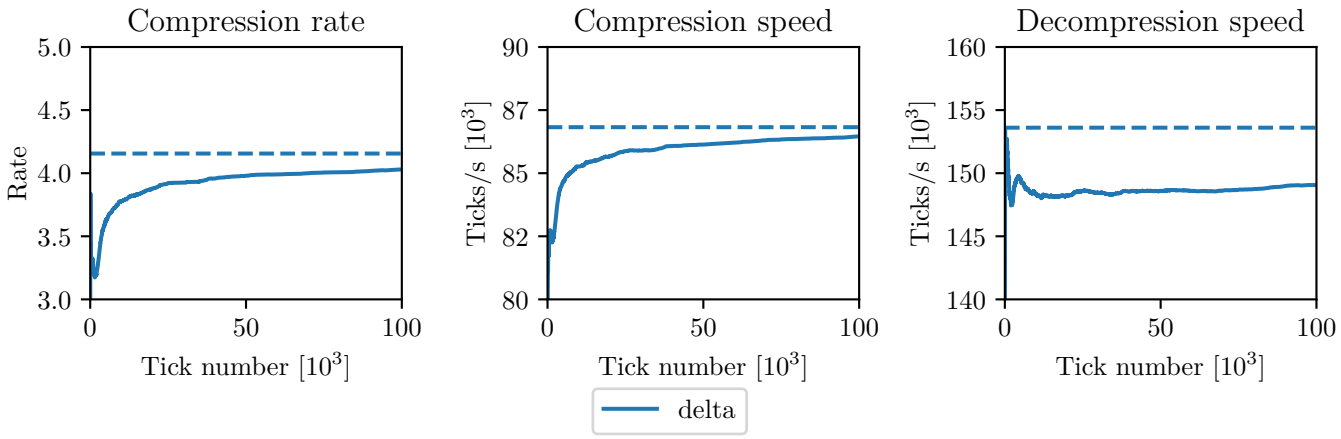


Figure 4.9: Snippets of the benchmark results from compressing Apple Inc.'s market data (January 3rd, 2018). Delta encoding is used on all five fields. The samples are stored in row-major order. The dotted line indicates the maximum values in the overarching results.

4.2.2 Block size

Figure 4.9 and Figure 4.10 show snippets of the lowest tick numbers from the multivariate benchmark results in Chapter 4.1.2. We see that the compression rate slowly starts to converge after about 25×10^3 ticks, reaching optimal values at around 100×10^3 ticks. Decompression speeds have a slight benefit from even larger blocks, with speeds increasing linearly after 25×10^3 ticks.

A block size of 100×10^3 ticks, with samples stored in row-major order, produces blocks with a memory footprint of less than 550 kB, assuming a compression rate of 4.4, as indicated by Figure 4.8. We know from Chapter 2.2.2, that small block sizes are highly beneficial, as they reduce the number of additional samples that need to be decompressed when retrieving a single entry.

For time series, a small block size is particularly interesting in the case of range search. Finding the start and end value in a range would need, on average, 275 kB + 275 kB = 550 kB additional decompression overhead; 275 kB for finding each of the values, assuming a trivial sequential scan over their compressed block. This overhead is constant for any range with $N_{blocks} > 1$,

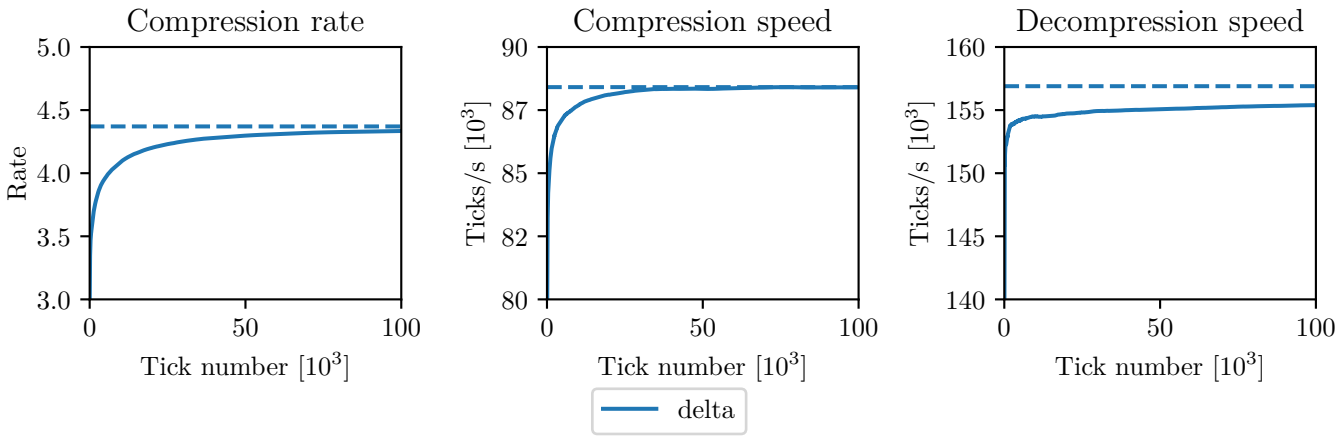


Figure 4.10: Snippets of the average benchmark results from compressing the market data from the top 100 most actively traded stocks (January 3rd, 2018). Delta encoding is used on all five fields. The samples are stored in row-major order. The dotted line indicates the maximum values in the overarching results.

and is therefore of less importance when extracting a broader range of values where a significant number of blocks have to be decompressed.

4.2.3 Major order

While the benchmark results in Chapter 4.1.2 are measuring the proposed algorithm using row-major order, column-major order is equivalent to compressing each column in order; sequential compression and decompression of each field. The results in Chapter 4.1.1 thus reflect the performance of a column-major version of the same algorithm.

We know from Chapter 2.1.1 that time series are unique in that each value is connected to a specific timestamp. This connectivity has an unfortunate side-effect when storing data in column-major order. While we can see from the benchmark results that storing each column separately yields increased speeds, the values are disconnected from their timestamp. Accessing a specific column by time thus requires us first to decompress and search the Time column for the indexes, and then the desired column. Though the benchmark results confirm that this may still be beneficial with regards to

speed, as opposed to row-major storage, it is both a more complicated task and less effective when the number of attributes queried for approaches the dimension of the data set.

A different approach is to store the multivariate data as multiple univariate time series. This approach has the added benefit of doing single-column lookups efficiently, e.g., accessing Bid Prices over a specific time interval. The obvious downside is the need to store the Time column with *each* of the D variables. Though this can be beneficial when the Time column is highly compressible, it increases the total storage footprint considerably for financial market data, where the field is of low compressibility.

In general, we see that row-major order is the preferred storage method for multivariate market data when the connectivity between variables is of importance. It requires no additional storage of timestamps while still enabling non-complex and relatively fast lookups of values by time. Additionally, row-major order has the added potential of leveraging SIMD, increasing the speeds significantly by calculating the delta of each value in parallel.

Chapter 5

Conclusion and Future Work

Software systems process ever-increasing amounts of real-time data. Recent work has shown that compression can significantly increase the performance of such systems. We have analyzed and assessed a scheme for continuous lossless compression of streams of high-frequency multivariate financial market data. The properties of this type of data require combining multiple efficient methods to obtain suitable rates and speeds.

We argue that the compression of multivariate streams is essentially the same as compressing multiple univariate streams with a shared time dimension. By studying current state-of-the-art systems for compression of univariate time series, we have implemented a novel algorithm for lossless compression of multivariate market data. The variables are processed independently, where each sample is compressed continuously, making no assumptions on future distributions.

Our implementation assumes the values of a sample to be interconnected. We show that row-major order is preferred, as it obtains suitable results without the increased overhead of storing each variable as a univariate time series. This storage is highly unfortunate as the timestamp dominates the information and storage footprint in high-frequency market data.

By leveraging techniques such as comparative encoding, we can continuously compress multivariate samples of high randomness losslessly, increasing the number of recent samples storable in-memory by a factor of 4.4 on average. Compression speeds one order of magnitude higher than the frequency of the most actively traded assets, and one-pass decompression with blocks of less than 550 kB, indicates that such techniques may be suitable for in-memory storage in data-intense real-time systems.

5.1 Future Work

The primary limiting factor of any compression scheme is in its compression rate or execution speeds. We expect a small upside on the compression rate by using more efficient compression techniques, and a significant upside on compression and decompression speeds, for instance, by leveraging advanced hardware instructions.

The compression rate is limited by the requirement of compressing each sample continuously and losslessly. The unpredictability of the streams yields a high amount of entropy, reducing the maximum possible compression rate. Regardless, more efficient compression techniques, such as optimized headers for storing each value, or combining and storing headers externally, could increase the compression rate (Chapter 3.2.3).

Compression and decompression speeds are not optimized in this thesis and have significant upsides. By leveraging techniques such as SIMD for calculating the delta of each variable in parallel, we can significantly reduce the number of instructions needed to process a sample (Chapter 4.2.1).

Finally, use case specific tailoring could be a fruitful avenue for future work. For instance, performing a second stage compression of blocks in isolation could further reduce the storage footprint of the data. Initial tests show that using the proposed algorithm and then compression the result using a universal lossless compression algorithm such as LZW (Chapter 2.2.1), yields increased compression rates. This rather obvious find shows that block-level compression could be applied for further reduction in size. Whether higher compression and decompression speeds are preferred versus an increased amount of data points stored in memory depends, as always, on the specific use case.

Bibliography

[1] M. M. Dacorogna, *An introduction to high-frequency finance*. Academic Press, 2001, p. 6, ISBN: 0122796713.

[2] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, In-Memory Time Series Database," Tech. Rep., 2150.

[3] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *VLDB Journal*, vol. 24, no. 2, pp. 193–218, 2015, ISSN: 0949877X. DOI: 10.1007/s00778-014-0368-8.

[4] N. I. Martina Rejsjö, head of Nasdaq Surveillance North America Equities, *"The massive and, in many cases, exponential growth in market data is a significant challenge for surveillance professionals"*. [Online]. Available: https://www.bloomberg.com/news/articles/2019-12-06/robots-in-finance-could-wipe-out-some-of-its-highest-paying-jobs (visited on 12/07/2019).

[5] Yaron Minsky, *"The US equity markets alone can peak at roughly 5 million messages per second, and volumes on the options markets are even higher."*, 2018. [Online]. Available: https://blog.janestreet.com/what-the-interns-have-wrought-2018/ (visited on 11/24/2019).

[6] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," Sep. 2012. DOI: 10.1002/spe.2203. arXiv: 1209.2137. [Online]. Available: http://arxiv.org/abs/1209.2137%20http://dx.doi.org/10.1002/spe.2203.

[7] Nico van der Wijst, *Finance: A Quantitative Introduction*. Cambridge University Press, 2013, p. 208, ISBN: 1107029228.

[8] T. A. Welch, "A Technique for High-Performance Data Compression," *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984, ISSN: 0018-9162. DOI: 10.1109/MC.1984.1659158. [Online]. Available: https://doi.org/10.1109/MC.1984.1659158.

[9] D. Blalock, S. Madden, and J. Guttag, "Sprintz," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, Sep. 2018. DOI: 10.1145/3264903.

[10] I. Aldridge, *High-Frequency Trading - A Practical Guide to Algorithmic Strategies and Trading Systems (Wiley Trading)*. Wiley, 2009, pp. 2, 61, ISBN: 0470563761.

[11] London Stock Exchange, *What is Real Time Data*. [Online]. Available: https://www.lseg.com/markets-products-and-services/market-information/real-time-data/what-real-time-data (visited on 06/07/2020).

[12] NYSE, *Daily TAQ Client Specification*, 2013. [Online]. Available: https://www.nyse.com/publicdocs/nyse/data/Daily%7B%5C_%7DTAQ%7B%5C_%7DClient%7B%5C_%7DSpec%7B%5C_%7Dv3.0d.pdf (visited on 12/08/2019).

[13] ——, *NYSE Daily TAQ*. [Online]. Available: https://www.nyse.com/market-data/historical/daily-taq (visited on 12/08/2019).

[14] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley Professional, 1997, vol. 1, pp. 298–305, 24, ISBN: 9780201896831. DOI: 9780201896831.

[15] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, p. 13, Oct. 1985. DOI: 10.1109/IEEESTD.1985.82928.

[16] S. Mittal and J. S. Vetter, *A Survey of Architectural Approaches for Data Compression in Cache and Main Memory Systems*, May 2016. DOI: 10.1109/TPDS.2015.2435788.

[17] D. Lemire, L. Boytsov, and N. Kurz, *SIMD compression and the intersection of sorted integers*, 2016. DOI: 10.1002/spe.2326. arXiv: 1401.6399.

[18] J. Von Neumann and M. D. Godfrey, "First Draft of a Report on the EDVAC," Tech. Rep. 4, 1993, pp. 27–75. DOI: 10.1109/85.238389.

[19] Google, *Protocol buffers encoding*. [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding (visited on 05/27/2020).

[20] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast Lossless Compression of Scientific Floating-Point Data," Tech. Rep.

[21] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, ISSN: 15387305. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[22] Cplusplus, *Std::vector::resize*. [Online]. Available: http://www.cplusplus.com/reference/vector/vector/resize/ (visited on 03/20/2020).

Appendix A

Delta-of-delta encoding

We have the initial value defined as

$$x_1 = x_0 + \delta_1,$$

where x_0 is a reference value. The next value is equal to the previous value, x_1, plus the change

$$x_2 = x_1 + (\delta_1 + \delta_2^2).$$

As the change for each consecutive value is *relative* to all previous changes, we have that the next value is

$$x_3 = x_2 + (\delta_1 + \delta_2^2 + \delta_3^2).$$

That is, for each x_i, we need to add $\delta_1 + \sum_{j=2}^{i} \delta_j^2$ to x_{i-1}. We thus have that

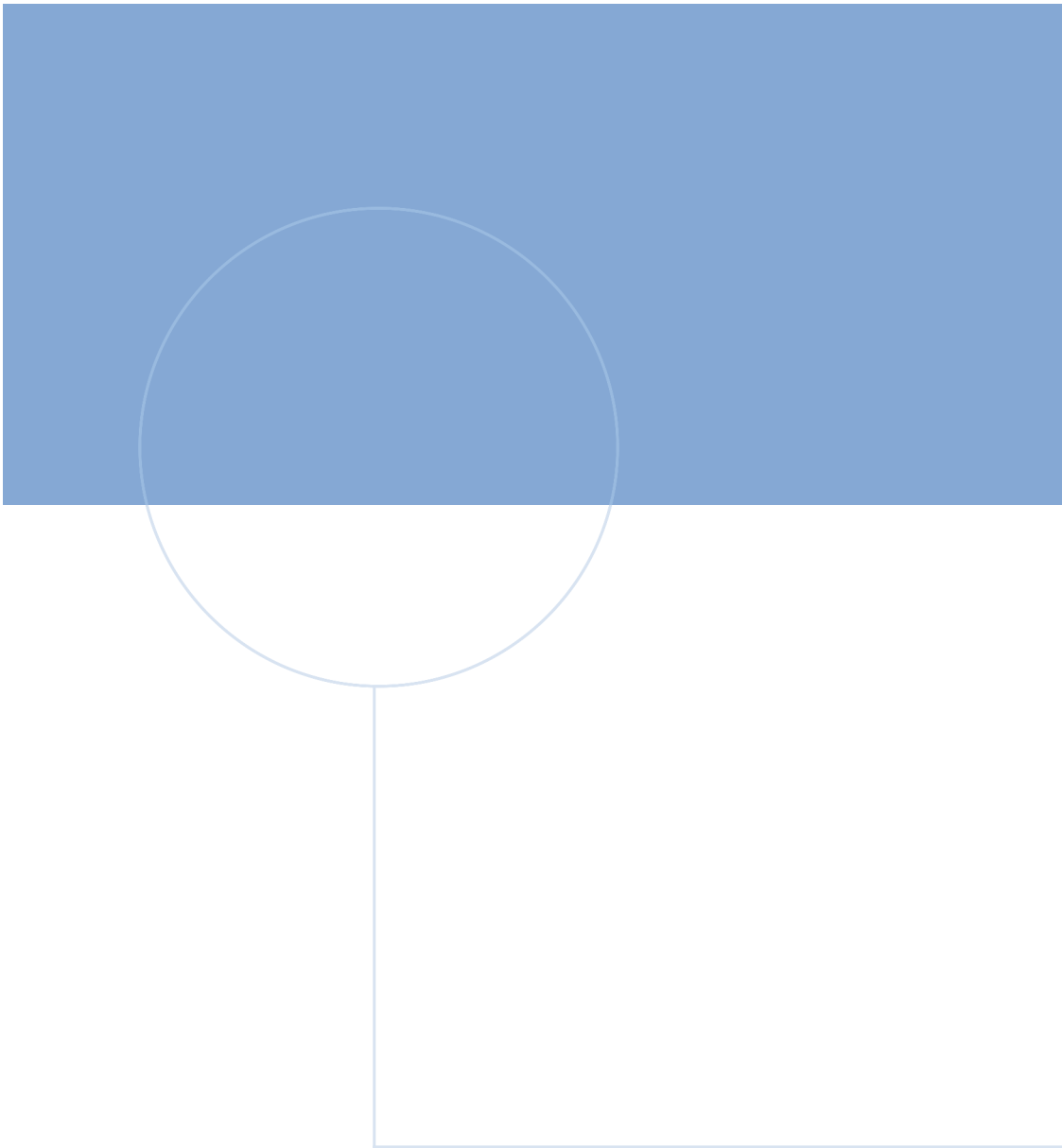
$$x_i = (x_0 + \delta_1) + (\delta_1 + \delta_2^2) + (\delta_1 + \delta_2^2 + \delta_3^2) + ... + (\delta_1 + \delta_2^2 + \delta_3^2 + ... + \delta_i^2),$$

which can be rewritten as

$$x_i = x_0 + i\delta_1 + (i-1)\delta_2^2 + (i-2)\delta_3^2 + ... + \delta_i^2.$$

This can be formulated generally as

$$x_i = x_0 + i\delta_1 + \sum_{j=2}^{i}(i-j+1)\delta_j^2.$$



NTNU

Norwegian University of
Science and Technology