Sprintz: Time Series Compression for the Internet of Things

Davis Blalock
Computer Science and Artificial Intelligence Laboratory
MIT
dblalock@mit.edu

Sam Madden
Computer Science and Artificial Intelligence Laboratory
MIT
madden@csail.mit.edu

John Guttag
Computer Science and Artificial Intelligence Laboratory
MIT
guttag@csail.mit.edu

ABSTRACT

Due to the rapid proliferation of connected devices, sensor-generated time series constitute a large and growing portion of the world's data. In order to reduce storage, power, and transmission costs, it is desirable to compress this data.

We introduce a time series compression algorithm that achieves state-of-the-art compression ratios and up to twice the speed of its closest rivals. Moreover, because our approach requires <1KB of memory, it can be run on low-power sensors at the network edge, offloading computation from the central database and reducing network load. A key component of our method is a high-speed forecasting algorithm that can be trained online and significantly outperforms alternatives such as delta coding.

Extensive experiments on datasets from many domains show that these results hold not only for sensor data, our primary application of interest, but also across a wide array of other time series.

PVLDB Reference Format:
Davis Blalock, Sam Madden, John Guttag. Sprintz: Time Series Compression for the Internet of Things. *PVLDB*, 11 (5): xxxx-yyyy, 2018.
DOI: https://doi.org/TBD

1. INTRODUCTION

Time series are ubiquitous and only growing in importance thanks to the proliferation of sensor data from autonomous vehicles, smart phones, wearables, and other connected devices. Much of this data is currently stored in databases—many of them designed specifically for time series [10, 57, 52, 29, 61, 7, 45]. Consequently, an improved means of compressing time series could greatly enhance the storage, network, and computation efficiency of countless database systems.

From a compression perspective, time series have four attributes uncommon in other data.

Proceedings of the VLDB Endowment, Vol. 11, No. 5
Copyright 2018 VLDB Endowment 2150-8097/18/1.
DOI: https://doi.org/TBD

1. **Lack of exact repeats**. In text or structured records, there are many sequences of bytes—often corresponding to words or phrases—that will exactly repeat many times. This makes dictionary-based methods a natural fit. In time series, however, the presence of noise makes exact repeats less common [8, 47].

2. **Multiple variables**. Real-world time series often consist of multiple variables that are collected and accessed together. For example, the Inertial Measurement Unit (IMU) in modern smart phones collects three-dimensional acceleration, gyroscope, and magnetometer data, for a total of nine variables sampled at each time step. These variables are also likely to be read together, since each on its own is insufficient to characterize the phone's motion.

3. **Low bitwidth**. Any data collected by a sensor will be digitized into an integer by an Analog-to-Digital Converter (ADC). Nearly all ADCs have a precision of 32 bits or fewer [2], and typically 16 or fewer of these bits are useful. For example, even lossless audio codecs store only 16 bits per sample [13, 50]. Even data that is not collected from a sensor can often be stored using six or fewer bits without loss of performance for many tasks [47, 30, 39].

4. **Temporal correlation**. Since the real world usually evolves slowly relative to the sampling rate, successive samples of a time series tend to have similar values. However, when multiple variables are present and samples are stored contiguously, this correlation is often present only with a lag—e.g., with nine IMU variables, every ninth value is similar. Such lag correlations violate the assumptions of most compressors, which treat adjacent bytes as the most likely to be related.

At present, even databases specialized for time series typically use general-purpose compression algorithms that are not optimized for these characteristics. Common codecs chosen include dictionary-based methods such as LZ4 [15] or Snappy [28], floating point compression algorithms [45], or generic integer compression algorithms [7, 5], possibly with some form of invertible preprocessing [7, 45, 4].

These methods can be effective, but have several drawbacks. First, they can be performance bottlenecks. Buevich et al. report that "Enabling compression adds a factor of six slowdown" [10] in their reads, even with the relatively fast

GZIP decoder. Second, as we show experimentally, there is considerable room for improvement in compression ratios. Finally, because of the nature of time series data and workloads, there are a number of requirements for a time series compression algorithm that existing approaches do not satisfy. In particular:

1. **Small block size**. To accelerate queries over arbitrary time intervals, many databases use indexing structures that partition time series into many small segments [4, 10]. A compression algorithm must therefore be effective even on small numbers (<1024) of samples. This also enables offloading compression to clients, which may be low-power sensors with only a few KB of RAM [10].
2. **High decompression speed**. Time series workloads are not only read-heavy [10, 4, 7], but often necessitate materializing data (or downsampled versions thereof) for visualization, clustering, computing correlations, or other operations [10]. At the same time, writing is often append-only [45, 10], so compression need only be fast enough to keep up with the rate of data ingestion.
3. **Lossless**. Given that time series are almost always noisy and often oversampled, it might not seem necessary to compress them losslessly. However, noise and oversampling 1) tend to vary across applications, and 2) can be addressed in an application-specific way as a preprocessing step. Consequently, instead of assuming that some level of downsampling or some particular smoothing will be appropriate for all data, it is better for the compression algorithm to preserve what it is given and leave preprocessing up to the application developer.

Existing methods used in databases almost universally violate Requirement 1. This is because modern general-purpose compressors tend to require large dictionaries of previously seen byte strings to be effective [15, 28, 25, 20]. Examples of such compressors used in databases are LZ4 [15, 52, 59, 61], Snappy [28, 57, 7, 29, 35], and gzip [10, 35], among others. In the time series literature, existing algorithms either violate Requirement 3 through lossy compression [39, 22, 41, 32, 37], or are specialized for floating point data [45] or timestamps [45, 4, 38].

The primary contribution of this work is Sprintz, a compression algorithm for integer time series that offers state-of-the-art compression ratios and speed while also satisfying all of the above requirements. It needs <1KB of memory, can use blocks of data as small as eight samples, and can decompress at up to 3GB/s in a single thread. Sprintz's effectiveness stems from exploiting 1) temporal correlations in each variable's value and variance, and 2) the potential for parallelization across different variables, realized through the use of vector instructions. A key component of how it does this is a novel, vectorized forecasting algorithm for integers. This algorithm can simultaneously train online and generate predictions at close to the speed of `memcpy` and significantly improves compression compared to delta coding.

2. METHOD

In this section we describe how Sprintz works. To do so, we first introduce relevant definitions and an overview of the algorithm. We subsequently discuss each component of the algorithm in detail.

2.1 Definitions

Definition 2.1. ***Sample.*** *A sample is a vector* $\mathbf{x} \in \mathbb{R}^D$. *$D$ is termed the sample's* ***dimensionality****. Each element of the sample is an integer represented using a number of bits w, termed the* ***bitwidth****. The bitwidth w is shared by all elements.*

Definition 2.2. ***Time Series.*** *A time series* $\mathbf{X}$ *of length T is a sequence of T samples,* $\mathbf{x}_1, \ldots, \mathbf{x}_T$. *All samples* $\mathbf{x}_t$ *share the same bitwidth w and dimensionality D. If $D = 1$,* $\mathbf{X}$ *is called* ***univariate****; otherwise it is* ***multivariate****.*

Definition 2.3. ***Rows, Columns.*** *In a database context, we assume that each sample of a time series is one row and each dimension is one column. Because data arrives as samples and memory constraints may limit how many samples can be buffered, we assume that the data is stored in row-major order—i.e., such that each sample is stored contiguously.*

2.2 Overview

Sprintz is a bit packing-based predictive coder. It consists of four components:

1. **Forecasting.** Sprintz employs a forecaster to predict each sample based on previous samples. It encodes the difference between the next sample and the predicted sample, which is typically closer to zero than the next sample itself.
2. **Bit packing.** Sprintz then bit packs the errors as a "payload" and prepends a header with sufficient information to invert the bit packing.
3. **Run-length encoding.** If a block of errors is all zeros, Sprintz waits for a block in which some error is nonzero and then writes out the number of all-zero blocks instead of the (otherwise empty) payload.
4. **Entropy coding.** Sprintz Huffman codes the headers and payloads.

These components are run on blocks of eight samples (motivated in Section 2.4), and can be modified to yield different compression-speed tradeoffs. Concretely, one can 1) skip entropy coding for greater speed and 2) choose between delta coding and our online learning method as forecasting algorithms. Our method is slightly slower but often improves compression.

We chose these steps since they allow for high speed and exploit the characteristics of time series. Forecasting leverages the high correlation of successive samples to reduce the entropy of the data. Run-length encoding allows for extreme compression in the (common) scenario that there is no change in the data—e.g., a user's smartphone may be stationary for many hours while the user is asleep. Our method of bit packing exploits temporal correlation in the variability of the data by using the same bitwidth for points that are within the same block. Huffman coding is not specific to time series but has low memory requirements and improves compression ratios.

An overview of how Sprintz compresses one block of samples is shown in Algorithm 1. In lines 2-5, Sprintz predicts each sample based on the previous sample and any state stored by the forecasting algorithm. For the first sample in a block, the previous sample is the last element of the previous block, or zeros for the initial block. In lines 6-8, Sprintz determines the number of bits required to store the largest error in each column and then bit packs the values in that column using this many bits. (Recall that each column is one variable of the time series). If all columns required 0 bits, Sprintz continues reading in blocks until some error requires >0 bits (lines 11-13). At this point, it writes out a header of all 0s and then the number of all-zero blocks. Finally, it writes out the number of bits required by each column in the latest block as a header, and the bit packed data as a payload. Both header and payload are compressed with Huffman coding.

Algorithm 1 encodeBlock($\{\mathbf{x}_1, \ldots, \mathbf{x}_B\}$, forecaster)

```
1:  Let buff be a temporary buffer
2:  for $i \leftarrow 1, \ldots, B$ do                // For each sample
3:      $\hat{\mathbf{x}}_i \leftarrow$ forecaster.predict($\mathbf{x}_{i-1}$)
4:      $\texttt{err}_i \leftarrow \mathbf{x}_i - \hat{\mathbf{x}}_i$
5:      forecaster.train($\mathbf{x}_{i-1}$, $\mathbf{x}_i$, $\texttt{err}_i$)
6:  for $j \leftarrow 1, \ldots, D$ do                // For each column
7:      $\texttt{nbits}_j \leftarrow \max_i\{$requiredNumBits($\texttt{err}_{ij}$)$\}$
8:      $\texttt{packed}_j \leftarrow$ bitPack($\{x_{1j}, \ldots, \mathbf{x}_{Bj}\}$)
9:  // Run-length encode if all errors are zero
10: if $\texttt{nbits}_j$ == 0, $1 \leq j \leq D$ then
11:     repeat                          // Scan until end of run
12:         Read in another block and run lines 2-8
13:     until $\exists_j[\texttt{nbits}_j \neq 0]$
14:     Write $D$ 0s as headers into buff
15:     Write number of all-zero blocks as payload into buff
16:     Output huffmanCode(buff)
17: Write $\texttt{nbits}_j$, $j = 1, \ldots, D$ as headers into buff
18: Write $\texttt{packed}_j$, $j = 1, \ldots, D$ as payload into buff
19: Output huffmanCode(buff)
```

Sprintz begins decompression (Algorithm 2) by decoding the Huffman-coded bitstream into a header and a payload. Once decoded, these two components are easy to separate since the header is always first and of fixed size. If the header is all 0s, the payload indicates the length of a run of zero errors. In this case, Sprintz runs the predictor until the corresponding number of samples have been predicted. Since the errors are zero, the forecaster's predictions are the true sample values. In the nonzero case, Sprintz unpacks the payload using the number of bits specified for each column by the header.

2.3 Forecasting

Sprintz forecasting can use either delta coding or Fire (Fast Integer REgression), a novel online forecasting algorithm we introduce.

2.3.1 Delta coding

Forecasting with delta coding consists of predicting each sample $\mathbf{x}_i$ to be equal to the previous sample $\mathbf{x}_{i-1}$. This method is stateless given $\mathbf{x}_{i-1}$ and is extremely fast. It is particularly fast when combined with run-length encoding, since it yields a run of zero errors if and only if the data is constant. This means that decompression of runs requires only copying a fixed vector, with no additional forecasting or

Algorithm 2 decodeBlock(bytes, B, D, forecaster)

```
1:  nbits, payload $\leftarrow$ huffmanDecode(bytes, $B$, $D$)
2:  if $\texttt{nbits}_j$ == 0 $\forall j$ then         // Run-length encoded
3:      numblocks $\leftarrow$ readRunLength()
4:      for $i \leftarrow 1, \ldots, (B \cdot \texttt{numblocks})$ do
5:          $\mathbf{x}_i \leftarrow$ forecaster.predict($\mathbf{x}_{i-1}$)
6:          Output $\mathbf{x}_i$
7:  else                                  // Not run-length encoded
8:      for $i \leftarrow 1, \ldots, B$ do
9:          $\hat{\mathbf{x}}_i \leftarrow$ forecaster.predict($\mathbf{x}_{i-1}$)
10:         $\texttt{err}_i \leftarrow$unpackErrorVector($i$, nbits, payload)
11:         $\mathbf{x}_i \leftarrow \texttt{err}_i + \hat{\mathbf{x}}_i$
12:         Output $\mathbf{x}_i$
13:         forecaster.train($\mathbf{x}_{i-1}$, $\mathbf{x}_i$, $\texttt{err}_i$)
```

training. Moreover, when answering queries, one can sometimes avoid decompression entirely—e.g., one can compute the max of all samples in the run by computing the max of only the first value.

2.3.2 FIRE

Forecasting with Fire is slightly more expensive than delta coding but often yields better compression.

The basic idea of Fire is to model each value as a linear combination of a fixed number of previous values and learn the coefficients of this combination. Specifically, we learn an autoregressive model of the form:

$$x_i = ax_{i-1} + bx_{i-2} + \varepsilon_i \tag{1}$$

where x_i denotes the value of some variable at time step i and ε_i is a noise term.

Different values of a and b are suitable for different data characterisics. If $a = 2$, $b = -1$, we obtain double-delta coding, which extrapolates linearly from the previous two points and works well when the time series is smooth. If $a = 1$, $b = 0$, we recover delta coding, which models the data as a random walk. If $a = \frac{1}{2}$, $b = \frac{1}{2}$, we predict each value to be the average of the previous two values, which is optimal if the x_i are i.i.d. Gaussians. In other words, these cases are appropriate for successively noisier data.

The reason Fire is effective is that it learns online what the best coefficients are for each variable. To make prediction and learning as efficient as possible, Fire restricts the coefficients to lie within a useful subspace. Specifically, we exploit the observation that all of the above cases can be written as:

$$x_i = x_{i-1} + \alpha x_{i-1} - \alpha x_{i-2} + \varepsilon_i \tag{2}$$

for $\alpha \in [-\frac{1}{2}, 1]$. Letting $\delta_i \triangleq x_i - x_{i-1}$ and subtracting x_{i-1} from both sides, this is equivalent to

$$\delta_i = \alpha\delta_{i-1} + \varepsilon_i \tag{3}$$

This means that we can capture all of the above cases by predicting the next delta as a rescaled version of the previous delta. This requires only a single addition and multiplication, and reduces the learning problem to that of finding a suitable value for a single parameter.

To train and predict using this model, we use the functions shown in Algorithm 3. First, to initialize a Fire forecaster, one must specify three values: the number of columns D, the learning rate η, and the bitwidth of the integers stored in the columns w. Internally, the forecaster also maintains an accumulator for each column (line 4) and the difference (delta) between the two most recently seen samples (line 5). The accumulator is a scaled version of the current α value with a bitwidth of $2w$. It enables fast updates of α with greater numerical precision than would by possible if modifying α directly.

Algorithm 3 FIRE_Forecaster Class

```
 1: function Init($D$, $\eta$, $w$)
 2:     self.learnShift ← $\lg(\eta)$
 3:     self.bitWidth ← $w$        // 8-bit or 16-bit
 4:     self.accumulators ← zeros($D$)
 5:     self.deltas ← zeros($D$)
 6: function Predict($\mathbf{x}_{i-1}$)
 7:     alphas ← self.accumulators >> self.learnShift
 8:     $\hat{\delta}$ ← (alphas $\odot$ self.deltas) >> self.bitWidth
 9:     return $\mathbf{x}_{i-1} + \hat{\delta}$
10: function Train($\mathbf{x}_{i-1}$, $\mathbf{x}_i$, err$_i$)
11:     gradients ← $-$sign(err$_i$) $\odot$ self.deltas
12:     self.accumulators -= gradients
13:     self.deltas ← $\mathbf{x}_i - \mathbf{x}_{i-1}$
```

To predict, the forecaster first derives the coefficient α for each column based on the accumulator. By right shifting the accumulator $\log_2(\eta)$ bits, the forecaster obtains a learning rate of $2^{-\log_2(\eta)} = \eta$. It then estimates the next deltas as the elementwise product (denoted $\odot$) of these coefficients and the previous deltas. It predicts the next sample to be the previous sample plus these estimated deltas.

Because all values involved are integers, the multiplication is done using twice the bitwidth w of the data type—e.g., using 16 bits for 8 bit data. The product is then right shifted by an amount equal to the bit width. This has the effect of performing a fixed-point multiplication with step size equal to 2^{-w}.

The forecaster trains by performing a gradient update on the L1 loss between the true and predicted samples. I.e., given the loss:

$$\mathcal{L}(x_i, \hat{x}_i) = |x_i - \hat{x}_i| = |x_i - (x_{i-1} + \frac{\alpha}{2^w} \cdot \delta_{i-1})| \quad (4)$$

$$= |\delta_i - \frac{\alpha}{2^w} \cdot \delta_{i-1}| \quad (5)$$

for one column's value $x_i = \mathbf{x}_{ij}$ for some j and coefficient α, the gradient is:

$$\frac{\partial}{\partial \alpha} |\delta_i - \frac{\alpha}{2^w} \cdot \delta_{i-1}| = \begin{cases} -2^{-w}\delta_{i-1} & \mathbf{x}_i > \hat{\mathbf{x}}_i \\ 2^{-w}\delta_{i-1} & \mathbf{x}_i \leq \hat{\mathbf{x}}_i \end{cases} \quad (6)$$

$$= -\operatorname{sign}(\varepsilon) \cdot 2^{-w}\delta_{i-1} \quad (7)$$

$$\propto -\operatorname{sign}(\varepsilon) \cdot \delta_{i-1} \quad (8)$$

where we define $\varepsilon \triangleq \mathbf{x}_i - \hat{\mathbf{x}}_i$ and ignore the 2^{-w} as a constant that can be absorbed into the learning rate. In all experiments reported here, we set the learning rate η to $\frac{1}{2}$. This value is unlikely to be ideal for any particular dataset, but preliminary experiments showed that it consistently worked reasonably well.

In practice, Fire differs slightly from the above pseudocode. First, instead of computing the coefficient for each sample, we compute it once at the start of each block. Second, instead of performing a gradient update after each sample, we average the gradients of all samples in each block and then perform one update. Third, we truncate the coefficient to the four most significant bits. This facilitates learning coefficients of exactly 0 or 2^w; the former reduces variance in the case of noisy data and the latter, which amounts to double delta coding, often predicts the next value exactly for sufficiently smooth data. Finally, we only compute a gradient for every other sample, since this has little or no effect on the accuracy and slightly improves speed.

2.4 Bit Packing

An illustration of Sprintz's bit packing is given in Figure 1. The prediction errors from delta coding or Fire are zigzag encoded [27] and then the minimum number of bits required is computed for each column. Zigzag encoding is an invertible transform that interleaves positive and negative integers such that each integer is represented by twice its absolute value, or twice its absolute value minus one for negative integers. This makes all values nonnegative and maps numbers closer to zero to smaller values.

Given the zigzag encoded errors, the number of bits w' required in each column can be computed as the bitwidth minus the fewest leading zeros in any of that column's errors. E.g., in Figure 1a, the first column's largest encoded value is 16, represented as `00010000`, which has three leading zeros. This means that we require $w' = 8 - 3 = 5$ bits to store the values in this column. One can find this value by ORing all the values in a column together and then using a built-in function such as GCC's `__builtin_clz` to compute the number of leading zeros in a single assembly instruction (c.f. [38]). This optimization motivates our use of zigzag encoding to make all values nonnegative.

Once the number of bits w' required for each column is known, the zigzag-encoded errors can be bit packed. First, Sprintz writes out a header consisting of D unsigned integers, one for each column, storing the bitwidths. Each integer is stored in $\log_2(w)$ bits, where w is the bitwidth of the data. Since there are $w + 1$ possible values of w' (including 0), width $w - 1$ is treated as a width of w by both the encoder and decoder. E.g., 8-bit data that could only be compressed to 7 bits is both stored and decoded with a bitwidth of 8.

After writing the headers, Sprintz takes the appropriate number of low bits from each element and packs them into the payload. When there are few columns, all the bits for a given column are stored contiguously (i.e., column-major order). When there are many columns, the bits for each *sample* are stored contiguously (i.e., row-major order). In the latter case, up to seven bits of padding are added at the end of each row so that all rows begin on a byte boundary. This means that the data for each column begins at a fixed bit offset within each row, facilitating vectorization of the decompressor. The threshold for choosing between the two formats is a sample width of 32 bits.

The reason for this threshold is as follows. Because the block begins in row-major order and we seek to reconstruct it the same way, the row-major bit packing case is the more natural. For small numbers of columns, however, the row padding can significantly reduce the compression ratio. Indeed, for univariate 8-bit data, it makes compression ratios greater than 1 impossible. This gives rise to the column-major case; using a block size of eight samples and column-major order, each column's data always falls on a byte bound-

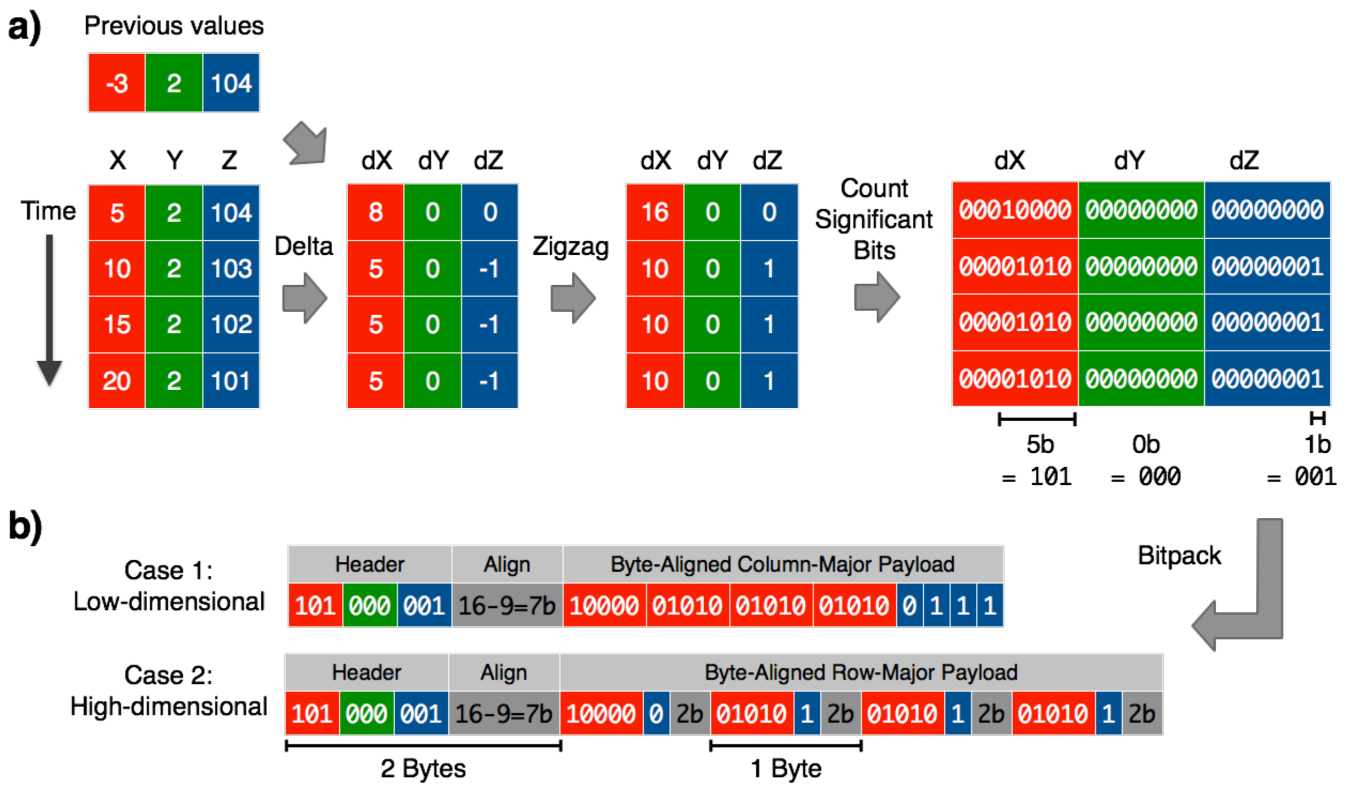

Figure 1: Overview of Sprintz using a delta coding predictor. *a)* Delta coding of each column, followed by zigzag encoding of resulting errors. The maximum number of significant (nonzero) bits is computed for each column. *b)* These numbers of bits are stored in a header, and the original data is stored as a (byte-aligned) payload, with leading zeros removed. When there are few columns, each column's data is stored contiguously. When there are many columns, each row is stored contiguously, possibly with padding to ensure alignment on a byte boundary.

ary without any padding. The downside of this approach is that both encoder and decoder must transpose the block. However, for up to four 8-bit columns or two 16-bit columns, this can be done quickly using SIMD shuffling instructions.[1] This gives rise to the cutoff of 32 bit sample width for choosing between the formats.

As a minor bit packing optimization, one can store the headers for two or more blocks contiguously, so that there is one group of headers followed by one group of payloads. This allows many headers to share one set of padding bits between the headers and payload. Grouping headers does not require buffering more than one block of input, but it does require buffering the appropriate number of blocks of output. In addition to slightly improving compression ratio, it also enables more headers to be unpacked with a given number of vector instructions in the decompressor. Microbenchmarks show up to 10% improvement in decompression speed as the number of blocks in a group grows towards eight. However, we use groups of only two in all reported experiments to ensure that our results tend towards pessimism and are applicable under even the most extreme memory constraints.

[1]For recent processors with AVX-512 instructions, one could double these column counts, but we refrain from assuming that these instructions will be available.

2.5 Entropy Coding

We entropy code the bit packed representation of each block using Huff0, an off-the-shelf Huffman coder [14]. Note that this is faster than Huffman coding the original data or the errors, since the bit packed block is (usually) shorter than the original data. We use Huffman coding instead of Finite-State Entropy [14] or an arithmetic coding scheme these are slower and we never observed a meaningful increase in compression ratio.

2.6 Vectorization

Much of Sprintz's speed comes from vectorization. For headers, the fixed bitwidths for each field and fixed number of fields allows for packing and unpacking with a mix of vectorized byte shuffles, shifts, and masks. For payloads, delta (de)coding, zigzag (de)coding, and Fire all operate on each column independently, and so naturally vectorize. Because the packed data for all rows is the same length and aligned to a byte boundary (in the high-dimensional case), the decoder can compute the bit offset of each column's data one time and then use this information repeatedly to unpack each row. In the low-dimensional case, all packed data fits in a single vector register which can be shuffled/masked appropriately for each possible number of columns. This is possible since there are at most four columns in this case.

On an `x86` machine, bit packing and unpacking can be accelerated with the `pext` and `pdep` instructions, respectively.

3. EXPERIMENTAL RESULTS

To asses SPRINTZ's effectiveness, we compared it to a number of state-of-the art compression algorithms on a large set of publicly available datasets. All of our code and raw results are publicly available on the SPRINTZ website.[2] This website also contains additional experiments, as well as thorough documentation of both our code and experimental setups. All experiments use a single thread on a 2013 Macbook Pro with a 2.6GHz Intel Core i7-4960HQ processor.

All reported timings and throughputs are the best of ten runs. We use the best, rather than average, since this is common practice in performance benchmarking.

3.1 Datasets

- **UCR** [12] — The UCR Time Series Archive is a repository of 85 univariate time series datasets from various domains, commonly used for benchmarking time series algorithms. Because each dataset consists of many (often short) time series, we concatenate all the time series from each dataset to from a single longer time series. This is to allow dictionary-based methods to share information across time series (instead of compressing each in isolation). To mitigate artificial jumps in value from the end of one time series to the start of another, we linearly interpolate five samples between each pair.
- **PAMAP** [48] — The PAMAP dataset consists of inertial motion and heart rate data from wearable sensors on subjects performing everyday actions. It has 31 variables, most of which are acceleromter and gyroscope readings.
- **MSRC-12** [24] — The MSRC-12 dataset consists of 80 variables of (x, y, z, depth) positions of human joints captured by a Microsoft Kinect. The subjects performed various gestures one might perform when interacting with a video game.
- **UCI Gas** [23] — This dataset consists of 18 columns of gas concentration readings and ground truth concentrations during a chemical experiment.
- **AMPDs** [40] — The Almanac of Minutely Power Datasets describes electricity, water, and natural gas consumption recorded once per minute for two years at a single home.

For datasets stored as delimited files, we first parsed the data into a contiguous, numeric array and then dumped the bytes as a binary file. Before obtaining any timing results, we first load each dataset into main memory. For datasets that were not integer-valued, we quantized them such that the largest and smallest values observed corresponded to the largest and smallest values representable with the number of bits tested. Note that this is the worst case scenario for our method since it maximizes the number of bits required to represent the data.

For multivariate datasets, we allowed all methods but our own to operate on the data one variable at a time; i.e., instead of interleaving values for every variable, we store all values for each variable contiguously. This corresponds to allowing them an unlimited buffer size in which to store incoming data before compressing it. We allow these ideal conditions in order to ensure that our results for existing methods err towards optimism and to eliminate buffer size as a lurking variable.

[2]https://github.com/dblalock/sprintz

3.2 Comparison Algorithms

- **SIMD-BP128** [38] — The fastest known method of compressing integers.
- **FastPFOR** [38] — An algorithm similar to SIMD-BP128, but with better compression.
- **Simple8b** [5] — An integer algorithm compression algorithm used by InfluxDB [7].
- **Snappy** [28] — A general-purpose compression algorithm developed by Google and used by InfluxDB, KairosDB, OpenTSDB [57], RocksDB [59], the Hadoop Distributed File System [55] and numerous other projects.
- **Zstd** [16] — Facebook's state-of-the-art general purpose compression algorithm. It is based on LZ77 and entropy codes using a mix of Huffman coding and Finite State Entropy (FSE) [14]. It is available in RocksDB [59].
- **LZ4** [15] — A widely-used general-purpose compression algorithm optimized for speed and based on LZ77. Used by RocksDB and ChronicleDB [52].
- **Zlib** [20] — A popular implementation of the DEFLATE [19] dictionary coder, which also underlies gzip [25].

For Zlib and Zstd, we use a compression level of 9 unless stated otherwise. This level heavily prioritizes compression ratio at the expense of increased compression time. We use it to improve the results for these methods in experiments wherein compression time is not penalized.

We also assess three variations of SPRINTZ, corresponding to different speed/ratio tradeoffs:

1. `SprintzFIRE+Huf`. The full algorithm described in Section 2.
2. `SprintzFIRE`. Like `SprintzFIRE+Huf`, but without Huffman coding.
3. `SprintzDelta`. Like `SprintzFIRE`, but with delta coding instead of FIRE as the forecaster.

3.3 Compression Ratio

In order to rigorously assess the compression performance of both SPRINTZ and existing algorithms, it is desirable to evaluate on a large corpus of time series from heterogeneous domains. Consequently, we use UCR Time Series Archive [12]. This corpus is contains dozens of datasets and is almost universally used for evaluating time series classification and clustering algorithms in the data mining community.

The distributions of compression ratios on these datasets for the above algorithms are shown in in Figure 2. SPRINTZ exhibits consistently strong performance across almost all datasets. High-speed codecs such as Snappy, LZ4, and the integer codecs (FastPFOR, SIMDBP128, Simple8B) hardly compress most datasets at all.

Perhaps counter-intuitively, 8-bit data tends to yield higher compression ratios than 16-bit data. This is a product of the fact that the number of bits that are "predictable" is roughly constant. I.e., suppose that an algorithm can correctly predict the four most significant bits of a given value; this enables a 2:1 compression ratio in the 8-bit case, but

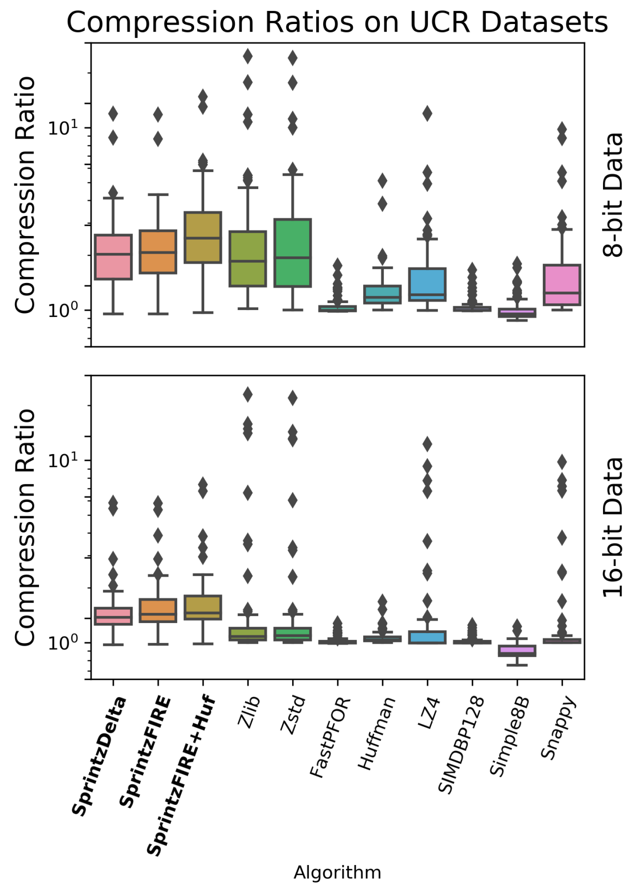

Figure 2: Boxplots of compression performance of different algorithms on the UCR Time Series Archive. Each boxplot captures the distribution of one algorithm across all 85 datasets.

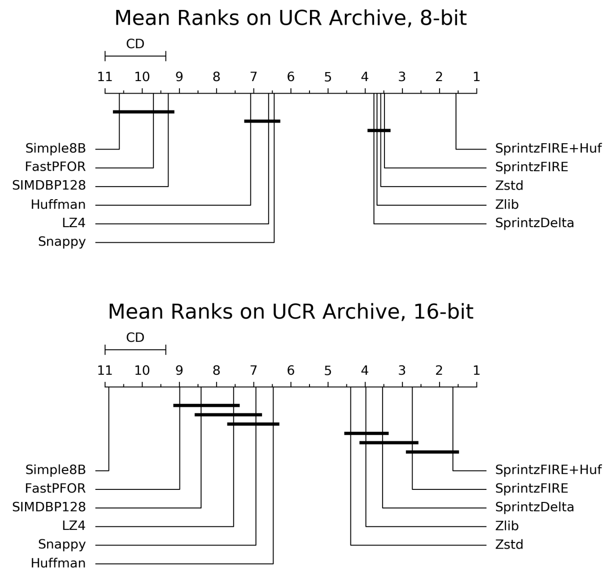

Figure 3: Compression performance of different algorithms on the UCR Time Series Archive. The x-axis is the mean rank of each method, where rank 1 on a given dataset had the highest ratio. Methods joined with a horizontal black line are not statistically significantly different.

only a 16:12 = 4:3 ratio in the 16-bit case. Interestingly, the fact that trailing bits tend to be too noisy to compress also suggests that one could use a lower bitwidth with little loss of information.

To assess SPRINTZ's performance statistically, we use a Nemenyi test [43] as recommended in [18]. This test compares the mean rank of each algorithm across all datasets, where the highest-ratio algorithm is given rank 1, the second-highest rank 2, and so on.

The inuition for why this test is desirable is that it not only accounts for multiple hypothesis testing in making pairwise comparisons, but also prevents a small number of datasets from dominating the results.

The results of the Nemenyi test are visualized in the Critical Difference Diagrams [18] in Figure 3. These diagrams show the mean rank of each algorithm on the x-axis and join methods that are not statistically significantly different with a horizontal line. SPRINTZ on high compression settings is significantly better than any existing algorithm. On lower settings, it is still as effective as the best current methods (Zlib and Zstd).

In addition to this overall comparison, it is important to assess whether FIRE improves performance compared to delta coding. Since this is a single hypothesis with matched pairs, we assess it using a Wilcoxon signed rank test. This yields to p-values of .0094 in the 8-bit case and 4.09e-12 in the 16-bit case. As a more interpretable measure, FIRE obtains better compression on 51 of 85 datasets using 8 bits and 74 of 85 using 16. These results suggest that FIRE is generally beneficial on 8-bit data but even more beneficial on 16-bit data.

To understand why 16-bit data benefits more, we examined datasets where FIRE gives differing benefits in the two cases. The difference most commonly occurs when the data is highly compressible with just delta coding. With 8-bits and $\sim 4\times$ compression, the forecaster's task is effectively to guess whether the next delta is -1, 0, or 1 given a current delta drawn from this same set. The Bayes error rate is high for this problem, and FIRE 's attempts to learn add variance compared to the delta coding approach of always predicting 0. In contrast, with 16 bits, the deltas span many more values and retain some amount smoothness that FIRE can exploit.

3.4 Decompression Speed

To systematically assess the speed of SPRINTZ , we ran it on time series with varying numbers of columns and varying levels of compressibility. Because real datasets have a fixed and limited number of columns, we ran this experiment on synthetic data. Specifically, we generated a synthetic dataset of 100 million random values uniformly distributed across the full range of those possible for the given bitwidth. This data is incompressible and thus provides a worst-case estimate of SPRINTZ's speed (though in practice, we find that the speed is largely consistent across levels of compressibility).

We compressed the data with SPRINTZ set to treat it as if it had 1 through 80 columns. Numbers that do not evenly divide 100 million result in SPRINTZ memcpy-ing the trailing bytes.

While using synthetic data cannot tell us anything about SPRINTZ's compression ratio, it is suitable for throughput measurement. This is because both SPRINTZ's instructions executed and memory access patterns are effectively independent of the data distribution—SPRINTZ's core loop has no conditional branches and SPRINTZ 's memory accesses are always sequential. Moreover, it exhibits throughputs on real data matching or slightly exceeding the numbers below for the corresponding number of columns (c.f. Figure 7).

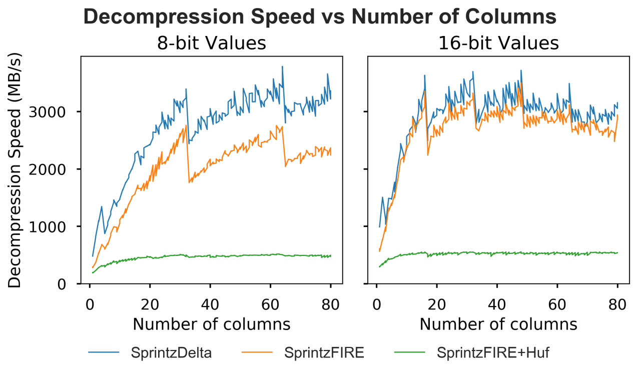

Figure 4: Sprintz becomes faster as the number of columns increases and as the width of each sample approaches multiples of 32B (on a machine with 32B vector registers).

As shown in Figure 4, SPRINTZ becomes faster as the number of columns increases and as the number of columns approaches multiples of 32 for 8-bit data or 16 for 16-bit data. These values correspond to the 32B width of a SIMD register on the tested machine. There is small but consistent overheaded associated with using FIRE over delta coding, but both approaches are extremely fast. Without Huffman coding, SPRINTZ decompresses at multiple GB/s once rows exceed ∼16B. With Huffman coding, the other components of SPRINTZ are no longer the bottleneck and SPRINTZ consistently decompresses at over 500MB/s.

3.5 Compression Speed

Though less important than decompression speed, it is also important that SPRINTZ's compression speed be fast enough to keep up with the rate of data ingestion. We measured SPRINTZ's compression speed using the same methodology as decompression speed. As shown in Figure 5, SPRINTZ compresses 8-bit data at over 200MB/s on the highest-ratio setting and 600MB/s on the fastest setting. These numbers are roughly 50% larger on 16-bit data. Moreover, this implementation is not vectorized, and would be even faster if it were. We refrained from vectorizing in our prototype implementation since 1) 200MB/s is already fast enough to run in real time even if every thread were fed data from its own gigabit network connection, and 2) edge devices often lack vector instructions, so the below speeds are more indicative of the rate at which these devices could compress (if scaled to the appropriate clock frequency).

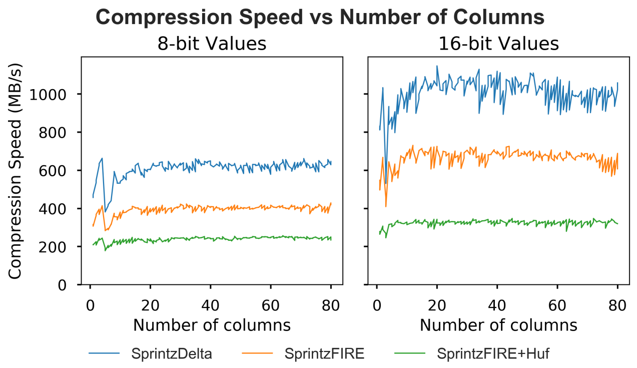

Figure 5: Sprintz compresses at hundred of MB/s even in the slowest case: its highest-ratio setting with incompressible 8-bit data. On lower settings with 16-bit data, it can exceed 1GB/s.

The dip after 4 columns in 8-bit data and 2 columns in 16-bit data corresponds to the switch from column-major bit packing to rowmajor bit packing.

3.6 FIRE Speed

To characterize the speed of the FIRE we repeated the above throughput experiments with both it and two other predictors commonly seen in the literature: delta and double delta coding. As shown in Figure 6, FIRE can compress at up to 5GB/s and decompress at up to 6GB/s. This is nearly the exact same speed as the competing methods and close to the 7.5 GB/s speed of `memcpy` on the tested machine. Note that "compression" and "decompression" here mean converting raw samples to errors and reconstructing samples from sequences of errors, respectively. These operations do not change the data size, but are what run as subroutines in the SPRINTZ compressor and decompressor.

3.7 When to use Sprintz

The above experiments provide a fairly complete characterization of SPRINTZ's speed and a statistically meaningful assessment of its compression ratio in general. However, because one often wants to obtain the best results on a *particular* dataset, it is helpful to know when SPRINTZ is likely to work well or poorly.

Regarding speed, SPRINTZ is most desirable when there are many variables to be compressed. We have found that the speed is largely insensitive to compression ratio, so the results in Sections 3.4 and 3.5) offer a good estimate of the speed one could expect on similar hardware. The exception to this is if the data contains long runs of constants (or constant slopes if using FIRE). In this case, the decompression speed approaches the speed of `memcpy` for `SprintzDelta` or the speed of FIRE for `SprintzFIRE` and `SprintzFIRE+Huf`.

Regarding accuracy, the dominant requirement is that the data must be relatively smooth. When this is not the case, predictive filtering (with only a two-component filter) has little value. Indeed, it can even be counterproductive. Consider the case of data that has an isolated nonzero value every few samples—e.g., the sequence $\{0, -1, 0, 0\}$. When delta coded, this yields $\{0, -1, 1, 0\}$, which requires an extra bit for SPRINTZ bit packing. In general, SPRINTZ has to pay the cost of abrupt changes twice—once when they

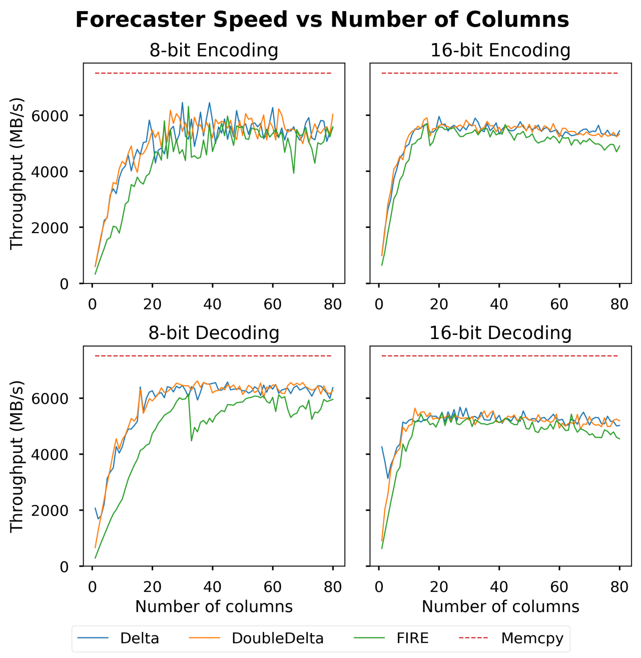

Figure 6: Fire is just as fast as double delta coding and only slightly slower than delta coding. For a moderate number of columns, it runs at 5-6GB/s on a machine with 7.5GB/s `memcpy` speed.

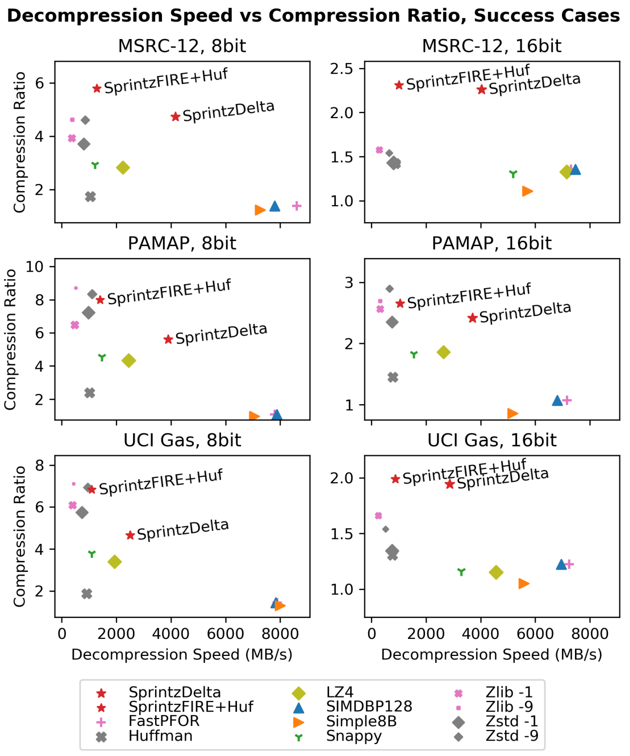

Figure 7: Sprintz achieves excellent compression ratios and speed on relatively smooth time series with many variables.

happen, and once when they "revert" to the previous level.

Another specific case in which SPRINTZ is undesirable is when the the data distribution tends to switch between discrete states. For example, in electricity consumption data, an appliance tends to use little or no electricity when it is off, and a relatively constant amount when it is on. Switches between these states are expensive for SPRINTZ, and predictive filtering offers little benefit on sequences of samples that are already almost constant. SPRINTZ can still achieve reasonably good compression in this situation, but dictionary-based compressors will likely perform better. This is because they suffer no penalty from state changes and runs of constants are their best-case input in terms of both ratio and speed. Their ratio benefits because they can often run-length encode the number of repeated values, and their speed benefits because they can decode runs at memory speed by `memcpy`-ing the repeated values.

As an illustration of when SPRINTZ is and is not preferable, we ran it and the comparison algorithms on several real-world datasets with differing characteristics. In Figure 7, we use the MSRC-12, PAMAP and UCI Gas datasets. These datasets contain relatively smooth time series and have 80, 31, and 18 variables, respectively. SPRINTZ achieves an excellent ratio-speed tradeoff on all three datasets, and the highest compression of any method *even on its lowest-compression setting* on the MSRC-12 dataset.

In contrast, SPRINTZ performs poorly on the AMPD Gas and AMPD Water datasets (Figure 8). These datasets chronicle the natural gas and water consumption of a house over a year, and often switch between discrete states and/or have isolated nonzero values. They also have only 3 and 2 variables, respectively. SPRINTZ achieves more than 10× compression, but dictionary-based methods such as Zstd and LZ4 achieve even greater compression while also decompressing faster.

3.8 Generalizing to Floats

While floating point values are not the focus of our paper, it is possible to apply SPRINTZ to floats by first quantizing the floating point data. The downside of doing this is that, because floating point numbers are not uniformly distributed along the real line, such quantization is lossy. To assess the degree of loss, we carried out an experiment to measure the error induced when quantizing real data. Note that this experiment does not assess whether SPRINTZ is the *best* means of compressing floats—it merely suggests that using integer compressors like SPRINTZ as lossy floating point compressors is reasonable and could be a fruitful avenue for future work.

We assessed the magnitude of typical quantization errors by quantizing the UCR time series datasets. Specifically, we linearly offset and rescaled the time series in each dataset such that the minimum and maximum values in any time series correspond to $(0, 255)$ for 8-bit quantization or $(0, 65535)$ for 16-bit quantization. We then obtained the quantized data by applying the floor function to this linearly transformed data.

To measure the error this introduced, we then inverted the linear transformation and computed the mean squared error between the original and this "reconstructed" data. The resulting error values for each dataset, normalized by the dataset's variance, are shown in Figure 9. These normalized values can be understood as signal-to-noise ratio measure-

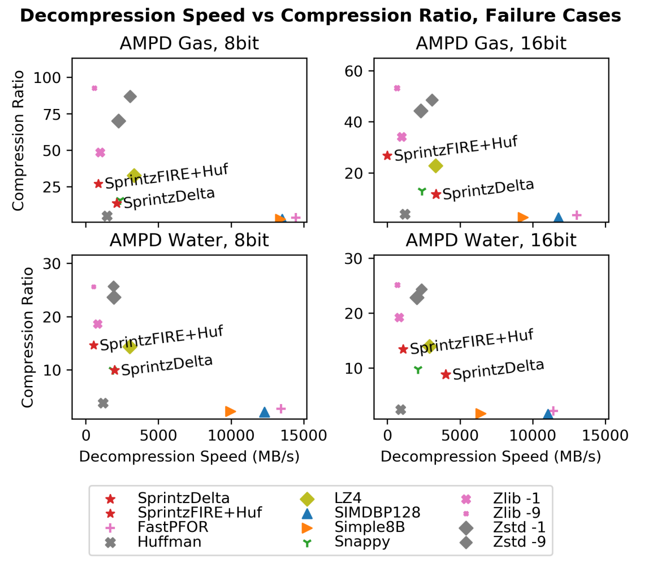

Figure 8: Sprintz is less effective than other methods when data is non-smooth and has few variables.

ments, where the noise is the quantization error. As the figure illustrates, the quantization error is orders of magnitude smaller than the variance for nearly all datasets, and never worse than 10× smaller even for 8-bit quantization.

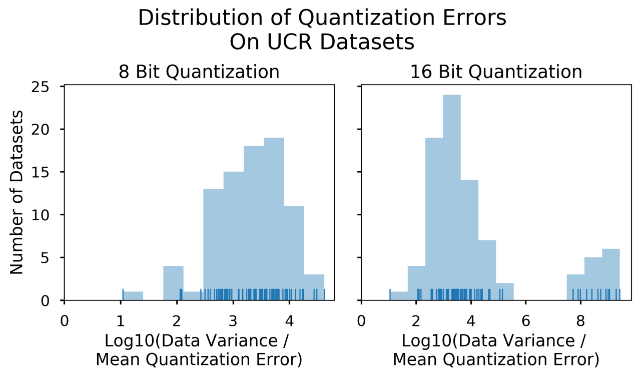

Figure 9: Quantizing floating-point time series introduces error that is orders of magnitude smaller than the variance of the data. Even with 8 bits, quantization introduces less than 1% error on 84 of 85 datasets.

This of course does not indicate that all time series can be safely quantized. Two counterexamples of which we are aware are 1) timestamps where microsecond or nanosecond resolution matters, and 2) GPS coordinates, where small decimal places may correspond to many meters. However, the above results suggest that quantization is a suitable means of applying Sprintz to floating-point data in many applications. This is bolstered by previous work showing that quantization even to a mere six bits [47] rarely harms classification accuracy, and quantizing to two bits sometimes improves it [53].

4. RELATED WORK

4.1 Compression of time series

Most work on compressing time series has focused on lossy techniques. In the data mining literature, Symbolic Aggregate Approximation (SAX) [39] and its variations [54, 11] dominate. These approaches preserve enough information about time series to support specific data mining algorithms (e.g. [46, 34]), but are extremely lossy; a hundred-sample time series might be compressed into one or two bytes, depending on the exact SAX parameters.

Other lossy approaches include Adaptive Piecewise Constant Approximation [32], Piecewise Aggregate Approximation [31], and numerous other methods [33, 37, 22] that approximate time series as sequences of low-order polynomials.

For audio time series specifically, there are a large number of lossy codecs [42, 50, 1, 60], as well as a small number of lossless [13, 6] codecs. In principle, some of these could be applied to non-audio time series. However, modern codecs make such strong assumptions about the possible numbers of channels, sampling rates, bit depths, or other characteristics that it is infeasible to use them on non-audio time series.

Many fewer algorithms exist for lossless time series compression. For floating-point time series, the only algorithm of which we are aware is that of the Gorilla database [45]. This method XORs each value with the previous value to obtain a diff, and then bit packs the diffs. In contrast to our approach, it assumes that time series are univariate and have 64-bit floating-point elements.

For lossless compression of integer time series (including timestamps), existing approaches include directly applying general-purpose compressors [10, 57, 52, 29, 61], (double) delta encoding and then applying an integer compressor [7, 45], or predictive coding and byte-packing [36]. These approaches can work well, but tend to offer both less compression and less speed than Sprintz.

4.2 Compression of integers

The fastest methods of compressing integers are generally based on bit packing—i.e., using at most b bits to represent values in $\{0, 2^b - 1\}$, and storing these bits contiguously [56, 63, 38]. Since b is determined by the largest value that must be encoded, naively applying this method yields limited compression. To improve it, one can encode fixed-size blocks of data at a time, so that b can be set based on the largest values in a block instead of the whole dataset [51, 63, 38]. A further improvement is to ignore the largest few values when setting b and store their omitted bits separately [63, 38].

Sprintz bit packing differs significantly from existing methods in that it compresses much smaller blocks of samples. This reduces its throughput as compared to, e.g., [38], but significantly improves compression ratios, since large values only increase b locally. It also allows significantly lowers the memory requirements.

A common [13, 50] alternative to bit packing is Golomb coding [26], or its special case Rice coding [49]. The idea is to assume that the values follow a geometric distribution, often with a rate constant fit to the data.

Both bit packing and Golomb coding are bit-based methods in that they do not guarantee that encoded values will be aligned on byte boundaries. When this is undesirable, one can employ byte-based methods such as 4-Wise Null Suppression [51], LEB128 [17], or Varint-G8IU [58]. These methods reduce the number of bytes used to store each sample by encoding in a few bits how many bytes are necessary to represent its value, and then encoding only that many bytes. Some, such as Simple8B [5] and SIMD-GroupSimple [62], allow fractional bytes to be stored while preserving byte alignment for groups of samples.

4.3 General-purpose compression

While Sprintz is not intended to be a general-purpose compression algorithm, a reasonable alternative to using a specialized method would be to apply a general-purpose compression algorithm, possibly after delta coding or other preprocessing. Thanks largely to the development of Asymmetric Numeral Systems (ANS) [21] for entropy coding, general purpose compressors have advanced greatly in recent years. In particular, Zstd [16], Brotli [3], LZ4 [15] and others have attained speed-compression tradeoffs significantly better than traditional methods such as GZIP [25], LZO [44], etc. As described in Section 1, however, these methods have much higher memory requirements that Sprintz.

4.4 Predictive Filtering

For numeric data such as time series, there are four types of predictive coding commonly in use: predictive filtering [9], delta coding [38, 56], double-delta coding [7, 45], and XOR-based encoding [45]. In predictive filtering, each prediction is a linear combination of a fixed number of recent samples. This can be understood as an Autoregressive model or the application of a Finite Impulse Response (FIR) filter. When the filter is learned online, this is termed “adaptive filtering.”

Delta coding is a special case of predictive filtering where the prediction is always the previous value. Double-delta coding, also called delta-delta coding or delta-of-deltas coding, consists of applying delta coding twice in succession. XOR-based encoding is similar to delta coding, but replaces subtraction of the previous value with the XOR operation. This modification is often desirable for floating-point data [45].

Fire can be understood as a special case of adaptive filtering. While adaptive filtering is a well-studied mathematical problem in the signal processing literature, we are unaware of a practical algorithm that attains speed within an order of magnitude of that of Fire .

5. CONCLUSION

We introduce Sprintz, a compression algorithm for multivariate integer time series that achieves state-of-the-art compression ratios and speeds of up to 3GB/s in a single thread. Sprintz compression imposes extremely small memory requirements, making it suitable for both compression at the network edge and server-side compression in the presence of massive numbers of streams.

6. REFERENCES

[1] Information technology – generic coding of moving pictures and associated audio information – part 7: Advanced audio coding (aac), 2006. `https://www.iso.org/standard/43345.html`.

[2] Digi-key electronics, 2017. `https://www.digikey.com/products/en/integrated-circuits-ics/data-acquisition-analog-to-digital-converters-adc/700?k=adc\&k=\&pkeyword=adc\&pv1989=0`.

[3] J. Alakuijala and Z. Szabadka. Brotli compressed data format. Technical report, 2016.

[4] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *FAST*, pages 39–52, 2016.

[5] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[6] Apple. Apple lossless audio codec, 2011. `https://github.com/macosforge/alac`.

[7] S. Beckett. Influxdb, 2017. `https://influxdata.com`.

[8] D. W. Blalock and J. V. Guttag. Extract: Strong examples from weakly-labeled sensor data. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 799–804. IEEE, 2016.

[9] T. Boutell. Png (portable network graphics) specification version 1.0. 1997.

[10] M. Buevich, A. Wright, R. Sargent, and A. Rowe. Respawn: A distributed multi-resolution time-series datastore. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 288–297. IEEE, 2013.

[11] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. isax 2.0: Indexing and mining one billion time series. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 58–67. IEEE, 2010.

[12] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista. The ucr time series classification archive, July 2015. `www.cs.ucr.edu/~eamonn/time_series_data/`.

[13] J. Coalson. Flac-free lossless audio codec, 2008. `http://flac.sourceforge.net`.

[14] Y. Collet. Finite state entropy. `https://github.com/Cyan4973/FiniteStateEntropy`.

[15] Y. Collet. Lz4–extremely fast compression, 2017. `https://github.com/Cyan4973/lz4`.

[16] Y. Collet. Zstandard - fast real-time compression algorithm, 2017. `https://facebook.github.io/zstd/`.

[17] D. D. I. F. Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

[18] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30, 2006.

[19] L. P. Deutsch. Deflate compressed data format specification version 1.3. 1996.

[20] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[21] J. Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[22] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[23] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

[24] S. Fothergill, H. M. Mentis, P. Kohli, and S. Nowozin.

Instructing people for training gestural interactive systems. In J. A. Konstan, E. H. Chi, and K. Höök, editors, *CHI*, pages 1737–1746. ACM, 2012.

[25] J.-L. Gailly and M. Adler. The gzip home page, 2003. `https://www.gzip.org/`.

[26] S. Golomb. Run-length encodings. *IEEE transactions on information theory*, 12(3):399–401, 1966.

[27] Google. Protocol buffers encoding, 2001. `https://developers.google.com/protocol-buffers/docs/encoding#types`.

[28] S. Gunderson. Snappy: A fast compressor/decompressor, 2015. `https://code.google.com/p/snappy`.

[29] B. Hawkins. Kairos db: Fast time series database on cassandra. `https://github.com/kairosdb/kairosdb`.

[30] B. Hu, T. Rakthanmanon, Y. Hao, S. Evans, S. Lonardi, and E. Keogh. Discovering the intrinsic cardinality and dimensionality of time series using mdl. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 1086–1091. IEEE, 2011.

[31] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286, 2001.

[32] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Sigmod Record*, 30(2):151–162, 2001.

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

[34] E. Keogh, J. Lin, and A. Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *Data mining, fifth IEEE international conference on*, pages 8–pp. Ieee, 2005.

[35] J. Kestelyn. Introducing parquet: Efficient columnar storage for apache hadoop. *Cloudera Blog*, 3, 2013.

[36] E. Lazin. Akumuli time-series database. `https://akumuli.org`.

[37] D. Lemire. A better alternative to piecewise linear time series segmentation. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 545–550. SIAM, 2007.

[38] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[39] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11. ACM, 2003.

[40] S. Makonin, B. Ellert, I. V. Bajic, and F. Popowich. Electricity, water, and natural gas consumption of a residential house in Canada from 2012 to 2014. *Scientific Data*, 3(160037):1–12, 2016.

[41] S.-G. Miaou and H.-L. Yen. Multichannel ecg compression using multichannel adaptive vector quantization. *IEEE transactions on biomedical engineering*, 48(10):1203–1207, 2001.

[42] J. Moffitt. Ogg vorbisopen, free audioset your media free. *Linux journal*, 2001(81es):9, 2001.

[43] P. Nemenyi. Distribution-free multiple comparisons. In *Biometrics*, volume 18, page 263. INTERNATIONAL BIOMETRIC SOC 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210, 1962.

[44] M. Oberhumer. Lzo-a real-time data compression library. *http://www.oberhumer.com/opensource/lzo/*, 2008.

[45] 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*, 8(12):1816–1827, 2015.

[46] T. Rakthanmanon and E. Keogh. Fast shapelets: A scalable algorithm for discovering time series shapelets. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 668–676. SIAM, 2013.

[47] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 547–556. IEEE, 2011.

[48] A. Reiss and D. Stricker. Towards global aerobic activity monitoring. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, page 12. ACM, 2011.

[49] R. F. Rice. Some practical universal noiseless coding techniques, part 3, module psl14, k+. 1991.

[50] T. Robinson. Shorten: Simple lossless and near-lossless waveform compression, 1994.

[51] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, pages 34–40. ACM, 2010.

[52] M. Seidemann and B. Seeger. Chronicledb: A high-performance event store. In *EDBT*, pages 144–155, 2017.

[53] P. Senin and S. Malinchik. Sax-vsm: Interpretable time series classification using sax and vector space model. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1175–1180. IEEE, 2013.

[54] J. Shieh and E. Keogh. isax: disk-aware mining and indexing of massive time series datasets. *Data Mining and Knowledge Discovery*, 19(1):24–57, 2009.

[55] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[56] H. Siedelmann, A. Wender, and M. Fuchs. High speed lossless image compression. In *German Conference on Pattern Recognition*, pages 343–355. Springer, 2015.

[57] B. Sigoure. Opentsdb: The distributed, scalable time series database. *Proc. OSCON*, 11, 2010.

[58] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of

posting lists. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 317–326. ACM, 2011.

[59] F. D. E. Team. Rocksdb: A persistent key-value store for fast storage environments. `http://rocksdb.org`.

[60] J.-M. Valin, K. Vos, and T. Terriberry. Definition of the opus audio codec. Technical report, 2012.

[61] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2014.

[62] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J.-Y. Nie, H. Yan, and J.-R. Wen. A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems (TOIS)*, 33(3):15, 2015.

[63] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.