Hierarchical Residual Encoding for Multiresolution Time Series Compression

BRUNO BARBARIOLI, The University of Chicago, USA
GABRIEL MERSY, The University of Chicago, USA
STAVROS SINTOS, University of Illinois Chicago, USA
SANJAY KRISHNAN, The University of Chicago, USA

Data compression is a key technique for reducing the cost of data transfer from storage to compute nodes. Increasingly, modern data scales necessitate lossy compression techniques, where exactness is sacrificed for a smaller compressed representation. One challenge in lossy compression is that different applications may have different accuracy demands. Today's compression techniques struggle in this setting either forcing the user to compress at the strictest accuracy demand, or to re-encode the data at multiple resolutions. This paper proposes a simple, but effective multiresolution compression algorithm for time series data, where a single encoding can effectively be decompressed at multiple output resolutions. There are a number of benefits over current state-of-the-art techniques for time series compression. (1) The storage footprint of this encoding is smaller than re-encoding the data at multiple resolutions. (2) Similarly, the compression latency is generally smaller than re-encoding at multiple resolutions. (3) Finally, the decompression latency of our encoding is significantly faster than single encodings at the strictest accuracy demand.

CCS Concepts: • **Information systems → Data compression**; **Temporal data**; **Sensor networks**; *Data streaming*.

Additional Key Words and Phrases: Storage, Temporal databases, Compression

ACM Reference Format:
Bruno Barbarioli, Gabriel Mersy, Stavros Sintos, and Sanjay Krishnan. 2023. Hierarchical Residual Encoding for Multiresolution Time Series Compression. *Proc. ACM Manag. Data* 1, 1, Article 99 (May 2023), 26 pages. https://doi.org/10.1145/3588953

1 INTRODUCTION

In today's data analytics infrastructure, it is common for data storage to be separated from computational resources [2]. For example, very large datasets can be stored in a block storage system like Amazon S3 and only transferred to compute nodes for analysis. Similarly, in edge computing, data might reside on edge nodes for privacy or cost reasons and only be transferred to a central location when needed [33, 47]. In such on-demand retrieval architectures, the time needed to transfer data between storage and computation nodes is an important bottleneck, and data compression is one of the main techniques for controlling the cost of data movement.

Traditionally, data compression approaches are divided into two categories: lossless and lossy. In lossless compression, the data are compressed and decompressed without loss of any information.

Authors' addresses: Bruno Barbarioli, barbarioli@uchicago.edu, The University of Chicago, Chicago, Illinois, USA; Gabriel Mersy, gmersy@uchicago.edu, The University of Chicago, Chicago, Illinois, USA; Stavros Sintos, stavros@uic.edu, University of Illinois Chicago, Chicago, Illinois, USA; Sanjay Krishnan, skr@uchicago.edu, The University of Chicago, Chicago, Illinois, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/5-ART99 $15.00
https://doi.org/10.1145/3588953

Examples of such approaches include dictionary coding for string compression [3, 16], GZip/BZip for byte-sequence compression [39], and turbo-coding for integer compression [38]. In contrast, lossy compression allows for minor errors in the reconstruction that would not affect a downstream application. By nature, lossy compression is mostly aimed at high-dimensional quantitative data. Examples include JPEG for images [43], H.264 for video [45], and a variety of techniques for scientific data [7, 28, 41]. Lossy compression techniques sacrifice accuracy (the degree of errors in the reconstruction) for storage size (the size of the compressed data).

To cope with ever growing datasets, lossy compression has been increasingly adopted in edge-computing and sensing systems [33–35, 40, 47]. Data compression can be pushed towards the point of data collection to save on downstream data transfer, storage, and computation. For example, a visual dashboard monitoring a sensor only needs the data to be accurate up to the screen pixel resolutions. On the other hand, machine learning models that consume the data are often robust to small amounts of imprecision in the input features. While such "compression pushdown" can be extremely effective, it is most useful when there is only a single downstream application consuming the data.

Multiple downstream applications can have differing accuracy demands (e.g., a visual dashboard requires a `1e-3` maximum error in all values, but the anomaly detection framework only requires `1e-1` precision). In such cases, the available pushdown strategies essentially are: (Strict Encoding) encode the data once at the strictest accuracy demand, (Multiple Encoding) re-encode the data at all of the different accuracy demands, and (Lazy Encoding) first encode and store the data at the strictest accuracy demand, then at retrieval time decode the data and re-encode it at the error target. The *strict encoding* strategy has the obvious drawback that it forces every application to pay the same data transfer cost as the one with the strictest accuracy requirement. On the other hand, the *multiple encoding* strategy allows different applications to selectively retrieve data encoded at their particular accuracy demands. However, the multiple encoding strategy has a steep cost in terms of compression latency or compression throughput (i.e., linear in the number of applications), and local storage (i.e., stores one encoding per application). Finally, lazy encoding does not use as much storage as the multiple encoding strategy and has a lower transfer cost than strict encoding. However, the decompression process is significantly slower, since it requires re-encoding the data.

This paper navigates this impasse on time series data and studies efficient methods for supporting multiple downstream consumers of lossy (or lossless) data. Ideally, it should be possible to store a *single encoding that can be selectively decompressed at all of the different resolutions* and thus mitigate the downsides of both strategies described above. We call this problem the *multiresolution compression problem*, where the objective is to construct a single encoding of a time series that can be decompressed at different cell-level error tolerances (hereafter called L_∞ errors). While related problems have been proposed in a number of adjacent areas such as in reduced-precision ML [44] and approximate query processing [5, 29]; to the best of our knowledge, multiresolution compression has not been extensively evaluated in the data compression literature. This problem setting subtly changes the typical metrics of interest for data compression. An effective multiresolution compression algorithm: (Compression ratio) should require significantly less storage than a separate encoding at each error tolerance; (Compression latency) should be significantly faster to construct the encoding than a separate encoding at each error tolerance; (Decompression latency) should be faster to decompress data for higher error tolerances. This means that a multiresolution compression algorithm might not be the most effective at any single error tolerance, but in aggregate across multiple tolerances, it outperforms the strict and multiple encoding strategies.

This paper proposes an effective multiresolution compression algorithm, called Hierarchical Residual Encoding (HIRE), for univariate and multivariate time series data. HIRE assumes a mini-batch data acquisition model where data are streamed to the system in small blocks, and these blocks

represent contiguous time-segments of collected data. HIRE constructs a synopsis data structure through a recursion of partitioning and residualizing steps. The partitioning step approximates a time series with a piecewise approximation, and the residualizing step calculates a signal that represents the approximation error. This error signal can further be approximated at increasingly finer granularities. We find that these residual signals are often highly compressible, since many values may lie under the error threshold of the strictest application. To retrieve data at a particular error threshold, we apply a simple summation of the preceding layers. This result is not surprising as such residualizing operations are effective in time series forecasting [11], and in differential equation solvers [22].

2 BACKGROUND

First, we will motivate the general problem statement and give context to our contributions in multiresolution compression. Through this paper, we will consider the following running example.

Example 2.1 (Edge Data Storage and Retrieval). Data transfer over a network is one of the most expensive (in terms of cost and energy) tasks in any distributed sensing application. Many recent sensing architectures argue for lazy data retrieval [33], where data are persisted on the edge for some period of time and only centralized if/when needed. Let's consider a simplified two server version of this architecture. Consider a sensor deployed at a climate research observatory that collects a time series of numerical data. The data are compressed online at an "edge server" during data collection (located at the observatory), and stored locally for a maximum of 10 days. The compressed versions can be retrieved from the edge server, transferred to a remote server (e.g., at a research university), and decompressed at the remote server.

2.1 Time Series Compression Basics

Let us consider a time series with observations:

$$X = [x_0, x_1, \ldots, x_{T-1}].$$

For now, let us assume that each x_i is a single floating point number (i.e., a univariate time series). We will relax this assumption later, but it will be easier to understand the baselines in the univariate case. A compression algorithm consists of an encoder and decoder pair

$$C_X = \mathsf{enc}(X) \qquad X' = \mathsf{dec}(C_X)$$

that produces a compressed representation of X called C_X and reconstructs an estimate of the original time series X' from C_X. Without loss of generality, we can consider C_X to be a vector as well. There are several important metrics of interest that describe the performance of such a compression algorithm:

- **Compression Ratio.** Let $H(\cdot)$ denote the size in bits of a vector. The compression ratio is defined as $\frac{H(C_X)}{H(X)}$. A lower compression ratio indicates better performance in terms of storage. *In our running example, the compression ratio directly affects how much data are transferred from the edge to the remote server.*
- **Compression Latency.** The compression latency of an algorithm is the time needed to produce C_X from X. *In our running example, the compression latency affects the data collection throughput of the edge server.*
- **Decompression Latency.** The decompression latency of an algorithm is the time needed to produce X' from C_X. *In our running example, the decompression latency affects how much extra time beyond data transfer is spent on the remote server.*

- **Reconstruction Error.** The difference between X and X' is called the reconstruction error, for some measure of dissimilarity. *In our running example, the reconstruction error measures how different the data processed on the remote server is compared to the edge server.*

While there are many such compression algorithms, we primarily focus on the ones with deterministic L_∞ reconstruction error guarantees. That is, we bound the maximum allowable error of the reconstructed time series $X' = \text{dec}(\text{enc}(X))$ with respect to the original time series X by specifying an error threshold parameter ε. The L_∞ reconstruction error $||X - X'||_\infty$ is defined as the maximum absolute disagreement between X and X' at any time-step i:

$$L_\infty = ||X - X'||_\infty = \max_i |x_i - x'_i|$$

We then enforce an error guarantee by ensuring that the L_∞ error is within the pre-specified threshold ε, which means $||X - X'||_\infty \leq \varepsilon$. For example, ε may represent the maximum error that the observatory is willing to tolerate in the reconstructed time series to avoid a negative impact on subsequent weather forecasting tasks. We choose such guarantees because they are the strictest and most compatible with a wide variety of applications. The main trade-off in most techniques is when ε is increased (i.e., more error), the compression ratio decreases.

2.2 Compression with Bounded L_∞ Error

There is an extensive body of literature on time series compression techniques (refer to survey [25]). However, not all compression algorithms can provide L_∞ error guarantees. For example, a spectral approach like PCA, or an FFT, can only control the average error but not the worst-case error for any given observation. For techniques that do provide L_∞ error guarantees, they generally follow the same design pattern composing three main components: (1) Quantization, (2) Temporal Decorrelation, and (3) Byte Encoding.

2.2.1 Quantization. The most basic technique for compressing numerical data with an error guarantee is quantization. Quantization is the process of rounding a floating point number to nearest valid value of fixed precision. Even simple data quantization can be very effective and is employed in Amazon Redshift [19]. Proceeding to the technical formulation of quantization, let us suppose that x^-, x^+ are the minimum and maximum of X respectively. Define R_ϵ to be the *relative* L_∞ error, i.e., a desired error tolerance relative to the range of X:

$$R_\epsilon = \frac{\varepsilon}{x^+ - x^-}$$

A uniform quantization scheme cuts the range $[x^-, x^+]$ into $\frac{1}{R_\epsilon}$ equal-sized buckets [1]. To encode a dataset using this scheme, one sets each numerical value to its resident integer bucket (note that the floor function discretizes the formerly continuous input):

$$\text{enc}(x_i) = \left\lfloor \frac{(x_i - x^-)}{(x^+ - x^-)} \cdot \frac{1}{R_\epsilon} \right\rfloor \tag{1}$$

We examine the quantization formula, we notice that the main term $\lfloor \frac{(x_i - x^-)}{(x^+ - x^-)} \cdot \frac{1}{R_\epsilon} \rfloor$ is integer-valued and ranges from 0 to $\lceil \frac{1}{R_\epsilon} \rceil$. In its most basic implementation, we can assign a fixed-length binary code to each of the integer values. This would result in storage size of $\log_2 \lceil \frac{1}{R_\epsilon} \rceil$ per values.

To decode, one simply reverses the transformation:

$$x'_i = \text{dec}(\text{enc}(x_i)) = \text{enc}(x_i) \cdot R_\epsilon \cdot (x^+ - x^-) + x^- \tag{2}$$

[1] In the degenerate case of $\epsilon = 0$ one just keeps each element of X in its own bucket

As long as we also store x^+, x^-, R_ϵ, we can translate the stored integer into its corresponding floating point quantum. Unfortunately, regardless of whether quantization is applied to a column sampled from a normal distribution or to a time series with high autocorrelation, it yields the same mapping.

2.2.2 Temporal Decorrelation. For this reason, either before or after quantization most time series compression algorithms attempt to factor out all of the "predictable" terms, thereby only leaving uncorrelated errors to be compressed. The simplest approach to accomplish this is delta-encoding, which transforms a time series so that every value is represented as a successive difference from the previous value:

$$x_i^\delta = x_i - x_{i-1}$$

This transformation is completely reversible with a left-to-right cumulative sum, so no information is lost. Since time series often have highly similar values along the time axis, the range of delta values is smaller in magnitude and variation than the original values. Thus, the deltas can often be effectively stored with reduced precision.

We can think of delta-encoding as a simple form of "predictive" compression. The previous value x_{i-1} can be interpreted as a simple model that predicts the value of its neighboring element [4]. The same trick would work for any function f of the previous j lagged elements:

$$x_i^\delta = x_i - f(x_{i-1}, x_{i-2}, \ldots, x_{i-j})$$

Given the dependence of the current value on the previous (lagged) values, the structure of delta encoding is suitable for autoregressive models of any flavor. It is worth noting that the better the predictive model, the more skewed the X_δ values will be towards zero. If the effective domain of the numbers can be greatly reduced, then fewer bits can be used to represent X_δ.

In general, any time series modeling technique can be used to decorrelate the data. For example, there are a number of piecewise approximations for time series that exploit trend structure in a typical time series. Piecewise approaches decompose a time series into segments and use the segments to approximate the time series such as in Piecewise Aggregate Approximation (PAA) [26] and Piecewise Linear Approximation (PLA) [24]. Like delta encoding, we can think of piecewise approximation as a simple model that predicts the next value. One only needs to store the model and the compressed residual. If this model is accurate, the residual error is likely very sparse and highly compressible.

2.2.3 Byte-Encoding. Finally, after quantization and decorrelation, general-purpose compression algorithms can be used to simply translate the remaining data into a series of bytes and reduce redundancy. Most of these approaches are based on run-length encoding or the LZ77 algorithm that look for byte-level repetitions within sliding windows of data [39]. Generally speaking, byte-oriented techniques are lossless—meaning that the L_∞ error is always 0—so they can provide a trivial error guarantee. In numerical data, the obvious limitation is that while two floating point numbers may be close to each other numerically, there might not be a very strong similarity between their binary representations leading to poor compression with an LZ77 technique. This is why quantization and decorrelation are generally applied first to exploit the numerical structure.

2.3 The Multiresolution Problem

For any combination of quantization and decorrelation described above, the encoded representation C_X is tied to a specific target error. Let us consider how this can cause an unnecessary performance bottleneck in our running example.

Example 2.2 (Two Applications With Different Error Tolerances). Consider two remote applications consuming the climate data collected at the observatory. The first application is a nightly report

that is generated over all of the collected data. This report requires that every value processed is no more than $\varepsilon = 1e-4$ of the true value. The second application is a minute-by-minute anomaly detector to detect whether the observatory has abnormal readings. This application is far more tolerant to error and requires that each value processed is only within $\varepsilon = 1e-1$ of the true value.

If we use the state of the art, there are three clear solutions—all of which have significant drawbacks.

- **Compress at the Strictest Tolerance.** The most obvious approach would be to compress the data at the strictest error tolerance. Unfortunately, this would mean that every retrieval would pay a data transfer cost of the strictest threshold. *In our running example, the frequent anomaly detection tests would have to repeatedly transfer data encoded with a target of* $\varepsilon = 1e-4$ *due to the relatively infrequent nightly reporting.*
- **Compress at All Tolerances.** Another approach would be to encode the data at all relevant error tolerances. While this approach allows each application to only transfer data encoded at an appropriate error tolerance, it shifts the burden towards encoding. Each additional error target would reduce the effective throughput available for data collection, since we are compressing the data twice. *In our running example, we would cut our ingestion capacity by roughly a factor of two.*
- **Lazy Re-Encoding.** A hybrid approach would be to first encode the data at the strictest error tolerance, then at retrieval time decode the data and re-encode it at the error target of each retrieving application. The core challenge is that decompression is often significantly slower than compression in many popular algorithms, and this hybrid approach incurs these costs at the edge. *In our running example, the frequent anomaly detection tests would trigger expensive re-encoding processes that would burden the edge server.*

These drawbacks suggest the need for a new type of time series compression algorithm aimed at supporting multiple downstream applications. Ideally, there should be a single encoding that can be selectively decompressed at different target resolutions.

Definition 2.3 (Multiresolution Compression). A multiresolution compression algorithm produces a single encoding C_X that can be selectively decomposed into sub-encodings $C_X^{\epsilon_1}, \ldots, C_X^{\epsilon_l}$ with corresponding error thresholds $\epsilon_1, \ldots, \epsilon_l$:

$$C_X = C_X^{\epsilon_1} \bigoplus C_X^{\epsilon_2} \bigoplus \ldots \bigoplus C_X^{\epsilon_l}$$

where $\bigoplus$ denotes some combination operation of the sub-encodings.

Decomposable encodings allow one to selectively transfer data for any target resolution. The multiple encoding strategy described above can be thought of as a trivial case where $\bigoplus$ is simply the concatenation operation over independent encodings of the data. The key issue with this strategy is that no work is shared between any of the encodings, and effectively sharing work will be the main premise of this paper. We will show that an additive decomposition, where $\bigoplus$ is a linear combination operation, is a simple but effective solution. Consequently, the goal of this paper is to investigate such algorithms, understand how to evaluate them, and how to optimize for different performance objectives in the multiresolution setting.

3 MATHEMATICAL INTUITION

We will provide some basic technical intuition on how such an algorithm can be constructed.

3.1 Residualization

From the previous section, let X be a time series and X' be an approximation of it (e.g., a decoding of a lossy encoding). The *residual* series is also a time series and is defined as:

$$R = X - X'$$

which is the difference between the original series and its reconstruction. It clearly follows from this definition that if one knows the approximation and the residual, one can fully reconstruct the original series $X' + R = X$. Such a decomposition of terms is called an additive decomposition and is well-studied in modeling time-dependent phenomena outside of data compression [11]. For example, it is common to decompose time series into a trend component (which represents the general trend of the series) and noise (which represents variation along the trend). The trend component would be our X' and the noise would be our R.

Additive decompositions are recursive in nature, since the residual R is itself another time series and can be further decomposed. Now, let us suppose that we have an approximation to R denoted R'. We can similarly define a residual series $S = R - R'$. Through substitution, we can see that our additive decomposition now looks like this:

$$X = X' + R' + S$$

This process can be repeated, further and further decomposing the residual, which yields a natural recurrence equation:

$$\begin{aligned} R_0 &= X \\ A_i &= \text{approx}(R_i) \\ R_i &= R_{i-1} - A_{i-1} \end{aligned} \tag{3}$$

where $\text{approx}(\cdot)$ is some approximation function. We can clearly see from this equation that for k such recursions:

$$X = \sum_{i=0}^{k-1} A_i \quad + \quad R_k$$

We leverage this basic intuition to construct an effective multiresolution compression algorithm. Written in another way the equation above is simply $\sum_{i=0}^{k-1} A_i \approx X$. We need to construct a sequence of successive approximations that reduces the size of the residual signal until the residual is less than our desired error threshold.

3.2 Relevance to Time Series Compression

The next section will show how to leverage a series of successive refinements of the residual series to represent the original series. At each step, we compress the residual vector from the previous step, progressively increasing the fidelity of the compression. As more of the variation in the data is explained by each subsequent step, the residual vector becomes smaller in magnitude and sparser.

Let us consider a concrete example. In Figure 1(A), we have plotted an example time series. One can coarsely approximate this time series with a piecewise constant approximation of 3 segments (Figure 1(B)). Between this approximation and the original series, there is a residual (Figure 1(C)). This residual can be captured by another piecewise constant approximation with 6 segments (Figure 1(D)). As long as each subsequent approximation includes additional information (i.e., increasing the number of segments), the remaining residual series reduces in magnitude. A summation of these approximations gets closer to the original value (Figure 1(E)). While the intuition is simple, realizing this idea turns out to be more difficult. The next section describes how we can enforce an error guarantee in this process of successive refinement and how to implement such an approach efficiently.

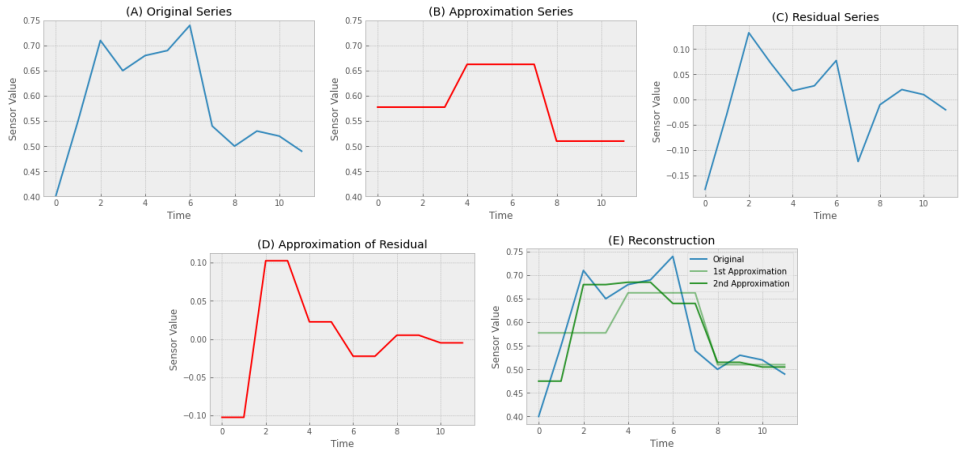


Fig. 1. (A) Illustrates an example time series, (B) illustrates a coarse approximation of the time series, (C) illustrates the residual series, (D) illustrates a finer approximation of the residual series, and (E) shows how the summations of the residuals lead to progressively better approximations

3.3 Novelty

Our new compression method for recursively approximating a time series has some similarities with other classical mathematical methods for approximating functions such as Fourier and Taylor. For instance, a Taylor series can in fact provide an upper bound error for each distinct partial sum, and the error decreases as the size of the partial sum grows. However, there are key differences in our formulation because of how we refine our approximation and measure the resulting bounds. In particular, in series approximation, error is only used to decide when to terminate the series (i.e., a stopping condition), which is entirely different from how we recursively approximate the error values themselves. We leverage the intuition that residuals often contain a sparser signal than the function values. By recursively applying approximation to the residuals at increasingly granularity, the resulting residual gets sparser (more blocks under the error threshold). This leads to more long runs of 0 values and thus better compressibility. By setting the midrank function as the pool function, we show that the L_∞ error is strictly non-increasing with respect to the level of the hierarchy, a similar theoretical guarantee to the Taylor series.

The novelty of our submission is therefore: i) a new problem definition highlighting the emerging need for multiresolution (de)-compression systems, ii) hierarchical recursive approximation of residual vectors in the domain of time series compression with L_∞ guarantees—which has not been proposed before, iii) the use of pool function properties along with vector theory to propose linear time compression and decompression algorithms computing the errors on the fly, and iv) practical implementation of the theoretical algorithms making use of vectorization and parallel computing.

4 HIERARCHICAL RESIDUAL ENCODING

Based on our intuition from the previous section, we design an algorithm that constructs a progressively refined set of residual signals. This algorithm is called Hierarchical Residual Encoding (HIRE). We describe our method considering a univariate time series. We can also handle multivariate time series by running independent encodings.

4.1 Algorithm Basics

Before we introduce the algorithm, it would be informative to define a few general building blocks. As before, let $X = [x_0, x_1, \ldots, x_{T-1}]$ be a univariate time series represented as a vector of data, where $x_i \in \mathbb{R}$ and $X \in \mathbb{R}^T$. We further assume that the user provides us with an error ε^* or the strictest error threshold that the encoding must guarantee.

4.1.1 Quantized Pooling. HIRE relies on a piecewise approximation of each residual series: over disjoint windows, the value of the window is approximated by a single scalar aggregate. To use machine learning terminology, this operation is called (temporal) pooling. Pooling reduces the dimensionality of a data series along the time axis over a series of fixed-size windows. The pooling operation is defined by two parameters, an aggregation function f and a time series X. In particular let $p_f(X) : \mathbb{R}^n \rightarrow \mathbb{R}$, $p_f(X) = f(x_0, \ldots, x_{n-1})$, for any positive integer n [2]. In other words, a pool function provides a concise estimate of a given time window with a single value. The choice of pooling function is a hyper-parameter and different pooling functions have unique properties. We can think of them as different ways of approximating the values in a time window. For a window $X = [x_0, \ldots, x_{n-1}]$, we define:

- Mean: $f(X) = \frac{1}{n} \sum_i x_i$. The mean value is a natural pooling function that minimizes the squared error with respect to the window.
- Midrank: $f(X) = \frac{1}{2} (\max_i\{x_i\} - \min_i\{x_i\})$. We can show that midrank is the optimum pool function to minimize the L_∞ error of the residual vector in each node of the hierarchy, as described in the next subsection.
- Median: $f(X)$ returns the median value of vector X.
- Random: $f(X) = x_i$ with probability $1/n$. We can show that a random pooling function is robust to seasonal variation within windows of a time series.

To efficiently store the results of a pooling operation, we further quantize the aggregate value to a particular error threshold (using the process described in Section 2). Quantization ensures that all of the pool values are integers, so more efficient compression methods can be used to compress/decompress them. During the decompression we derive the pool values P and transform them to the corresponding real values before we take their sum. Care must be taken in how these values are quantized, and Section 2.2.1 describes how to translate ε^* into a quantization threshold.

4.1.2 Spline interpolation. Pooling is generally a lossy operation for $n > 1$ and is only lossless for $n = 1$ (i.e., window size of 1). Thus, inverting this operation will only give us an approximation of the original series. To do so, we require some function that estimates the original time series from the pooled values. We define a spline function $s_w(p_f(X)) \approx X$, where $w = |X|$. The user can choose the spline function that they prefer. Our default option is a simple interpolation function that duplicates the value $p_f(X)$, w times. In particular, $s_w(p_f(X)) = [p_f(X), \ldots \times w]$. This duplication-oriented spline can be thought of as constructing a step function interval with length w and value $p_f(X)$. Specifically, the default spline computes a step function where each interval captures a pooled value that covers w time steps. In doing so, we map from a lower dimensional summary to an approximation of the input series of a certain coarseness. With an increasingly smaller w, the step approximation of X improves proportionally to w itself.

4.1.3 Residual Vectors. The spline function returns an approximation of the original time series. Let $R_X = X - s_w(p_f(X))$ be the residual vector representing the error of the spline function with respect to the original vector. The key insight of our work is that residual vectors are generally more compressible than the original time series. It is clearly true that $X = R_X + s_w(p_f(X))$, and we

[2]We slightly abuse the notation and use X as either the input time series with size T, or any time series with size n.

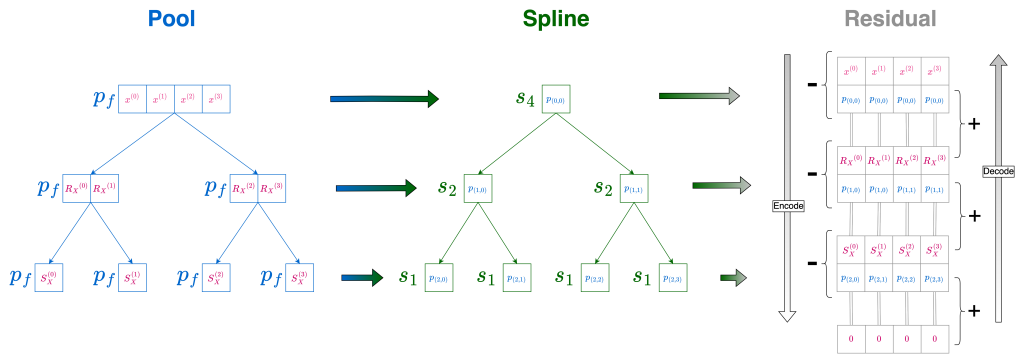


Fig. 2. The main algorithmic workflow. Compression flows from left to right and top to bottom just like reading text. Decompression flows from bottom to top and inverts residualization with a summation.

can plug in this approximation into our recurrence equation of the previous section (Equation 3), where $\text{approx}(R_i) = s_w(p_f(R_i))$.

4.2 Algorithm Description

The main idea is to run the procedure described above recursively in a hierarchical manner following the recurrence described in Equation 3. For simplicity we consider that size of the time series is a power of 2, so $T = 2^k$. First, we check if the L_∞ norm of vector X is at most ε^*. If yes, then we can stop the recursion setting the pool value to 0. Otherwise, we compute the pool value $p_f(X)$ over the entire vector X. As stated, we also apply quantization on $p_f(X)$. To simplify the notation and the proposed procedure we use the same notation for the pool values and the quantized pool values [3]. Then we set $R_X = X - s_w(p_f(X))$ as described above. In the next level of the recursion, we call the same procedure twice: the first time with input $X \leftarrow R_X[0, \ldots, T/2 - 1]$ and the second time with input $X \leftarrow R_X[T/2, \ldots, T - 1]$. Let $P_i = [p_{(i,0)}, \ldots, p_{(i,2^i-1)}]$, for $i = 0, \ldots, k$, be the vector of the pool values in the i-th level of the hierarchy sorted from left to right. Let $P = \bigcup_{i=0,\ldots,k} P_i$. We stop the recursion when the error is at most ε^* or after having $|X| = 1$. In other words, as we traverse down the hierarchy, there is a successively more accurate approximation of the residual at each level. After running this algorithm, the quantized pool values can be stored with any byte-encoding format.

We describe the pseudocode in Algorithm 1 (for simplicity, we describe the pseudocode considering that P is a global set of variables over the recursion) and provide a visual in Figure 2.

Intuitively, we can think of a binary tree structure where the original time series lies in the root and its residual vector is split into two equal length sub-vectors creating two children in the tree structure. In each node of the tree, we store the corresponding singular pool value. Let $\mathcal{T}$ be the tree structure representing the hierarchical compression. For a node u in $\mathcal{T}$, let p_u be the pool value in this node. For example, if u is the j-th node in the i-th level, we have $p_u = p_{(i,j)}$. Let also R_u be the residual vector that is found from our algorithm at node u.

We note that it is not necessary to start the hierarchical compression from the zero level—considering the entire time series X. Instead, we can start the hierarchical compression from any level Γ, splitting the original time series into 2^Γ parts, $X[0, \ldots, T/2^\Gamma - 1], X[T/2^\Gamma, \ldots, 2 \cdot T/2^\Gamma - 1], \ldots, X[(2^\Gamma - 1) \cdot T/2^\Gamma, \ldots, T - 1]$ and run the Hierarchical algorithm for each of them

[3]Otherwise, we can consider that any pool function applies quantization before it returns the final result.

Algorithm 1: HIERARCHICAL

```
Input  : X, i, j, ε*
Output : P
if ||X||_∞ ≤ ε* then
    p_(i,j) = 0;
    return;
T = |X|;
p_(i,j) = p_f(X);
R_X = X − s_T(p_f(X));
if T > 1 then
    HIERARCHICAL(R_X[0, ..., T/2 − 1], i + 1, 2 · j, ε*);
    HIERARCHICAL(R_X[T/2, ..., T − 1], i + 1, 2 · j + 1, ε*);
```

independently. In fact, we mostly run the hierarchical compression for the last 10 or 12 levels, i.e., we set $\Gamma = k - 10$ or $\Gamma = k - 12$ in our experiments.

Finally, we note that it is not necessary to keep the pool values in different variables $p_{(i,j)}$ or p_u storing the indexes (i, j) or u. We only use this notation to make the description of the algorithm easier. Algorithm 1 can put all pool values in a single table P following the ordering: $P[h_1] = p_{(i_1,j_1)}, P[h_2] = p_{(i_2,j_2)}$ for $h_1 < h_2$ if and only if $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$. When the decompression algorithm needs to access $p_{(i,j)}$, it corresponds to the element $P[2^i + j - 1]$, so we have direct access in $O(1)$ time. Furthermore, as we will see in the next subsection, sometimes we might need to have access to the pool value of the parent or the left (right) child of a node u. If u corresponds to the j-th node in the i-th level then $p_{(i-1,j \mod 2)}, p_{(i+1,2\cdot j)}, p_{(i+1,2\cdot j+1)}$ is the pool value of the parent, left child, and right child, respectively.

4.3 Decompression

Before we describe how we can decompress P to derive an approximation with L_∞ reconstruction of X (lossy) or the exact X (lossless) we show an interesting property of the residual vectors over the nodes of $\mathcal{T}$. Let u_i be the j-th node of the i-th level in the hierarchy. Let $\bar{X} = X[j \cdot w, \ldots, (j+1) \cdot w - 1]$, where $w = T/2^i$, be the corresponding part of the original time-series in node u_i. Let $u_0 \rightarrow \ldots \rightarrow u_i$ be the path from the root of $\mathcal{T}$ to u_i. From their definitions, $R_{u_i} = \bar{X} - \sum_{\ell \le i} s_w(p_{u_\ell})$. Hence, it follows that $\max_h\{|R_{u_i}[h]|\} = ||\bar{X} - \sum_{\ell \le i} s_w(p_{u_\ell})||_\infty$, i.e., the L_∞ error of the sum of the spline vectors from the root node to the current node with respect to the original $\bar{X}$ vector, is the L_∞ error of the residual vector R_{u_i}. We extend the previous observation to each level of $\mathcal{T}$. For each level $i \le k$ let E_i be the maximum L_∞ error of all residual vectors found at level i. Let $E = \bigcup_{i \le k} E_i$. Our system compresses both P, E using any known compression method. If M_1, M_2 are the compression methods used for P, E, respectively, the overall compression ratio of our method is $\frac{H(M_1(P)) + H(M_2(E))}{H(X)}$, where again $H(\cdot)$ denotes the size in bits.

Multiresolution Decompression. Let us see how the decompression algorithm works with our running example. Recall that we have two different applications that require error thresholds of $\varepsilon = 1e-4$ and $\varepsilon = 1e-1$ after retrieving data from an edge server. First, using HIRE, we construct a compressed residual encoding with $\varepsilon^* = 1e-4$. Along with every retrieval request, a desired error threshold is sent ε. The edge server first finds the maximum error E_m such that $E_m \le \varepsilon$. Then we transfer all compressed pool values P_i for $i \le m$, from the edge server to the remote machine that made the request. Finally, the decompression procedure runs on the remote machine, finding

$X' = \sum_{i \leq m} s_{T/2^i}(P_i)$, where the function $s_{T/2^i}(P_i)$ takes as input the vector P_i and returns another vector repeating every value of P_i, $T/2^i$ times. Using the observations above, we have the guarantee that $||X - X'||_\infty \leq \varepsilon$.

5 OPTIMIZATIONS

In this section we describe multiple algorithmic and implementation optimizations that improve the running time of our algorithms. In the previous section $k = \log T$, i.e., the maximum height of the tree in the compression algorithm. Recall that the execution of our compression algorithm finishes when the L_∞ error is at most ε^*, hence the final depth of the recursion (or height of the tree) might be less than $\log T$. We slightly abuse the notation and we use k to denote the actual number of levels of recursion (or the actual height of the tree) after finishing the compression algorithm.

5.1 Algorithmic Optimizations

5.1.1 Compression. We first note that Algorithm 1 runs in $O(kT)$ time. There are at most k levels of recursion and in each level we construct the residual vectors of all nodes. Hence, in each level we spend $O(T)$ time. We show how we can run Algorithm 1 in only linear, $O(T)$ time.

The main observation is that we do not need to construct the residual vectors explicitly. These vectors are helping to run the recursion algorithm and at the same time show what is the maximum L_∞ error in each node. We claim that we can still run the same algorithm without constructing the residual vectors. Let u be the j-th node in the i-th level and let $v_0 \rightarrow \ldots \rightarrow v_{i-1} = v_i = u$ be the path of the nodes from the root to node u. Let also $\bar{X} = X[j \cdot w, \ldots, (j+1) \cdot w - 1]$, for $w = T/2^i$, be the subset of time series X that corresponds to the part of node u. It is straightforward to see that: $p_{(i,j)} = p_u = f\left(\bar{X} - \sum_{\ell=0}^{i-1} s_w(p_{v_\ell})\right)$. This is a very important observation because it shows that by using only the original time series along with the previously computed pool values, we can get the new pool value that we need to store and compress.

Next, we show an efficient way to compute $f\left(\bar{X} - \sum_{\ell=0}^{i-1} s_w(p_{v_\ell})\right)$. Of course, the actual algorithm depends on the function f. Hence, we check all of the main functions that we used. This method can be extended to a large family of functions. For all functions we are using, we have the next observation: it holds that either $f\left(\bar{X} - \sum_{\ell=0}^{i-1} s_w(p_{v_\ell})\right) = f(\bar{X}) - \sum_{\ell=0}^{i-1} p_{v_\ell}$ or $f\left(\bar{X} - \sum_{\ell=0}^{i-1} s_w(p_{v_\ell})\right) = f(\bar{X})$. It is easy to maintain (when needed) the term $\sum_{\ell=0}^{i-1} p_{v_\ell}$ during the execution of the compression algorithm. Let S_v be the sum of all pool values from the root to the node v. We can update $S_{child(v)} = S_v + p_{child(v)}$ in constant time. Hence, we only focus on how to compute $f(\bar{X})$ efficiently. In particular, we aim to construct a data structure $\mathcal{D}$ in $O(T)$ time such that given a query range $[a, b]$, compute $f(X[a, \ldots, b])$ in $O(1)$ time.

We start considering the midrank function f. We pre-process X and we build a range MAX/MIN data structure $\mathcal{D}$ using the LCA technique [17]. It is known that $\mathcal{D}$ can be computed in $O(T)$ time, it has $O(T)$ space, and can answer a range MAX or MIN query in $O(1)$ time. Hence, we can run Algorithm 1 in $O(T)$ time.

Next, we consider the mean function f. Again, we need a data structure to find the mean of X in a query range. We compute and store the prefix sums of X: $\mathcal{D}[h] = \sum_{z=0}^{h} X[z]$. Overall, we construct a data structure $\mathcal{D}$ in $O(T)$ time such that given a range $[a, b]$ we return $f(X[a, b]) = \frac{\mathcal{D}[b] - \mathcal{D}[a-1]}{b-a+1}$ in $O(1)$ time. Again, we run Algorithm 1 for the mean function in $O(T)$ time.

It is straightforward how to get a random item in $X[a, \ldots, b]$ efficiently for the random function f. We just get a random number $t \in [a, b]$ and we return $X[t]$. So we can also run Algorithm 1 in $O(T)$ time for the random function.

Next, we note that it is also straightforward to run the compression algorithm for the pool functions with quantization. The only difference is that when we find the value of $f\left(\bar{X} - \sum_{\ell=0}^{i-1} s_w(p_{v_\ell})\right)$

we compute in $O(1)$ time the integer bucket it belongs to. Unfortunately, to the best of our knowledge, there is not any known data structure to compute the median function in a query range in $O(1)$ time, using at most linear pre-processsing time.

Interestingly, the data structure we used for the midrank pool function is actually needed in all pool functions to measure the L_∞ error in each node/level in the hierarchy. Previously, having a residual vector we could find the L_∞ error by checking the absolute values of its elements. In the optimum algorithm we do not construct explicitly the residual vectors, so we cannot do the same procedure. Instead, we argue as before. From the definitions it follows that $R_u = \bar{X} - \sum_{\ell=0}^{i} s_w(p_{v_\ell})$. Hence, $e_u = ||R_u||_\infty = \max_h\{|\bar{X}[h] - S_u|\} = \max\{|\max_h\{\bar{X}[h]\} - S_u|, |\min_h\{\bar{X}[h]\} - S_u|\}$. As we explained above, we can calculate S_u (from the parent node) in $O(1)$ time. Using the same data structure $\mathcal{D}$ we used for the midrank function [17], we can find the MAX and the MIN values of the original time series X in a query range in $O(1)$ time. Hence, we can compute the L_∞ error in a node u of $\mathcal{T}$ in $O(1)$ time. The overall running time of Algorithm 1 for every pool function f we use (except the median), including the error calculation, is $O(T)$.

5.1.2 Decompression. The decompression algorithm we described in the previous section runs in $O(kT)$ time, since we take the sum of the residual vectors to retrieve the original time series or an approximation of it. We show how we can execute the decompression algorithm in only linear $O(T)$ time. More specifically, the algorithm can be executed in time linear to the number of nodes in the hierarchy that we need to retrieve to run the decompression procedure. While the algorithm is more tedious to describe than the compression algorithm, it is independent of the pool function f that we used in the compression phase.

For simplicity, assume that the decompression method needs to take the sum over all the pool values (using spline interpolation) up to level $L \leq k$. The algorithm can be extended in case that we need to take the sum of vectors (starting) from different levels. We describe an algorithm doing it without computing the spline interpolation vectors explicitly. The main idea is the following: we run a sweep-line algorithm starting from left to right maintaining the total sum of all the corresponding pool values in the hierarchy. For example, imagine that we are currently considering an index $h \leq T$ in a node u at level L. Let S_h be the total sum of the pool values from the root to u. We observe that the decompressed value is $X'[h] = S_h$. Hence, the goal is to maintain the correct values S_h over all indexes h from left to right. In order to derive the value S_h from S_{h-1}, we subtract the pool values that correspond to the nodes that index $h-1$ belongs to and add the pool values that correspond to the nodes that index h belongs to. The pseudocode can be seen in Algorithm 2. The while condition on line 5 checks if there is a change in the pool values from index $h-1$ to h at level i. Then variable j stores the node at level i that index h belongs to. We update the value S_h subtracting the pool value $p_{(i,j-1)}$ (i.e., pool value in $(j-1)$-th node at level i) and adding the pool value $p_{(i,j)}$ (i.e., pool value in j-th node at level i). We recall that the notation $p_{(i,j)}$ is only used to simplify the description of the algorithm. We can always access the pool value in the j-th node at the i-th level, taking the value $P[2^i + j - 1]$ in constant time. Algorithm 2 visits each compressed value in P two times, one to add it to the sum and one to subtract it from the sum. Hence, the running time is $O(T)$.

5.2 Implementation Optimizations

Moving now from theory to practice, here we highlight a few best practices for implementing a scalable encoder-decoder pair—that is, a pair with a low latency and runtime memory footprint.

5.2.1 Compression. The first step in optimizing the encoder (Algorithm 1) is to convert the recursive formulation to an iterative formulation—thereby avoiding the allocation of unnecessary stack space. At each level, we apply the pool and spline operations in succession. A naive implementation of

Algorithm 2: FastDecompression

Input : P
Output : X'

$S_{-1} = 0$;
for $h = 0$ *to* $T - 1$ **do**
  $i = L$;
  $S_h = S_{h-1}$;
  while $h \mod \frac{T}{2^i} == 0$ *AND* $i \geq 0$ **do**
    $j = \frac{h}{T/2^i}$;
    $S_h = S_h + p_{(i,j)} - p_{(i,j-1)}$;
    $i = i - 1$;
  $X'[h] = S_h$
return X';

pooling might map the function of choice to each individual window. However, there is a large amount of extra work in that we might perform an unnecessary memory allocation operation to rearrange the array into windows of size n and then redundantly compute w small sums—only to eventually divide each sum by the exact same value. Instead, we can compute a prefix sum over the entire input with a single optimized function call and then in constant time deposit each pooled value into a pre-allocated array. This technique exploits the vectorization and instruction-level parallelism present in conventional superscalar processors.

Now suppose that we have a multivariate time series $Y \in \mathbb{R}^{T \times p}$ that consists of p univariate columns. We can apply the encoding algorithm to each column in parallel and thus achieve a latency speed up. Further suppose that we have b blocks in each univariate column. We can also apply the encoding algorithm to individual blocks—or groups of blocks, for that matter—which introduces a more granular form of parallelism.

5.2.2 Decompression. As an alternative approach to the linear time technique expressed in Algorithm 2, we can also in principle exploit threaded parallelism within the decoder. The HIRE decoding algorithm must calculate a linear combination over a large number of recomputed spline arrays. This summation does not need to be done in a sequential order due to the fact that the pooled values are already in memory. Visually, we can partition the hierarchy along the depth axis of the tree such that each individual spline reconstruction and summation operation (Figure 2) is assigned to a single thread. As a concrete example, if ten residual subtraction operations are performed during encoding, ten addition operations must be performed during decoding. If we have two cores available, then we can assign five operation pairs (reconstruction and summation) to one thread and the remaining five to the other thread. We then perform a meta summation over the vectors returned by each thread which therefore yields the reconstructed time series X'.

5.3 Extensions

We extend HIRE to work with other error functions, and we show how we can split a time series to optimize the hierarchical compression algorithm. Furthermore, we show how our technique can be optimized to handle smoother reconstruction errors. We only show the high level ideas and skip the low level details.

5.3.1 L_p error. Our compression method can actually bound the error of any L_p norm, extending the previous results for the L_∞ norm. For simplicity we focus on L_1, and L_2 norms. The main observation is that the residual vector R_X explicitly computed by Algorithm 1, or implicitly computed by the optimized algorithm, contains the absolute differences from the original vector. Hence, we argue that the L_p error of a node u is the L_p norm of vector R_X in node u. More specifically, in line 1 of Algorithm 1, we check whether $||X||_\infty \leq \varepsilon^*$. For any L_p norm we can use the condition $||X||_p \leq \varepsilon^*$ to check the L_p error in the current node of the hierarchical compression. If we want to measure the overall L_p error of the compressed time series we take the sum of the errors over the nodes within the same level. In particular, if $w_1, \ldots, w_n$ are the L_p errors of n nodes at level h, then the overall L_p error at level h is defined as $\left(\sum_{i \leq n} w_i^p\right)^{1/p}$. We can show that the optimum pool function to minimize the L_1 error is the median function, while the optimum pool function for the L_2 error is the mean function. The linear time optimized compression algorithm can be applied for the L_2 error. The L_2 norm can be computed without constructing the residual vector explicitly by calculating prefix sums for both the values of the original time series and their squared values. The mean function can also be computed in constant time for each node as we described in Section 5.1.1. For a general L_p error function, the compression algorithm runs in $O(kT)$ time, as we had with the L_∞ error (recall that k is the number of levels in the hierarchical compression). Finally, we note that the linear time optimized decompression algorithm is independent of the error function.

5.3.2 Optimum splitting. We also explore different ways to split a time series during the compression algorithm. For example, using the midrank function for bounding the L_∞ error, the best option is to split the time series such that the maximum pairwise absolute difference of elements in each sub-time series is minimized. Specifically, given a time series X with n elements we want to find the element j such that $\max\{\max_{i \leq j} X[i] - \min_{\ell \leq j} X[\ell], \max_{i > j} X[i] - \min_{\ell > j} X[\ell]\}$ is minimized. In order not to define a different optimization problem for every error and pool function, we consider the following splitting function that can split a time series in any scenario: split at the element j such that the maximum squared error of the two sub-time series is minimized. In particular, the maximum squared error of splitting a time series X on j is defined as $\max\{\sum_{i \leq j}(X[i] - \hat{X}_{\leq j})^2, \sum_{\ell > j}(X[\ell] - \hat{X}_{>j})^2\}$, where $\hat{X}_{\leq j}$ is the mean of $X[1], \ldots, X[j]$ and $\hat{X}_{>j}$ is the mean of $X[j+1], \ldots, X[n]$. Intuitively, the maximum squared error captures how homogeneous each sub-time series is. Ideally, we would like to create homogeneous time series so that by applying a pool function we minimize its error. It is known that both functions, maximum difference and max of squared errors, are increasing with respect to the number of elements in a time series. Hence, we run a standard binary search on the elements of X and for each element j we evaluate the splitting function on the ranges $[1, .., j]$ and $[j+1, .., n]$ corresponding to the left and right side of the split, respectively. By constructing a data structure in linear time during the pre-processing phase, we can evaluate the squared error of a query range in $O(1)$ time. The binary search on a residual vector of size n takes $O(\log n)$ steps. Given an input time series of size T, the overall compression algorithm takes $O(T + 2^k \log T) = O(T \log T)$ time. Finally, we note that while non-trivial splitting functions can help to reduce the error faster, it should explicitly store the size of each sub-time series in the hierarchical encoding.

5.3.3 Smoother reconstruction errors. One drawback of our solution is that we do not have any control of the errors E in each level of the hierarchy. Using the midrank function we know that these errors are non-increasing, however there are only k of them and they might not be smooth. For example, as described earlier, given a reconstruction error ε, our method first finds the largest error E_m such that $E_m \leq \varepsilon$. Recall that E_m is the maximum error at the m-th level of $\mathcal{T}$. Then by transferring the pool values stored in nodes with depth at most ε, we make sure that the

reconstruction error is E_m. However, E_m might be much larger than ε. Here, we describe a few ways to decompress in a larger variety of error thresholds without changing the main ideas of our compression method. In particular, instead of defining the errors with respect to the levels of $\mathcal{T}$ we define them with respect to the nodes of $\mathcal{T}$.

Let $e_u = ||R_u||_\infty$ be the L_∞ error in node u. Let $E = \{e_u \mid u \in \mathcal{T}\}$ contains all errors in each node u of $\mathcal{T}$. Given a reconstruction error threshold ε, we could traverse $\mathcal{T}$ to find the set of nodes U_ε having the largest errors that are at most ε. Then we transfer only the compressed pool values of all the ancestor nodes (along with U_ε) $U \supseteq U_\varepsilon$. For the decompression method we compute the sum of the spline functions of the pool values in U. It guarantees that the L_∞ error is at most ε. While this method works, it is very inefficient to store and compress all errors e_u in E because of the high compression ratio. Ideally, we would like to keep the compression ratio as low as possible. Next, we describe three different ways to do it.

Imagine that the error in a node e_u is high and the it remains high in the next t levels in the subtree of u as we run the hierarchical compression method. For instance, assume that the error is always at least $\alpha \cdot e_u$ for $\alpha < 1$ in the next t levels. After the hierarchy visits the node u at level i, we can directly jump to the level $i + t$ and continue the hierarchical approach, skipping all of the intermediate levels in the subtree of u. The selection of the parameters t, α depend on a given compression ratio upper bound and the original errors E in the nodes of $\mathcal{T}$.

As we observe in the experiments, the compression ratio of our method is better than the overall compression ratio of the other methods. Hence, we have the ability to store more errors in the nodes and still improve on other techniques. However, we need to be strategic about the selection of those nodes. Similar to what we had in the previous technique, if the error at node v is not much smaller than the error at its parent node u, i.e., $e_v \geq \alpha e_u$, then we can skip e_v from E. The real parameter $\alpha \leq 1$ can be selected based on a given compression ratio upper bound and the current errors E.

Before our system transfers the pool values from the edge to the remote server, one idea is to identify a set of nodes to transfer with error at most ε, without storing any error value, i.e., $E = \emptyset$. In order to do so, we should spend some time during the decompression phase on the edge server to identify these nodes. The idea is the following: in the edge server, before we transfer the compressed data, we run a bottom-up procedure on $\mathcal{T}$ finding the error in each node that we visit until we find nodes with error greater than ε. Let U_ε be these nodes and let $U \supseteq U_\varepsilon$ be the set U_ε along with all of their ancestors. We send all of the pool values stored in U to the remote server. While this method increases the overall time to decompress the data, it has two significant advantages. First, it has the lowest compression ratio, since we do not need to store any errors. Second, the method is parallelizable, so it can be executed extremely fast on the edge server.

We did not implement these methods in the current experiments. In most of the datasets that we used, the errors are quite smooth over the levels of $\mathcal{T}$, so it was left to future work.

6 RELATED WORK

There has been substantial work in lossy numerical compression. Beyond the earlier discussion and the baselines used in our evaluation, there has been work in lossy compression for scientific data [7, 28, 41]. Like our study, most of these techniques focus on spatio-temporal data, where the data are organized on some continuous axis (such as space, time, or both). Example of such techniques include SZ family of compression algorithms [13, 30, 48] and the ZFP algorithm [14]. These algorithms follow a familiar structure to those described in our work, and offer L_∞ error guarantees. They generally pre-process/transform the data, quantize it, and then apply a byte-level encoding algorithm. We omit an extensive comparison because the problem settings are quite different. Scientific data compression algorithms generally focus on maximizing compression for

data at rest, and the applicability of these techniques in an online or mini-batch setting is more limited. Furthermore, to the best our our knowledge, multiresolution extensions to these algorithms have not been developed.

There has also been significant interest in multiresolution problems in adjacent areas. In approximate query processing, the DAQ project [37] uses a vertical layout of floating point bits to construct incrementally more accurate query results with error guarantees. This approach does no compression (i.e., it does not save on storage), but it does reduce the query latency for aggregate queries. The MLWeaving project has a similar approach to achieve machine learning training at different levels of precision [44]. Similarly, multiresolution trees have been widely applied in approximate query processing where data are aggregated at hierarchy predicates [5, 29, 31]. Similar "multiresolution" results for aggregate queries are seen in wavelet techniques for AQP [8], online aggregation [20], and sketching [12]. This work inspires our approach in HIRE, but is unfortunately only restricted to answering aggregate queries and not point-lookups. Wavelet techniques beyond the scope of traditional linear algebra decompositions have also been explored for multiresolution matrix compression [27].

Thus, we focus our study on a key set of baseline compression algorithms that: (1) can run efficiently in the online setting with rapid incoming data, (2) provide L_∞ error guarantees for point queries, and (3) do not require dataset-specific modeling for compression. It is worth mentioning recent data compression work that has been excluded from this study. The DeepSqueeze project [23] uses an auto-encoder to learn a low dimensional set of features that can represent the original dataset. In our experiments, we found that the "encoder" portion of the auto-encoder was very large in size (often the same order of magnitude as the data), and since it is dataset-specific, it has to be included in the compression ratio measurement. The encoder is required to compress any new data that arrives. Similarly, we build a simplified version of Squish that works assuming column-independence [18].

7 EXPERIMENTS

We conducted most of the main experiments on an Intel NUC with a dual-core 2.30 GHz i3-6100U processor, 16GB RAM, and a 256GB SSD.[4] All implementations were done in Python 3.9. Our technique, in addition to each baseline, was applied to 7 different multivariate time series data sets from the UCI [15] repository. For HIRE and the relevant baselines, we use the Turbo Range Coder to encode the final codes into bytes [38].

7.1 Datasets

Our data sets are from four different projects within the UCI repository and in each case we used a subset of the entire data as described: Heterogeneity activity recognition data set [42], from which we used four different data sets 53.3MB each, phones accelerometer (PA), phones gyroscope (PG), watch accelerometer (WA), watch gyroscope (WG); Sensors for home activity monitoring (SHAM) data set [21] 46.2MB; Individual household electric power consumption data set (IHEPC) 29.4MB; Bitcoin heist ransomware address data set (BC) [1], 50.3MB. We removed all non-numerical columns, since numerical values are the focus of our present research. We also removed missing data when present. As mentioned earlier, we assume a mini-batch model for data arrival. Since HIRE assumes mini-batches that are sized as powers of 2, we simply cut the different datasets to a multiple of our block sizes. This modification does not change our experiments and is simply done for consistency.

[4]Buff failed on BC dataset. We ran LFZip on a comparable Macbook Pro with a 1.4 GHz quad-core i5 processor due to incompatibility.

7.2 Baselines

Our baselines feature both lossless and lossy techniques. For the lossless techniques, we simply consider a single encoding of the data (i.e., it defaults to the "strict encoding" strategy described before). Below is a brief description of each baseline:

- **Identity Gzip (IdG)**: Lossless compression baseline; we apply Gzip to an array of numbers represented as floating point values.
- **Quantize (Q)**: We convert each floating point number to an integer according to a user-defined error threshold, thereby saving exponent and mantissa bits (see 2.2.1 for more details). The numbers are stored as integers with bitpacking. The compression ratio is proportional to $\lceil \log_2 1/\epsilon \rceil$ which captures the effect of the error threshold ϵ alone on the size of the compressed representation.
- **Quantize Gzip (QGZ)**: This method consists of a quantization step and Gzip as the downstream compressor.
- **Quantize TRC (QTRC)**: This method consists of a quantization step and the Turbo Range Coder (TRC) as the downstream compressor. TRC uses a Burrows–Wheeler transform (BWT) [6] to rearrange blocks of values into runs of the same symbol (i.e., integer), and then applies an arithmetic encoder [46] during the entropy encoding step.
- **Sprintz (Spz)**: We apply quantization to map to integer time series. First, it predicts the current sample based on the previous sample and encodes its difference (see 2.2.2 for more details). Second, it bitpacks the errors and stores metadata to allow for unpacking. Third, it uses run length encoding on blocks of all zero errors. Lastly, it Huffman encodes the headers and payload [4].
- **AdaptivePiecewiseConstant (APC)** Piecewise approaches decompose a time series into segments and use the segments to approximate the time series. Our version of the algorithm adaptively sizes segments to enforce an error bound [9]. The downstream data are compressed with GZip.
- **Gorilla (Grl)**: This technique employs a scheme that consists of a bitwise XOR between pairs of consecutive values. It then produces a lossless encoding for each pair based on the number of leading or trailing zeros and the meaningful bits present [36]. The downstream data are compressed with GZip.
- **LFZip:** This method employs a pipeline of causal prediction, quantization, and entropy coding. It first uses a Normalized Least Mean Square filter to predict the next value in the sequence based on the previous values. Subsequently, the difference between the prediction and the actual value is obtained and quantized to a user-determined L_∞ error. Finally, a version of BWT is applied to the quantized data [10].
- **Buff**: This technique was designed to exploit bounded range and precision in floating point sensor data. It eliminates less-significant bits by adjusting for a certain precision. It also compresses the integer and mantissa bits independently [32].

7.3 Performance Overview

Table 5 from Subsection 7.6 exemplifies the main argument behind the benefits of using HIRE over competing methods. We show the compression ratio of the compressed data at 10 different error thresholds. The breakdown of compression ratio shows that at very low thresholds, we can always choose an error threshold using HIRE that will yield a more compressed representation than the competing methods. Hence, we need to encode the other methods at all possible error thresholds. On the other hand, because of how our method is constructed, we only need to store the lowest

threshold and we are still able to retrieve all of the intermediate thresholds by traversing the tree structure until we reach the desired resolution.

Overall, we perform better than all of the competing methods when it comes to the *combined compression ratio* of all resolutions considered. For the experiments that we performed, 10 different resolutions were chosen that are inside the scope of real world usage: $\varepsilon^* \in \{0.15, 0.1, 0.075, 0.05, 0.025, 0.01, 0.0075, 0.005, 0.0025, 0.001\}$. A key factor motivating our choice of thresholds is that for Quantize (Q) the combined compression ratio across the error thresholds adds up to approximately 1.0 regardless of the specific dataset. HIRE requires half of the space to store all of the resolutions when compared to other baselines. Furthermore, the compression latency of HIRE is significantly better than methods with low compression ratios, again due to the fact that we only need to run the encoder once to produce multiple resolutions.

7.4 Compression Ratio

We evaluated the compression performance of each method on all seven datasets. The resulting lossy compression ratios comprised of all of the resolutions are displayed in the bottom portion of Table 1; the single encoding compression ratios for the lossless baselines are included in the top portion of the table. The best methods for a *single error threshold scenario* are Quantize Turbo Range Coder (QTRC) and Sprintz (Spz), but when summing up all of the different resolutions, their performance is on average two times worse than HIRE. One exceptional case is the SHAM data set, where the performance was really close to ours. On this specific dataset, the sample coefficient of variation (CV) i.e., the standard deviation over the mean, is extremely low due to the relative stability of the data. This impacts HIRE's smoothness throughout the various levels, which consequently results in a worse performance.

	PA	PG	WG	WA	SHAM	HIEPC	BC
Grl	0.818	0.592	0.638	0.806	0.765	0.295	0.806
IdGZ	0.555	0.402	0.449	0.464	0.769	0.217	0.230
Buff	0.421	0.390	0.390	0.421	0.468	0.453	*
Q	1.000	1.000	1.000	1.000	1.000	1.000	1.000
QGZ	0.526	0.415	0.351	0.431	0.172	0.436	0.175
QTRC	0.258	0.156	0.119	0.170	0.029	0.183	0.116
Spz	0.275	0.157	0.119	0.179	0.023	0.222	0.164
APC	1.844	1.004	0.986	1.093	0.207	0.551	0.296
LFZip	0.349	0.212	0.168	0.210	0.043	0.545	0.434
HIRE	0.116	0.085	0.070	0.085	0.021	0.091	0.061

Table 1. The compression ratios for the lossy baselines in the multiresolution setting compared to HIRE (bottom). The single trivial resolution is reported for the lossless baselines (top). Some of the values are above 1 due to the multiresolution sum of different thresholds.

7.5 Compression and Decompression

The evaluation of compression latency includes the compression algorithm (counting entropy coding) and writing the compressed data to disk. HIRE outperforms all of the lossy low compression ratio baselines as displayed in Table 2.[5] The main driver of HIRE's significant performance advantage

[5]We report the latency *without bitpacking* times. We found in our experiments that bitpacking introduced a latency bottleneck that skewed some of the results in favor of HIRE. We removed bitpacking time from those baselines, inflating their results.

is the fact that we require only a single call to our compression routine in order to produce multiple resolutions. This directly contrasts with the lossy baseline methods, each of which compresses the data once per error threshold to produce 10 separable encodings. In the case of the lossless baselines, there is a single encoding at the trivial resolution $\epsilon = 0$ reflecting one function call.

	PA	PG	WG	WA	SHAM	HIEPC	BC
Grl	1.602	1.497	1.628	1.649	1.668	5.349	1.649
IdGZ	0.362	0.377	0.344	0.340	0.162	0.145	0.452
Buff	0.030	0.029	0.029	0.030	0.235	0.148	*
Q	0.056	0.051	0.066	0.056	0.102	0.073	0.120
QGZ	1.595	1.039	0.910	1.450	1.269	8.264	43.50
QTRC	7.617	7.423	6.930	7.221	29.72	20.99	33.07
Spz	7.125	7.465	7.241	7.150	55.52	35.30	63.12
APC	1.844	1.003	0.986	1.093	0.207	0.561	0.362
LFZip	6.509	5.880	5.854	6.175	78.78	37.58	54.36
HIRE	0.431	0.376	0.357	0.388	1.027	2.020	1.921

Table 2. Compression latency (s): sum of all resolutions (lossy) and single trivial resolution (lossless).

The evaluation of decompression latency includes reading the encoding from disk and the decompression algorithm (counting entropy coding). When it comes to decompression latency, we report the average value for all of the different resolutions in Table 3. HIRE performs about the same or better than the lossy low compression ratio methods on the first four data sets. However, HIRE is significantly outpaced by QTRC on the last three. Once again, this is likely due to the characteristics of the datasets. Applying HIRE to the first four datasets generates very smooth residuals, which allows for a shorter path of traversal along the tree until the desired resolution is reached. On data sets with extreme CVs, the traversal may not terminate until very low levels, which is overall detrimental to decompression latency. Gorilla performs considerably worse in this scenario due to its complex recursive encoding where adjacent values are compared and bitwise operations executed.[6]

	PA	PG	WG	WA	SHAM	HIEPC	BC
Grl	2148	1870	2118	2313	29753	8508	2313
IdGZ	0.08	0.07	0.07	0.08	0.48	0.22	0.34
Buff	0.02	0.02	0.02	0.02	0.17	0.12	*
Q	0.01	0.01	0.01	0.01	0.04	0.02	0.03
QGZ	0.01	0.01	0.01	0.01	0.05	0.02	0.04
QTRC	0.07	0.06	0.05	0.06	0.10	0.15	0.18
Spz	0.51	0.51	0.51	0.51	2.15	1.36	2.35
APC	0.18	0.10	0.10	0.11	0.02	0.06	0.04
LFZip	0.50	0.49	0.48	0.48	7.50	3.37	4.86
HIRE	0.07	0.06	0.06	0.07	0.19	0.40	0.23

Table 3. Decompression latency (s): average of all resolutions (lossy) and single trivial resolution (lossless).

[6]We do note the caveat that the Python implementation that we used for Gorilla likely does not perform bitwise operations efficiently which may explain some of the poor results for latency.

7.6 Edge Retrieval Experiments

We ran an experiment that mirrors the example in Section 2. We simulated a retrieval task with data (PA dataset) collected and stored on an NVIDIA Jetson Nano with an ARM64 processor (4 cores) and 4GB of RAM. In other words, data are stored on the edge device and are retrieved by remote applications. We measure the end-to-end latency of this process for four multiresolution encoding strategies: strict encoding, all encoding, lazy re-coding, and HIRE. We only report using a standard lossless method (IdGZ) and the best competing lossy method (QTRC) as baselines. Results across different baseline algorithms were highly similar and are available upon request.

The retrieval workload is simple. There are ten different resolution requirements. A retrieval request is a request to the edge for one block of data (53.3MB) at one of those given resolutions. We assume that choice is uniformly at random. We evaluate the multiresolution strategies with the following metrics:

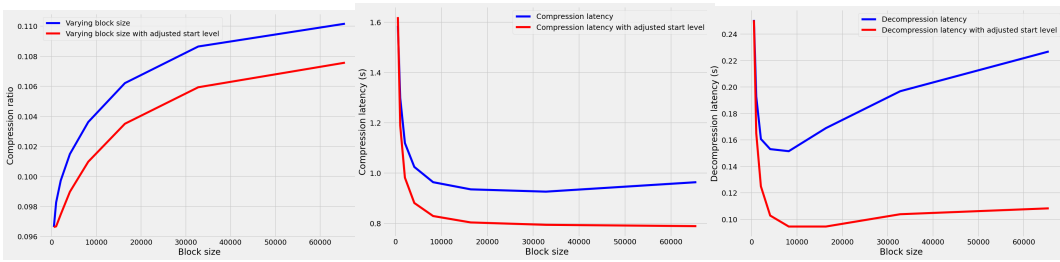
- **Ingestion Latency (IL).** The time needed to compress one block of new data (53.3MB).
- **Transfer Size (TS).** The average amount of data transferred from edge to remote per retrieval request.
- **Retrieval Overhead (RO).** The average time needed beyond data transfer to decompress or transcode per retrieval request.
- **Local Storage (LS).** The average amount of data stored per block locally.

Algorithm	Scheme	IL	TS	RO	LS
QTRC	Strictest	4.8s	3.8MB	0.3s	3.8MB
QTRC	All	44.5s	1.3MB	0.2s	12.9MB
QTRC	Lazy	4.8s	1.3MB	5.1s	3.8MB
GZip	Lossless	0.9s	28MB	0.17s	28MB
HIRE	Multiresolution	1.2s	2.0MB	0.18s	5.8MB

Table 4. This table compares different edge retrieval protocols on four metrics: ingestion latency, transfer size, retrieval overhead, local storage. HIRE has a much lower latency in both ingestion and retrieval compared to other lossy baselines, while improving on a lossless baseline by over 14x in terms of compression ratio.

The results for the compression and decompression latencies are displayed in Table 4. First, HIRE is significantly faster than the best compression baseline in terms of both ingestion latency and retrieval overhead. In fact, it only has a minor overhead over a lossless GZIP baseline. Second, while HIRE does transfer more data than the lossy baseline in the "all" or "lazy" settings, it is still competitive to them and is significantly smaller than the lossless compression (by 14x). We believe that this is a tradeoff worth making. Edge devices are storage constrained, and HIRE allows for a more efficient use of local storage. Next, network transfers are a major component in energy usage, which HIRE directly addresses. **In other words, we achieve similar latencies to a simple lossless compression framework but can significantly lower the data footprint if the downstream applications can tolerate inaccurate results.**

7.6.1 Detailed Breakdown of Compression Ratios. For the sake of completeness, we include the per-threshold compression ratios for HIRE. The results for the compression ratio are displayed in Table 5. One key point worth reemphasizing is that *we only need to store the encoding for the strictest threshold in HIRE*, which in this case is 0.1164 at 0.001 error, since we are able to reconstruct all of the intermediate representations from that single encoding.



(a) Compression ratio (b) Compression latency (c) Decompression latency

Fig. 3. Breakdown of performance at different block sizes

Thresholds	0.15	0.10	0.075	0.050	0.025	0.010	0.0075	0.005	0.0025	0.001
HIRE	0.008	0.009	0.011	0.014	0.023	0.038	0.045	0.057	0.082	0.116
IdGZ	0.555	0.555	0.555	0.555	0.555	0.555	0.555	0.555	0.555	0.555
QTRC	0.005	0.006	0.007	0.009	0.014	0.025	0.029	0.036	0.051	0.076

Table 5. Compression ratio for different resolutions on an edge device

7.7 Micro-benchmarks

We ran several different experiments in order to understand our method's most important hyperparameters: block size and start level. Additionally, we tested different pooling functions and their effects on the relevant metrics. All experiments in this subsection were performed on the Phones Accelerometer (PA) data set.

7.7.1 Block Size. A block represents a subset of the entire data set meant to be compressed. It allows for an online/streaming application of the method, since one can wait until a pre-determined block size is buffered before applying HIRE. However, it also affects the performance of the algorithm. We evaluate two different scenarios in which we vary the block size: with and without adjusting the starting level.

We notice in Figure 3a that as we increase the block size, the compression ratio increases. This behavior can be attributed to two distinct reasons. First, the larger the block size, the greater the range of values within each block. This phenomenon could possibly impact the variance of the residuals at each level, leading to less redundancy for the downstream compressor to exploit. Second, the presence of outliers can be mitigated at smaller block sizes, since their impact will be contained to a smaller subset of the data. Starting at a lower level in the tree structure also reduces the amount of data being stored, which leads to a positive impact on the compression ratio at each block size.

On the other hand, the increase in block size has a positive effect on compression latency, especially when going from a very small block (512 time steps) to 8192 as we see in Figure 3b. After that, the compression latency plateaus on the adjusted level scenario and slightly increases on the regular one. We attribute this fact to hardware limitations that impede further scalability due to the fixed number of CPU cores. Finally, when analyzing the decompression latency shown in Figure 3c, we can clearly see that once again it decreases with the block size until it reaches a min value at roughly the same point as in compression—8192. This observation is also due to hardware constraints. The level-adjusted scenario performs significantly better than the regular one while being less affected by the hardware constraints.

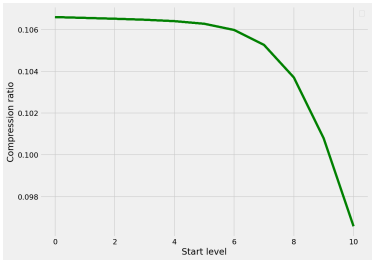


Fig. 4. Compression ratio for different starting levels

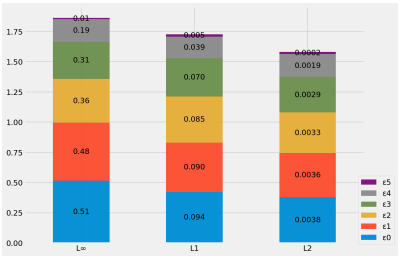


Fig. 5. L_∞, L_1, L_2 errors for 6 levels

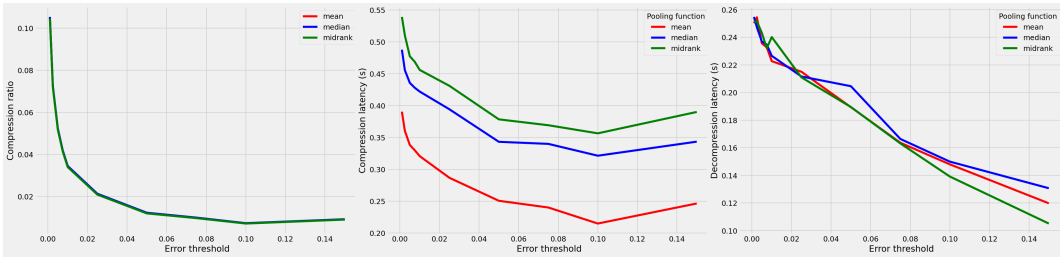
7.7.2 Starting level. The starting level corresponds to the initial number of segments into which we break the original time series X and apply the pooling function; or equivalently, the initial number of nodes at level Γ of the binary tree $\mathcal{T}$. That is, we do not need to start the recursion at the first level, or even at the first few levels for that matter. Concretely, there is often little to no value in pooling large segments, particularly when the block size is purportedly large. We can therefore adjust the starting level in order to improve both compression ratio and latency, albeit we lose the resolutions that correspond to the levels that are skipped. Such a trade-off is important in certain cases, as we may want to maintain a sufficient number of levels to allow for a specific number of resolutions. Figure 4 shows that as we increase the starting level, the compression ratio decreases. Furthermore, the inverse relationship observed here is exponential in nature, since the number of values stored after pooling increases exponentially—specifically by a factor of 2—until we reach the upper bound of T values at the leaf nodes of the hierarchy.

7.7.3 Pooling function. The pooling function plays an important role in how the data are summarized and consequently the resulting residuals. We described its role in detail in Section 4.1.1. In Figure 6a, we display the compression ratios at various error thresholds for the three pooling functions: mean, median, and midrank. In Figures 6b and 6c respectively, we compare the compression and decompression latency achieved by the three pooling functions. The compression latency of the mean is markedly lower at all of the thresholds we tested. The decompression latencies are consistent at smaller error thresholds but diverge at larger error thresholds.

7.8 Additional Experiments

In this section we run HIRE considering i) different error functions, and ii) a different splitting method. First, we implement Algorithm 1 using the mean pool function and measure the L_1 and L_2 errors for each level of the compression tree, as described in Section 5.3. In particular, we run HIRE on the first 1024 samples from the first column of IHEPC and measure the L_∞, L_1, and L_2 error in the last 6 levels of the compression tree. The results are displayed in Figure 5. Note that the L_1 and L_2 errors are divided by the size of the time series to obtain an element-wise metric (given in black), and the size of the bars are adjusted for scale. We observe that while HIRE was designed to explicitly bound the L_∞ error, the L_1 and L_2 errors are implicitly bounded. Furthermore, the errors decrease in lockstep with one another as HIRE progresses to lower levels. Note that we do not apply a distinct optimized pool function to each error. Instead, we run the traditional HIRE and show that it can still decrease the L_∞, L_1, and L_2 errors in low levels of the hierarchy.

Second, we implement a variation of HIRE that splits at the optimum location as described in Section 5.3. We compare the compression ratio and the compression latency of optimal split HIRE (OS) and midpoint split HIRE (MS). The experiment was run on a block of 4096 samples from the first column of IHEPC. Both versions of HIRE start from the first level and have an error threshold



(a) Compression ratio (b) Compression latency (c) Decompression latency

Fig. 6. Breakdown of performance for three different pooling functions

of 0.01. The results are displayed in Table 6. Midpoint split (MS) of HIRE has a smaller compression ratio and lower compression latency than OS. While OS optimizes the split (minimize the maximum squared error), there are two drawbacks that affect the results. First, OS needs to store twice as many values as MS because it stores the size of each time series in the hierarchical encoding. This increases the compression ratio. Second, OS needs additional time to find the optimal split with respect to the minimum squared error as described in Section 5.3. On the other hand, MS can find the splitting point in $O(1)$ time, so the compression latency is much lower for the MS method.

	Ratio	Latency (s)
OS	0.242	2.042
MS	0.083	0.012

Table 6. Compression ratio and compression latency (s) of OS and MS on a small block. The relative contribution to OS ratio is 0.114 for the optimum split pooled values alone and 0.146 for storing the sizes alone.

8 CONCLUSION

We presented HIRE, a novel system for multiresolution compression that uses hierarchical residual encoding for time series data. We showed that strict, multiple, and lazy encoding suffer from a high transfer cost, high compression ratio, or high retrieval overhead in edge storage and retrieval applications. We proposed an efficient technique to handle multiresolution compression that alleviates the limitations of the previous methods. Our experiments validate that our system performs better than the baselines at multiresolution compression for edge computing applications. HIRE can also be extended to the multidimensional case (e.g., image compression). As mentioned, one simple way is to encode each column independently. A more involved and efficient way is to extend our hierarchical method in order to natively support more dimensions. We leave the details and the implementation of the multidimensional case to future work.

9 ACKNOWLEDGMENTS

This work was made possible by generous grants from Intel, SRI/TATRC, CERES Center for Unstoppable Computing, and support from the Data Science Institute at the University of Chicago.

REFERENCES

[1] Cuneyt G Akcora, Yitao Li, Yulia R Gel, and Murat Kantarcioglu. 2021. BitcoinHeist: topological data analysis for ransomware prediction on the bitcoin blockchain. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 4439–4445.

[2] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[3] Gennady Antoshenkov. 1997. Dictionary-based order-preserving string compression. *The VLDB Journal* 6, 1 (1997), 26–39.

[4] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.

[5] Francesco Buccafurri, Filippo Furfaro, Giuseppe M Mazzeo, and Domenico Saccà. 2011. A quad-tree based multiresolution approach for two-dimensional summary data. *Information Systems* 36, 7 (2011), 1082–1103.

[6] M. Burrows and D. J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report.

[7] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1201–1220.

[8] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate query processing using wavelets. *The VLDB Journal* 10, 2 (2001), 199–223.

[9] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. 2002. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM Trans. Database Syst.* 27, 2 (jun 2002), 188–228. https://doi.org/10.1145/568518.568520

[10] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. 2020. LFZip: Lossy Compression of Multivariate Floating-Point Time Series Data via Improved Prediction. In *2020 Data Compression Conference (DCC)*. 342–351. https://doi.org/10.1109/DCC47342.2020.00042

[11] Chris Chatfield. 1978. The Holt-winters forecasting procedure. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 27, 3 (1978), 264–279.

[12] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294.

[13] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 730–739.

[14] James Diffenderfer, Alyson L Fox, Jeffrey A Hittinger, Geoffrey Sanders, and Peter G Lindstrom. 2019. Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing* 41, 3 (2019), A1867–A1898.

[15] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[16] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[17] Johannes Fischer and Volker Heun. 2007. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 459–470.

[18] Yihan Gao and Aditya Parameswaran. 2016. Squish: Near-optimal compression for archival of relational datasets. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1575–1584.

[19] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[20] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 171–182.

[21] Ramón Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai Rulkov, and Irene Rodriguez-Lujan. 2016. Online Decorrelation of Humidity and Temperature in Chemical Sensors for Continuous Monitoring. *Chemometrics and Intelligent Laboratory Systems* 157 (07 2016). https://doi.org/10.1016/j.chemolab.2016.07.004

[22] BR Hutchinson and GD Raithby. 1986. A multigrid method based on the additive correction strategy. *Numerical Heat Transfer, Part A: Applications* 9, 5 (1986), 511–537.

[23] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: deep semantic compression for tabular data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1733–1746.

[24] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2001. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining*. IEEE, 289–296.

[25] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2004. Segmenting time series: A survey and novel approach. In *Data mining in time series databases*. World Scientific, 1–21.

[26] Eamonn J Keogh and Michael J Pazzani. 2000. Scaling up dynamic time warping for datamining applications. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. 285–289.
[27] Risi Kondor, Nedelina Teneva, and Vikas Garg. 2014. Multiresolution matrix factorization. In *International Conference on Machine Learning*. PMLR, 1620–1628.
[28] David Krasowska, Julie Bessac, Robert Underwood, Jon C Calhoun, Sheng Di, and Franck Cappello. 2021. Exploring Lossy Compressibility through Statistical Correlations of Scientific Datasets. In *2021 7th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-7)*. IEEE, 47–53.
[29] Iosif Lazaridis and Sharad Mehrotra. 2001. Progressive approximate aggregate queries with a multi-resolution tree structure. *Acm sigmod record* 30, 2 (2001), 401–412.
[30] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 438–447.
[31] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. 2021. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proceedings of the 2021 International Conference on Management of Data*. 1129–1141.
[32] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. 2021. Decomposed Bounded Floats for Fast Compression and Queries. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2586–2598. https://doi.org/10.14778/3476249.3476305
[33] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. 2021. VergeDB: A Database for IoT Analytics on Edge Devices. In *CIDR*.
[34] John Paparrizos, Chunwei Liu, Aaron J Elmore, and Michael J Franklin. 2020. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1887–1905.
[35] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.
[36] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078
[37] Navneet Potti and Jignesh M Patel. 2015. Daq: a new paradigm for approximate query processing. *Proceedings of the VLDB Endowment* 8, 9 (2015), 898–909.
[38] Powturbo. 2022. *Turbo Range Coder*. Retrieved March 23, 2023 from https://github.com/powturbo/Turbo-Range-Coder
[39] Julian Seward. 1996. bzip2 and libbzip2. *avaliable at http://www. bzip. org* (1996).
[40] John Shahid. 2019. InfluxDB documentation.
[41] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing frontiers and innovations* 1, 2 (2014), 76–88.
[42] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. 2015. Smart Devices Are Different: Assessing and MitigatingMobile Sensing Heterogeneities for Activity Recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems* (Seoul, South Korea) *(SenSys '15)*. Association for Computing Machinery, New York, NY, USA, 127–140. https://doi.org/10.1145/2809695.2809718
[43] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE transactions on consumer electronics* 38, 1 (1992), xviii–xxxiv.
[44] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang. 2019. Accelerating generalized linear models with MLWeaving: A one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment* 12, 7 (2019), 807–821.
[45] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. 2003. Overview of the H. 264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology* 13, 7 (2003), 560–576.
[46] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic Coding for Data Compression. *Commun. ACM* 30, 6 (jun 1987), 520–540. https://doi.org/10.1145/214762.214771
[47] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M Grulich, Ariane Ziehn, and Volker Mark. 2020. Nebulastream: Complex analytics beyond the cloud. *Open Journal of Internet Of Things (OJIOT)* 6, 1 (2020), 66–81.
[48] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1643–1654.

Received July 2022; revised October 2022; accepted November 2022