THE UNIVERSITY OF CHICAGO

A PRINCIPLED FRAMEWORK FOR ADAPTIVE LOSSY DATA COMPRESSION WITH LINFINITY ERROR GUARANTEES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
BRUNO BARBARIOLI

CHICAGO, ILLINOIS
DECEMBER 2023

Copyright © 2023 by Bruno Barbarioli

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES vi

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT ix

1 INTRODUCTION 1

2 HIERARCHICAL RESIDUAL ENCODING FOR MULTIRESOLUTION TIME SERIES COMPRESSION 2
- 2.1 Introduction 2
- 2.2 Background 5
  - 2.2.1 Time Series Compression Basics 5
  - 2.2.2 Compression with Bounded L_∞ Error 7
  - 2.2.3 The Multiresolution Problem 10
- 2.3 Mathematical Intuition 12
  - 2.3.1 Residualization 12
  - 2.3.2 Relevance to Time Series Compression 13
  - 2.3.3 Novelty 15
- 2.4 Hierarchical Residual Encoding 15
  - 2.4.1 Algorithm Basics 16
  - 2.4.2 Algorithm Description 18
  - 2.4.3 Decompression 20
- 2.5 Optimizations 21
  - 2.5.1 Algorithmic Optimizations 22
  - 2.5.2 Implementation Optimizations 25
  - 2.5.3 Extensions 27
- 2.6 Related Work 30
- 2.7 Experiments 32
  - 2.7.1 Datasets 32
  - 2.7.2 Baselines 33
  - 2.7.3 Performance Overview 34
  - 2.7.4 Compression Ratio 35
  - 2.7.5 Compression and Decompression 36
  - 2.7.6 Edge Retrieval Experiments 38
  - 2.7.7 Micro-benchmarks 40
  - 2.7.8 Additional Experiments 43
- 2.8 Conclusion 45

3 RASTERSTORE: ADAPTIVE COMPRESSION FOR FULLY RASTERIZED GEOSPATIAL DATA . . . 46
3.1 Introduction . . . 46
3.2 Preliminaries . . . 48
3.2.1 Compression Basics . . . 49
3.2.2 RasterStore Overview . . . 50
3.3 Our Method . . . 51
3.4 Optimizations . . . 53
3.4.1 Statistical pivot selection . . . 54
3.4.2 Pivot selection given the ratio . . . 56
3.4.3 Ratio selection given the pivot . . . 58
3.4.4 Pivot and ratio selection . . . 59
3.5 Experiments . . . 61
3.5.1 Datasets . . . 62
3.5.2 Baselines . . . 62
3.5.3 Main Experiments . . . 63
3.5.4 Micro-benchmarks . . . 68
3.6 Conclusion . . . 74

4 GEOSPATIAL CASE STUDY: THE SOCIOME DATA COMMONS . . . 75
4.1 Introduction . . . 75
4.2 Materials and Methods . . . 76
4.3 Software Implementation . . . 77
4.4 Governance and Sustainability . . . 78
4.5 Asthma Pilot Methodology . . . 79
4.5.1 Geocoding . . . 80
4.5.2 Missing data . . . 80
4.5.3 Outcome definition . . . 80
4.5.4 Spatial clustering . . . 81
4.5.5 Sociome Data Commons . . . 82
4.5.6 Model . . . 83
4.5.7 Evaluation . . . 85
4.5.8 Protocol testing . . . 86
4.5.9 Challenges . . . 86
4.6 Results . . . 86
4.6.1 Sociome Data Commons . . . 86
4.6.2 Asthma pilot results . . . 88
4.6.3 Sociome Data Commons datasets . . . 89
4.7 Conclusion . . . 91
4.7.1 Platform discussion . . . 91
4.7.2 Analysis discussion . . . 92
4.7.3 Limitations . . . 94

REFERENCES 96

LIST OF FIGURES

2.1 (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 14
2.2 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. 18
2.3 Breakdown of performance at different block sizes 41
2.4 Compression ratio for different starting levels 42
2.5 L_∞, L_1, L_2 errors for 6 levels 42
2.6 Breakdown of performance for three different pooling functions 44

3.1 How RasterStore allocates error in a given window. 52
3.2 Compression ratio improvement over QTRC 63
3.3 Breakdown of performance for several different data resolutions 68
3.4 Breakdown of performance for several different window sizes 70
3.5 Breakdown of performance for several different error allocations 70
3.6 Breakdown of performance for several flattening methods: row, window, contiguous window 72
3.7 Breakdown of performance for several pivot methods 73

4.1 SDC Guiding principles 77
4.2 SDC Standards 79
4.3 SDC Data pipeline 84
4.4 Clinical Maps 89
4.5 Sociome Maps 91

LIST OF TABLES

2.1 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. 36
2.2 Compression latency (s): sum of all resolutions (lossy) and single trivial resolution (lossless). 37
2.3 Decompression latency (s): average of all resolutions (lossy) and single trivial resolution (lossless). 38
2.4 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. . . . 39
2.5 Compression ratio for different resolutions on an edge device 40
2.6 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. 45
3.1 Compression ratio for different error thresholds (%) 65
3.2 Compression latency for different error thresholds (s) 66
3.3 Decompression latency for different error thresholds (s) 67

ACKNOWLEDGMENTS

I thank my parents for their unconditional love and their support throughout my entire life. This work wouldn't be possible without my adviser's, Sanjay Krishnan, amazing insights and ideas and my main collaborators Gabriel Mersy and Stavros Sintos help.

ABSTRACT

Recent advances in edge computing and networking have led to a renaissance in the field of ubiquitous sensing. Emerging applications range from imaging in autonomous vehicles to geospatial data in remote satellite imagery. For such data, storage and communication is a major bottleneck and lossy data compression can help. However, lossy data compression is not as well developed beyond "perceptual" data formats such as images, audio, and video. The errors such algorithms introduce can interact with downstream sensing algorithms and lead to unforeseen consequences. This dissertation develops a principled approach to lossy data compression over structured arrays of floating-point data. It presents algorithms that guarantee the l_∞ error, or the maximum error over all elements. These algorithms rely on different combinations of floating point quantization and spatial quantization, both of which preserve l_∞ guarantees, and thus can be adaptively interspersed depending on the data. Results show improved compression ratios for high-resolution geospatial raster data and novel algorithms for optimizing edge-resident time-series data.

CHAPTER 1

INTRODUCTION

In recent years, the realm of edge computing and networking has witnessed remarkable strides, igniting a resurgence of interest in ubiquitous sensing. As the computing infrastructure moves closer to the data source, applications are broadening their reach, from autonomous vehicles utilizing real-time imaging, satellite systems capturing geospatial information to ubiquitous sensing in farms and remote areas. These advancements have undoubtedly revolutionized the way we perceive and interact with the world around us. Nevertheless, they have also precipitated new challenges, most notably in data storage and communication.

Data storage and communication bottlenecks can be a significant hindrance to the functionality of sensing systems. High-resolution data arrays are particularly demanding in terms of storage space and bandwidth, making it increasingly difficult to operate these systems efficiently. In order to mitigate these challenges lossy compression has been widely used in order to compress large scale data. Lossy compression introduces a trade-off between compression ratio and error introduced on the data. The greater the compression desired the worse the reconstruction of the encoding.

In this dissertation we explore a special case of lossy compression, where the error is bounded by the worst reconstruction obtained on all data points in the reconstructed set, i.e. l_∞ error. We propose methods to compress multivariate time series data in a hierarchical manner and raster geospatial data while respecting the user choice of a $l\infty$ bound. Our algorithms combine float point and spatial quantization with different techniques in order to improve the state of the art.

We evaluate our systems against several different baselines at each domain, comparing not only the compression ratio, but the latencies required to compress and decompress the data. We show that our methods bring significant advantages in their respective tasks improving either the overall compression ratio or one of the latency metrics.

CHAPTER 2

HIERARCHICAL RESIDUAL ENCODING FOR MULTIRESOLUTION TIME SERIES COMPRESSION

2.1 Introduction

In today's data analytics infrastructure, it is common for data storage to be separated from computational resources ?. 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 ??. 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. Examples of such approaches include dictionary coding for string compression ??, GZip/BZip for byte-sequence compression ?, and turbo-coding for integer compression ?. 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 ?, H.264 for video ?, and a variety of techniques for scientific data ???. 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 ?????. 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 sin-

gle 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 **?** and approximate query processing **??**; 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 **?**, and in differential equation solvers **?**.

2.2 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.

[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 **?**, 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.

2.2.1 Time Series Compression Basics

Let us consider a time series with observations:

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

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

2.2.2 Compression with Bounded L_∞ Error

There is an extensive body of literature on time series compression techniques (refer to survey **?**). 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.

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 **?**. 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):

$$\mathsf{enc}(x_i) = \left\lfloor \frac{(x_i - x^-)}{(x^+ - x^-)} \cdot \frac{1}{R_\epsilon} \right\rfloor \tag{2.1}$$

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:

$$x_i' = \mathsf{dec}(\mathsf{enc}(x_i)) = \mathsf{enc}(x_i) \cdot R_\epsilon \cdot (x^+ - x^-) + x^- \tag{2.2}$$

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.

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:

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

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

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 ?. 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) ? and Piecewise Linear Approximation (PLA) ?. 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.

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 **?**. 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.

2.2.3 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.

[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.

[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.

2.3 Mathematical Intuition

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

2.3.1 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 **?**. 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:

$$R_0 = X \tag{2.3}$$

$$A_i = \mathsf{approx}(R_i)$$

$$R_i = R_{i-1} - A_{i-1}$$

where $\mathsf{approx}(\cdot)$ 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.

2.3.2 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

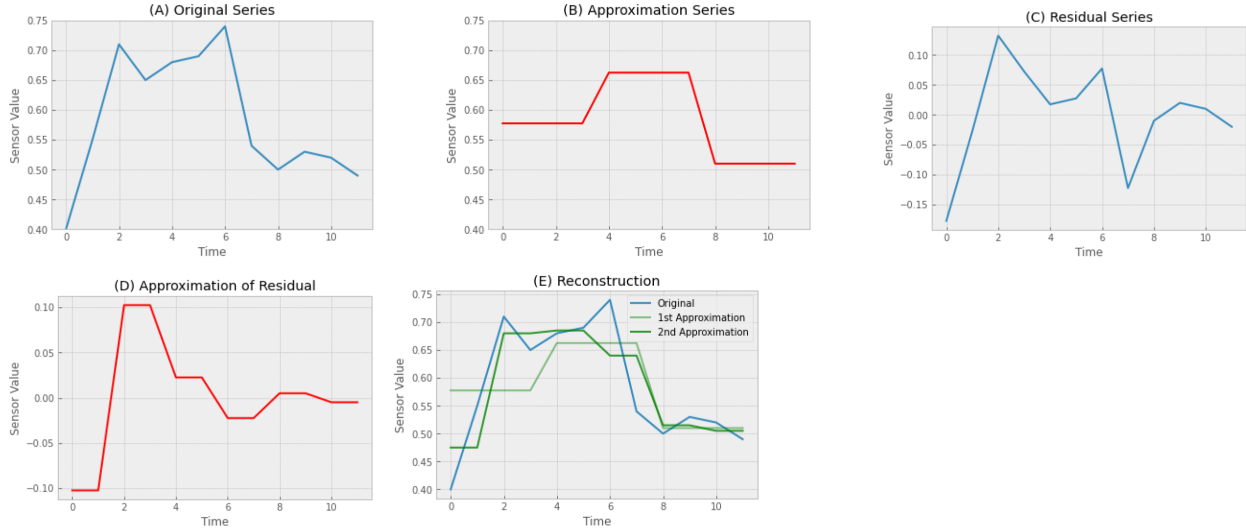


Figure 2.1: (A) Illustrates an example time series, (B) illustrates a coarse approximation of the time series, (C) illustrates the residual series, (D) illustrates a finer approximation of the residual series, and (E) shows how the summations of the residuals lead to progressively better approximations

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 2.1(A), we have plotted an example time series. One can coarsely approximate this time series with a piecewise constant approximation of 3 segments (Figure 2.1(B)). Between this approximation and the original series, there is a residual (Figure 2.1(C)). This residual can be captured by another piecewise constant approximation with 6 segments (Figure 2.1(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 2.1(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.

2.3.3 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.

2.4 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.

2.4.1 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.

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 $p_f(X) : \mathbb{R}^n \to \mathbb{R}$, $p_f(X) = f(x_0, \ldots, x_{n-1})$, 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

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.

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 2.2.2 describes how to translate ε^* into a quantization threshold.

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.

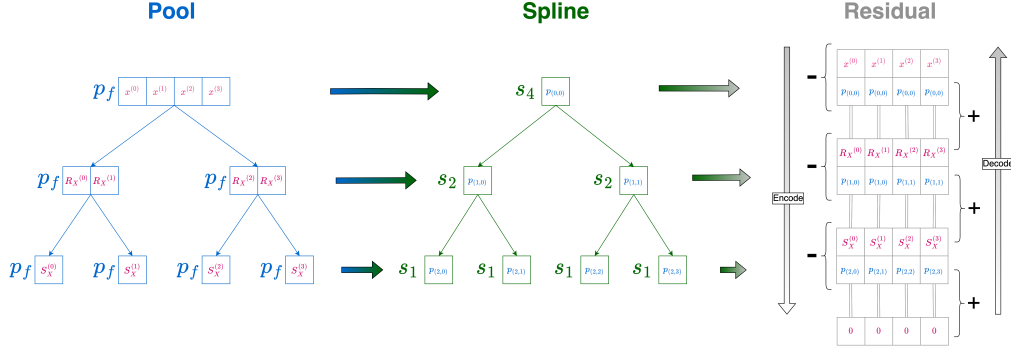


Figure 2.2: The main algorithmic workflow. Compression flows from left to right and top to bottom just like reading text. Decompression flows from bottom to top and inverts residualization with a summation.

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 can plug in this approximation into our recurrence equation of the previous section (Equation 2.3), where $\mathsf{approx}(R_i) = s_w(p_f(R_i))$.

2.4.2 Algorithm Description

The main idea is to run the procedure described above recursively in a hierarchical manner following the recurrence described in Equation 2.3. 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

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

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 2.2.

Algorithm 1: HIERARCHICAL

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

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 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.

2.4.3 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 $u_0 \to \ldots \to u_i$ be the path from the root of $\mathcal{T}$ to u_i. From their definitions, $R_{u_i} = \bar{X} - \sum_{\ell \leq i} s_w(p_{u_\ell})$. Hence, it follows that $\max_h\{|R_{u_i}[h]|\} = ||\bar{X} - \sum_{\ell \leq i} s_w(p_{u_\ell})||_\infty$, 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 $i \leq k$ let E_i be the maximum L_∞ error of all residual vectors found at level i. Let $E = \bigcup_{i \leq k} E_i$. 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 $E_m \leq \varepsilon$. Then we transfer all compressed pool values P_i for $i \leq m$, 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 $||X - X'||_\infty \leq \varepsilon$.

2.5 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.

2.5.1 Algorithmic Optimizations

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 **?**. 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 **?**, 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 $O(T)$.

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

Algorithm 2: FastDecompression

```
   Input  : P
   Output: X'
1  S_{-1} = 0;
2  for h = 0 to T - 1 do
3      i = L;
4      S_h = S_{h-1};
5      while h mod T/2^i == 0 AND i >= 0 do
6          j = h / (T/2^i);
7          S_h = S_h + p_{(i,j)} - p_{(i,j-1)};
8          i = i - 1;
9      X'[h] = S_h
10 return X';
```

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 $O(T)$.

2.5.2 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.

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

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.

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 2.2) 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'.

2.5.3 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.

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 2.5.1. 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.

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 $\max\{\max_{i \le j} X[i] - \min_{\ell \le j} X[\ell], \max_{i > j} X[i] - \min_{\ell > j} X[\ell]\}$ 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 $\max\{\sum_{i \le j}(X[i] - \hat{X}_{\le j})^2, \sum_{\ell > j}(X[\ell] - \hat{X}_{>j})^2\}$, where $\hat{X}_{\le j}$ 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.

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.

2.6 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 **???**. 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 ??? and the ZFP algorithm ?. 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 ? 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 ?. Similarly, multiresolution trees have been widely applied in approximate query processing where data are aggregated at hierarchy predicates ???. Similar "multiresolution" results for aggregate queries are seen in wavelet techniques for AQP ?, online aggregation ?, and sketching ?. 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 ?.

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 ? 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 **?**.

2.7 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 **?** repository. For HIRE and the relevant baselines, we use the Turbo Range Coder to encode the final codes into bytes **?**.

2.7.1 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 **?**, 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 **?** $46.2MB$; Individual household electric power consumption data set (IHEPC) $29.4MB$; Bitcoin heist ransomware address data set (BC) **?**, $50.3MB$. 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.

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.

This modification does not change our experiments and is simply done for consistency.

2.7.2 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 2.2.2 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) **?** to rearrange blocks of values into runs of the same symbol (i.e., integer), and then applies an arithmetic encoder **?** 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 **?**.
- **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 **?**. 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 **?**. 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 **?**.

- **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 **?**.

2.7.3 Performance Overview

Table 2.5 from Subsection 2.7.6 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.

2.7.4 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 2.1; 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 2.1: 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.

2.7.5 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 2.2.[5] The main driver of HIRE's significant performance advantage 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.

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.

	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 2.2: 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 2.3. 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]

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.

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

2.7.6 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 2.4: 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 2.4. 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.**

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 2.5. 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.

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

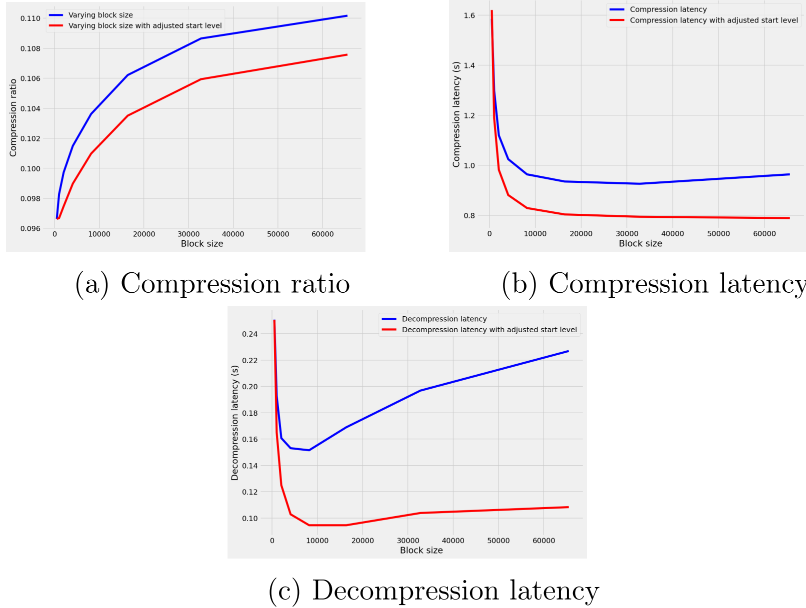
Table 2.5: Compression ratio for different resolutions on an edge device

2.7.7 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.

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



(a) Compression ratio (b) Compression latency

(c) Decompression latency

Figure 2.3: Breakdown of performance at different block sizes

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 2.3a 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 2.3b. 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

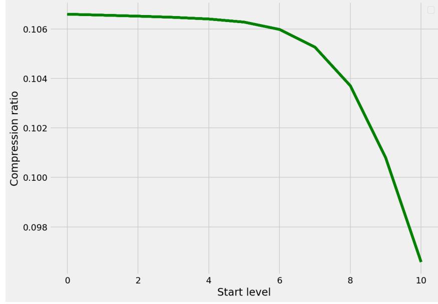


Figure 2.4: Compression ratio for different starting levels

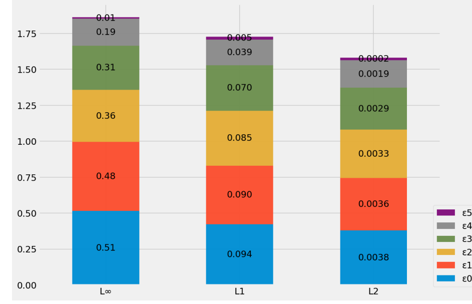


Figure 2.5: L_∞, L_1, L_2 errors for 6 levels

impede further scalability due to the fixed number of CPU cores. Finally, when analyzing the decompression latency shown in Figure 2.3c, 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.

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 2.4 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

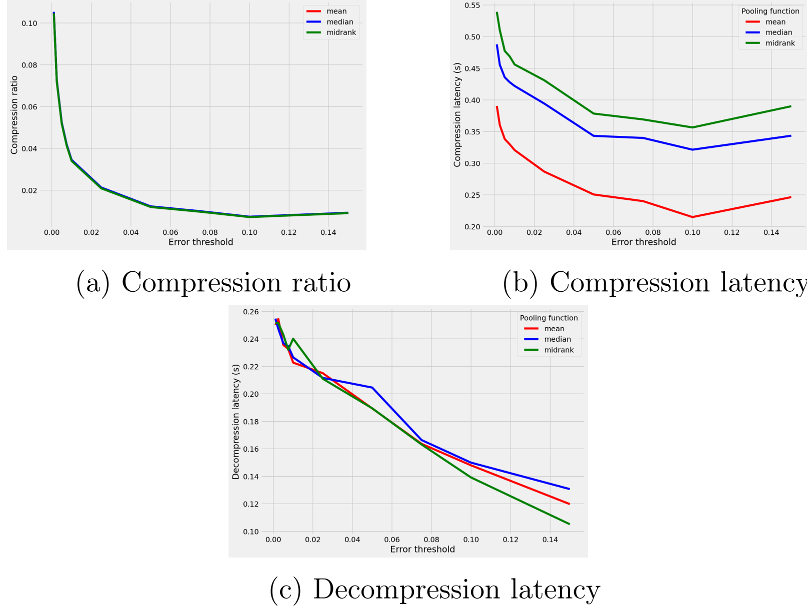
increases exponentially—specifically by a factor of 2—until we reach the upper bound of T values at the leaf nodes of the hierarchy.

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 2.4.1. In Figure 2.6a, we display the compression ratios at various error thresholds for the three pooling functions: mean, median, and midrank. In Figures 2.6b and 2.6c 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.

2.7.8 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 2.5.3. 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 2.5. 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.



(a) Compression ratio

(b) Compression latency

(c) Decompression latency

Figure 2.6: Breakdown of performance for three different pooling functions

Second, we implement a variation of HIRE that splits at the optimum location as described in Section 2.5.3. 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 of 0.01. The results are displayed in Table 2.6. 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 2.5.3. 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 2.6: 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.

2.8 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.

CHAPTER 3

RASTERSTORE: ADAPTIVE COMPRESSION FOR FULLY RASTERIZED GEOSPATIAL DATA

3.1 Introduction

From the early days of relational databases, supporting geospatial analytics has been an important research area. Motivated by applications such as ride-sharing and other location-based services, most geospatial databases are optimized for "vector data" that is data which consist of latitude, longitude (and perhaps altitude) points, polygons, and paths. In contrast, many do not support "rasterized data", which contains data organized like an image with discrete values in regularly-spaced pixels or voxels. Rasterized data is important for applications in satellite imaging, robotics, scientific imaging, and increasingly, applying machine learning over vector data.

Rasterized data is fundamentally array-structured data, but in most geospatial applications, the majority of this data is uninteresting. For example, in robotic mapping, one might be interested in identifying the location of an obstacle in a mostly empty room. A vector format is naturally compressed where only the relevant information is represented, but a rasterized format has to expend memory to store the empty pixels or voxels. Since rasterized data is often sensor collected, such data grows very rapidly even if much of it is uninteresting **?**.

Data compression **?** is a key tool to control the costs of rasterized data. Furthermore, lossy compression can allow users to trade-off data quality for improved storage and transfer performance. Traditionally, many simply apply image compression algorithms like JPEG **?** on such data . Firstly, while rasterized data resemble images, not all rasterized data are natural images, and thus, image codecs may elide key features valuable to analysis. Second, while image compression algorithms are highly effective at preserving perceptual similarity,

they do not offer strong pixel-wise guarantees on error rates. For lossy compression to be effective at a wide variety of rasterized data, it must offer a strong pixel-wise error guarantee; that is, maximum allowed error is bounded (hereafter called an ℓ_∞ guarantee). Maximum error guarantees are common in the context of both scientific data compression **????** and general time series codecs **??**.

To harmonize the need for ℓ_∞ guarantees with an effective codec that exploits the natural structure of rasterized data, we present RasterStore a compression framework that finds a good combination of spatial and pixel-wise error. In contrast to many ℓ_∞ preserving methods from scientific data compression (e.g., SZ **?** and ZFP **?**), expensive prediction and transformation steps are omitted from our framework. Additionally, RasterStore pushes the majority of computation toward the encoding stage of the pipeline and does not require equally-expensive decoding routines to invert decorrelation steps. Efficient decompression makes repeated decoding viable, a common requirement for machine learning pipelines that frequently move training data between disk and main memory.

Our approach addresses what we call the *spatial-pointwise precision tradeoff*, where the objective is to allocate error to competing spatial and pointwise dimensions. To illustrate what we mean by these two dimensions, consider the following example.

Define two blocks $A = [1.10, 1.14, 1.15, 1.16]$ and $B = [1.10, 2.51, 3.89, 4.26]$ from a 2D rasterized array that are both linearized for the sake of simplicity. We are given an ℓ_∞ requirement of $\epsilon = 0.05$.

For the first block, we notice that if we choose 1.15, all of the values in the block are contained in the interval 1.15 ± 0.05, and hence if we replace those three values with 1.15 we maintain the ℓ_∞ bound. In other words, this observation follows from the fact that $\max_i |a_i - 1.15| \leq 0.05$. This replacement strategy improves the redundancy in the block, since each value $a_i \in A$ will now be mapped to the same code.

On the other hand, it is clear that in block B, the same strategy will not work at the

strict guarantee of $\epsilon = 0.05$. The best strategy in this case is to simply quantize the values at the error $\epsilon = 0.05$ and pass the values to the entropy coder.

In many cases, however, the choice between spatial replacement and quantization will not be so clear-cut. Suppose that we combine the two blocks to form C = [1.10, 1.14, 1.15, 1.16, 1.10, 2.51, 3.89, 4.26]. In this instance, it is likely suboptimal to choose to exclusively apply either replacement or quantization, so it would make sense to try a combination of both strategies. Put simply, some of the error can be allocated to the replacement strategy and the remaining error can be allocated to quantization. This is precisely what is meant by the *spatial-pointwise precision tradeoff*.

Given a combined error threshold ϵ and a replacement allocation x RasterStore consists of the following steps:

1. **Blocking**: The 2D array is divided into multiple smaller blocks, which we call windows.
2. **Replacement**: In each window, a pivot value is selected, and all values in the window falling within $\pm x\epsilon$ of the pivot are replaced by the pivot value.
3. **Quantization**: Each modified window is quantized using the remaining error $(1-x)\epsilon$.
4. **Flattening**: The quantized windows are flattened (linearized) to form a 1D integer array.
5. **Entropy coding**: The resulting 1D array of integers is passed to an entropy coder that exploits redundancy over the discrete distribution of integer codes.

Note that blocking, flattening, and entropy coding are all lossless operations, while replacement and quantization are lossy operations. Each operation has multiple degrees of freedom.

3.2 Preliminaries

Below, we describe the main intuition behind our method. Raster data is represented as a multidimensional array of values, where each value is often associated with a spatial location

??. A common example of raster data is the LandSat satellite that collects hyper-spectral image data across the visible spectrum as well as several other frequency bands ?. A Landsat pixel corresponds to a location on the surface of the earth at a certain spatial resolution (e.g., 30x30 meters). For the sake of simplicity, we can consider a single frequency band such as the visible color red, and we will assume that the input data forms a square. In the running example, this yields a square matrix of pixels $\mathcal{A}$, where each pixel $\mathcal{A}_h$ represents the intensity of the color red at a certain location.

3.2.1 Compression Basics

A compression algorithm consists of an encoder and decoder. To mirror the terminology presented in later sections, we assume that the input to the compression algorithm is a square matrix $\mathcal{A} \in \mathbb{R}^{N \times N}$. The encoder (enc) produces a compressed representation $C_{\mathcal{A}}$ and the decoder (dec) returns a new version of the original matrix $\mathcal{A}'$:

$$C_{\mathcal{A}} = \mathsf{enc}(\mathcal{A}) \qquad \mathcal{A}' = \mathsf{dec}(C_{\mathcal{A}}) \tag{3.1}$$

In a lossless compression algorithm, it is always true that $\mathcal{A}' = \mathcal{A}$ but in a lossy compression algorithm such as HIRE it is only required that $\mathcal{A}' \approx \mathcal{A}$. Specifically, this means that $\mathcal{A}' = \mathcal{A} + \mathcal{E}$ for some error matrix $\mathcal{E}$.

There are many ways in which error can be introduced, which range from soft constraints on perceptual quality, as is common with images ?, to hard constraints on the pointwise error between any two values, as is common with scientific data ?. We restrict our discussion to the latter category of constraints and describe an effective algorithm that maintains a guarantee on the maximum pointwise error.

3.2.2 RasterStore Overview

Now moving on to our system, the user first specifies an error threshold ϵ, which is the pixel-wise ℓ_∞ error guarantee that must be met by the codec. In particular, the following must hold for any input:

$$||\text{vec}(\mathcal{A} - \mathcal{A}')||_\infty = \max_h |\mathcal{A}_h - \mathcal{A}'_h| \leq \epsilon \tag{3.2}$$

where vec($\cdot$) is the matrix vectorization operator and $\mathcal{A}'_h$ is any element of the decoded matrix that corresponds to element $\mathcal{A}_h$ in the input matrix.

Apart from standard entropy coding, there are two lossless steps in our framework: blocking and flattening. These two steps interact with each other to capture spatial redundancy. Blocking partitions the input matrix into a block matrix, where we call each block a *window*. Immediately prior to entropy coding, the flattening algorithm then defines how the blocks are to be linearized, effectively mapping the block matrix to a single vector. As will be shown in later sections, there are many possible flattening approaches.

At a high level, the encoding algorithm has two lossy steps that fully allocate the allowable ℓ_∞ error ϵ to the compressed representation. These two steps are replacement and quantization. Slightly abusing notation, for a given fraction $0 \leq x \leq 1$ the error quantity $x\epsilon$ is allocated to replacement and the remaining error $(1-x)\epsilon$ is allocated to quantization, so indeed $x\epsilon + (1-x)\epsilon = \epsilon$. Window replacement first deliberately chooses a *pivot* value from a particular block and then sets all values within a distance from the pivot of $\pm x\epsilon$ to the pivot value, increasing redundancy. The remaining error $(1-x)\epsilon$ is then used as a step size to quantize the result and produce integer codes.

In general, RasterStore codec combines the lossless and lossy operations that we described above to produce a decoding that has a strict pointwise guarantee on the error. While it is intuitive that the two lossy steps would preserve the desired ℓ_∞ error, we can easily prove that this is the case in the next section.

3.3 Our Method

In this section we describe our main lossy method. The principle is illustrated in Figure 3.1. Let $\mathcal{A} \in \mathbb{R}^{N \times N}$ be an array of real numbers. Without loss of generality, we assume that $\mathcal{A}$ is a square matrix; all our methods work for any general matrix. Let $t = \frac{N}{s}$. We partition $\mathcal{A}$ into $t^2 = \left(\frac{N}{s}\right)^2$ windows of size $s \times s$. Let $\mathcal{W} = \{W_{1,1}, \ldots, W_{t,t}\}$, where $W_{i,j} = \mathcal{A}[((i-1) \cdot t + 1) \times (i \cdot t), ((j-1) \cdot t + 1) \times (j \cdot t)]$.

The algorithm visits every window $W_{i,j}$ and applies our compression approach. More specifically, for every $W_{i,j}$ let $p_{i,j}$ be its pivot point, and $x_{i,j}$ be the ratio of the L_∞ error for window replacement. That is, the L_∞ error for window replacement is $x_{i,j} \cdot \varepsilon$ while the L_∞ error for quantization is $(1 - x_{i,j}) \cdot \varepsilon$. We describe how we choose $p_{i,j}$ and $x_{i,j}$ in the next Section.

We describe an algorithm to make two passes over the data to prepare for data compression. Given $W_{i,j}$, $p_{i,j}$, and $x_{i,j}$, the goal is to replace the input $W_{i,j}$ with $\bar{W}_{i,j}$ consisting of a set of integers that they will be used for compression. We go through all items in $W_{i,j}$ in any arbitrary order: For any $a_h \in W_{i,j}$ we check whether $|a_h - p_{i,j}| < x_{i,j}\varepsilon$. If this is true then we set $a' = p_{i,j}$. Otherwise we set $a'_h = a_h$. In the mean time, we keep track of the minimum value a'_h we encounter. Let $\mu_{i,j}$ be the minimum value of a'_h we find. Then we go over all items one more time to apply quantization: For every a'_h, we set $\bar{a}_h = \lfloor \frac{a'_h - \mu_{i,j}}{(1-x_{i,j}) \cdot \varepsilon} \rfloor$. In the end, we set $\bar{W}_{i,j} \in \mathbb{N}^{s \times s}$ to be the window similar to $W_{i,j}$ replacing the a_h values by the new computed $\bar{a}_h$ values. Finally, we set $\bar{\mathcal{A}}$ as the input to the data compression algorithm we use.

We make two passes over the input elements so the running time of the algorithm in every window is $O(s^2)$. Overall, the running time to construct all new windows $\bar{W}_{i,j}$, is linear with respect to the input size, i.e., $O(t^2 \cdot s^2) = O(N^2)$.

We state the fact that our algorithm guarantees the ℓ_∞ error ϵ in the following Proposition.

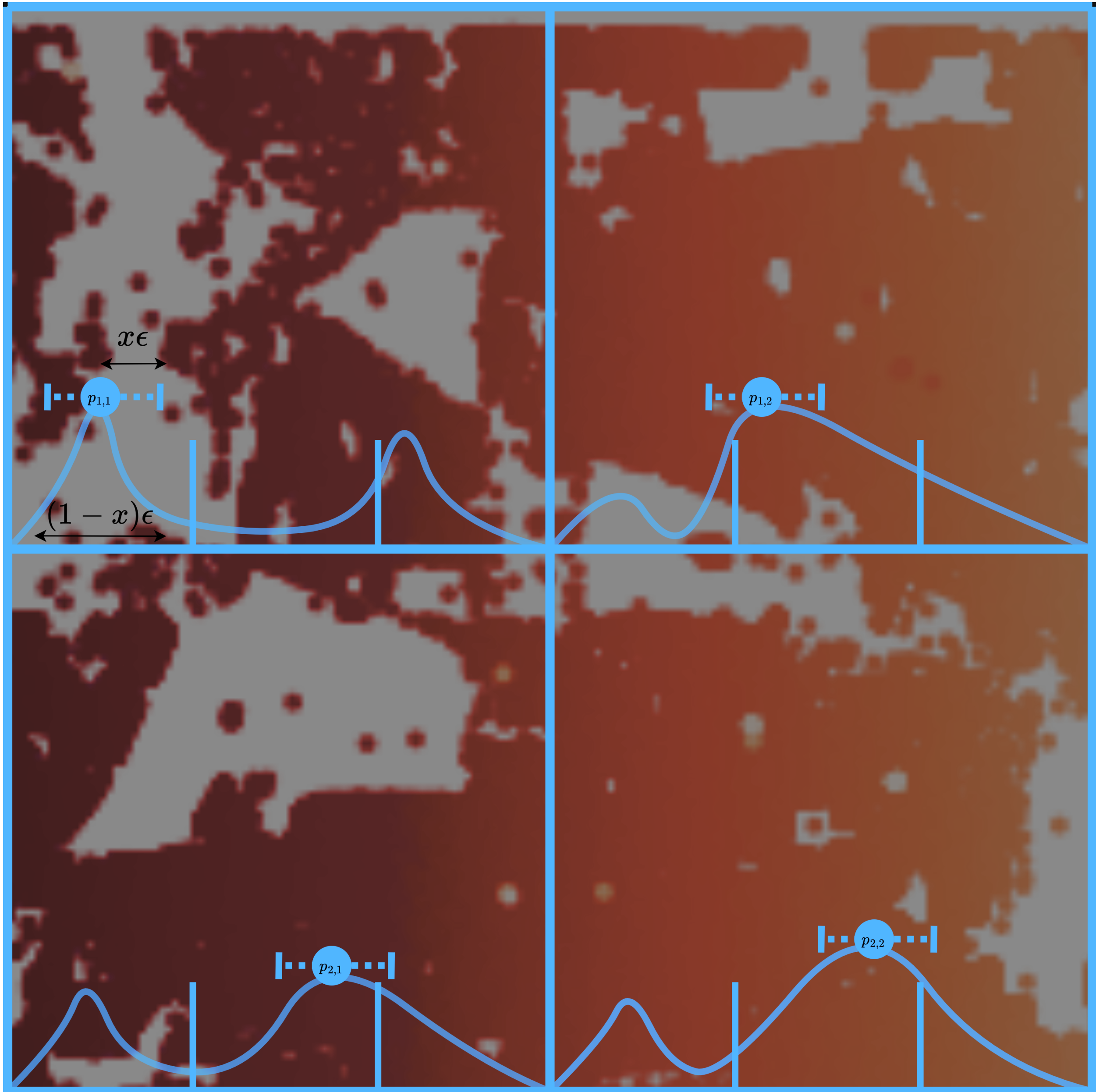


Figure 3.1: How RasterStore allocates error in a given window.

Given an ℓ_∞ error $\epsilon > 0$ and window replacement ratios $0 \leq x_{i,j} \leq 1$ for all i, j and any input matrix $\mathcal{A}$ the decoded matrix $\mathcal{A}'$ after applying window replacement and quantization has the following guarantee:

$$||\text{vec}(\mathcal{A} - \mathcal{A}')||_\infty = \max_{h \in [N^2]} |\mathcal{A}_h - \mathcal{A}'_h| \leq \epsilon$$

Proof. We simply need to show that for a given window, the two lossy operations from our codec preserve the ℓ_∞ error guarantee. Let $\mathcal{A}^w_{i,j} = R(\mathcal{A}_{i,j})$ be the result of applying window replacement with error $x_{i,j}\epsilon$. Let $\mathcal{A}' = Q^{-1}(Q(\mathcal{A}^w_{i,j}))$ be the result of quantizing (Q) the replacement array to integer codes with error $(1 - x_{i,j})\epsilon$ and then dequantizing (Q^{-1}) back to the reals. We observe that $||\text{vec}(\mathcal{A}_{i,j} - \mathcal{A}'_{i,j})||_\infty = ||\text{vec}((\mathcal{A}_{i,j} - \mathcal{A}^w_{i,j}) + (\mathcal{A}^w_{i,j} - \mathcal{A}'_{i,j}))||_\infty$. By construction of the algorithm described above $||\text{vec}(\mathcal{A}_{i,j} - \mathcal{A}'_{i,j})||_\infty \leq x_{i,j}\epsilon$ and $||\text{vec}(\mathcal{A}^w_{i,j} - \mathcal{A}'_{i,j})||_\infty \leq (1 - x_{i,j})\epsilon$. Since ℓ_∞ is a norm that satisfies the triangle inequality, we have $||\text{vec}((\mathcal{A}_{i,j} - \mathcal{A}^w_{i,j}) + (\mathcal{A}^w_{i,j} - \mathcal{A}'_{i,j}))||_\infty \leq ||\text{vec}(\mathcal{A}_{i,j} - \mathcal{A}^w_{i,j})||_\infty + ||\text{vec}(\mathcal{A}^w_{i,j} - \mathcal{A}'_{i,j})||_\infty \leq x_{i,j}\epsilon + (1 - x_{i,j})\epsilon = \epsilon$. Hence, the bound is satisfied for all windows $\mathcal{A}'_{i,j}$, so we conclude that the bound is satisfied for the full decoded matrix $\mathcal{A}'$ as required.

3.4 Optimizations

In this section, we describe various methods on how to optimally choose the values p and x in order to achieve the best compression ratio. Unfortunately, the actual compression ratio highly depends on the structure of the dataset and the entropy coder we use. In order to select the best parameters p and x, we introduce some metrics to estimate how compressible the data is. We use the variance, entropy, mode, and combinations of them to estimate the compression ratio. Let $A = \{a_1, \ldots, a_n\}$ be a list of n integer numbers. The list A corresponds to the items stored in a window $\bar{W}_{i,j}$.

- Variance: $Var(A) = \frac{1}{n^2} \sum_{h \in [1,n]} a_h^2 - \frac{1}{n^2} \left(\sum_{h \in [1,n]} a_h \right)^2$. Intuitively, the smaller the

variance, the more compressible the data is.

- Entropy: Let $D(A)$ be the set of distinct integers in A, $m = |D(A)|$, and let f_h be the frequency of item $a_h \in D(A)$. The entropy of A is defined as $H(A) = -\sum_{h\in[1,m]} f_h \log f_h$. Intuitively, the smaller the entropy, the more compressible the data is.
- Mode: $M(A) = \arg\max_{a_h \in D(A)} f_h$. Intuitively, the larger the mode is, the more compressible the data is.

3.4.1 Statistical pivot selection

In this section, we define the statistical pivot selection problem and state several practical algorithms for choosing a suitable pivot in a given window. We focus on the problem of finding an estimate for the mode of a window without explicitly requiring that the values are discretized by the quantization scheme, in contrast to what will be presented in later sections. We first assume that each window $W_{i,j}$ contains samples from a continuous distribution with some unknown probability density function $f_{i,j}(y)$ that can be approximated by an estimate $\hat{f}_{i,j}(y)$. This is a well-studied statistical problem called (non-parametric) density estimation **???**.

Formally, the objective in the context of compression is to choose an optimal pivot $p_{i,j}$ such that the probability density $\int_I \hat{f}_{i,j}(y)dy$ on the interval $I = [p_{i,j} - x_{i,j}\epsilon, p_{i,j} + x_{i,j}\epsilon]$ is maximized. Following the algorithm described in Section 3.3, this corresponds to maximizing the number of elements in a particular window that are contained in the interval I.

However, the interval may not be known in advance, so it suffices to consider the problem of finding a pivot that falls in a high density region of the probability distribution without explicit prior knowledge of the interval length $x_{i,j}\epsilon$. Concretely, the pivot is to be chosen so many values will fall near the pivot, regardless of the length of the interval.

What we seek is a pivot that maximizes an estimate $\hat{f}_{i,j}$ constructed from the values in

a given window $\mathcal{A}_{i,j}$ for the window probability density function $f_{i,j}(y)$. This optimal pivot is simply an estimator of a *mode*. Crucially, the mode estimate must be found in a non-parametric fashion, so the pivot selection procedure can use the actual values in a window as input. Without loss of generality, we can constrain the pivot to the set $\mathcal{A}_{i,j}$. In particular, it is satisfactory to find a value $a_h \in \mathcal{A}_{i,j}$ that is close to a mode estimate. This leads us to the following canonical form for statistical pivot selection:

$$p_{i,j} = \arg\max_{a_h \in \mathcal{A}_{i,j}} \hat{f}_{i,j}(a_h) \tag{3.3}$$

where again $\hat{f}_{i,j}$ is some density estimate.

Practical algorithms

Below, we list several practical algorithms based on non-parametric density estimates that can be used to select a pivot. The list is roughly in ascending order of latency overhead.

- **First element**: Choose the first element in the window. This is an efficient systematic sampling method that does not require random number generation.
- **Random element**: Choose an element uniformly at random from the window by generating a random number for each dimension.
- **Histogram**: Construct an equi-width histogram $\mathbb{H}$ with m bins $B = \{B_1, B_2, \ldots, B_m\}$ each having a corresponding bin density $p(B_r)$. Find $B_o = \arg\max_{B_r \in B} p(B_r)$ and select any element that satisfies $a_h \in B_o$. A histogram can be seen as a lower resolution version of quantization.
- **Kernel density estimate**: Using a valid kernel K the kernel density estimate $\hat{f}(a) = \frac{1}{s^2 z} \sum_{a_h \in \mathcal{A}_{i,j}} K(\frac{a-a_h}{z})$ is first computed, where z is the bandwidth. Then, the element $\arg\max_{a_h \in \mathcal{A}_{i,j}} \hat{f}(a_h)$ is chosen.

Note that at larger window sizes, the histogram and kernel density estimate can be constructed from a random sample drawn uniformly from the window, instead of the entire window. Sampling would likely reduce the overhead imposed by these more complex methods, while still ensuring that the general distribution is closely approximated.

3.4.2 Pivot selection given the ratio

Next, we assume that we know the ratio $x_{i,j}$ for a window $W_{i,j}$. We show how we can find the optimum pivot $p := p_{i,j}$ in near-linear time.

First, we construct $A = \{a_1, \ldots, a_m\}$, a sorted array ($a_h \leq a_k$ for $h < k$) that contains all items in $W_{i,j}$. We visit each $a_h \in A$, and we compute its quantization bin $b_h = \lfloor \frac{a_h - a_1}{(1-x)\varepsilon} \rfloor$, without considering the pivot. For every bin b_h we compute the number of items it contains, denoted by f_h. We also compute $S_1 = \sum_h f_h \cdot b_h$ and $S_2 = \sum_h (f_h \cdot b_h)^2$.

For simplicity, we search for the optimum pivot satisfying $p > a_1 + x \cdot \varepsilon$; the minimum value a_1 remains always the same. The algorithm can be extended to search for $p \leq a_1 + x \cdot \varepsilon$, however it becomes more tedious.

Let $I = [a_1, a_1 + x \cdot \varepsilon]$ be an interval in $\mathbb{R}^1$. We move this interval and we use its p_I to find all possible pivots. Let V, E, M be the current variance, entropy, and mode, respectively, of $W_{i,j}$, with pivot p_I, and ratio x. Let b_s be the bin that the current pivot p_I belongs to, i.e., $b_s = \lfloor \frac{p_I - a_1}{(1-x)\varepsilon} \rfloor$. Let also $p_s = |I \cap A|$, i.e., the number of elements in I after their quantization with pivot p_I. If b_ℓ and b_r are the bins that the left and right endpoints of I belongs to, let ℓ_I be the number of elements in the bin b_ℓ that are contained in I at the left of p_I and let r_I be the number of elements in the bin b_r that are contained in I and at the right of p_I. Finally, let T be a max-heap that stores each bin b_h with weight f_h. This max-heap is only used to find the mode.

We show how to update the values V, E, M efficiently each time that we try the next possible pivot. After visiting all possible pivot points, we return the one that leads to either

the lowest variance, the lowest entropy, or the highest mode. We move the interval I to the right until one of its endpoints intersects an item from A or one of its endpoints intersects the boundary of a bin or its center p_I intersects the boundary of a bin.

First, we consider the case where p_I intersects the right boundary of b_s. Equivalently, this is the left boundary of b_{s+1}.

If $b_r > b_{s+1}$ and $b_\ell < b_s$ then we update $S_2 = S_2 - (p_s \cdot b_s)^2 + (p_s \cdot b_{s+1})^2$, $S_1 = S_1 - p_s \cdot b_s + p_s \cdot b_{s+1}$, and $V = \frac{1}{m} S_2 - \frac{1}{m^2} S_1^2$. We also remove the item b_s from T and insert the item b_{s+1} with weight p_s. The mode M is updated by the top-weight in T. Finally, we set $b_s = b_{s+1}$. The entropy E remains unchanged.

If $b_r = b_{s+1}$ and $b_\ell = b_s$ then we update $S_2 = S_2 - ((p_s + f_s - \ell_I) \cdot b_s)^2 - ((f_{s+1} - r_I) \cdot b_{s+1})^2 + ((f_s - \ell_I) \cdot b_s)^2 + ((p_s + f_{s+1} - r_I) \cdot b_{s+1})^2$, $S_1 = S_1 - (p_s + f_s - \ell_I) \cdot b_s - (f_{s+1} - r_I) \cdot b_{s+1} + (f_s - \ell_I) \cdot b_s + (p_s + f_{s+1} - r_I) \cdot b_{s+1}$, and $V = \frac{1}{m} S_2 - \frac{1}{m^2} S_1^2$. We also update the weights of the items b_s and b_{s+1} from T to be $f_s - \ell_I$ and $f_{s+1} + \ell_I$. The mode M is updated by the top-weight in T. We also update the entropy $E = E - \frac{f_s + r_I}{m} \log \frac{m}{f_s + r_I} - \frac{f_{s+1} - r_I}{m} \log \frac{m}{f_s - r_I} + \frac{f_s - \ell_I}{m} \log \frac{m}{f_s - \ell_I} + \frac{f_{s+1} + \ell_I}{m} \log \frac{m}{f_{s+1} + \ell_I}$. Finally, we set $b_s = b_{s+1}$.

Equivalently, we handle the cases i) $b_r > b_{s+1}$, $b_\ell = b_s$, ii) $b_r = b_{s+1}$, $b_\ell < b_s$.

Next, we consider the case where an endpoint of I intersects the boundary of a bin. Without loss of generality assume that the left endpoint of I intersects the right boundary of a bin $b_h = b_\ell$. If $b_\ell < b_s - 1$ then $\ell_I = f_{h+1}$. Otherwise, we set $\ell_I = |A \cap [b_{h+1}, p_I]|$. We know $A \cap [b_{h+1}, p_I]$ by i) maintaining the number of elements between the left endpoint of I and p_I with straightforward manner, or by ii) construct a search binary tree over A and ask a $O(\log(n))$ time count query to compute it. In the end we update $b_\ell = b_{h+1}$. Equivalently, we handle the case where the right endpoint of I intersects the right boundary of a bin b_h.

Finally, we assume that an endpoint of I intersects an element $a_k \in A$ from bin b_k. Without loss of generality we assume that a_k intersect the right endpoint of I and $b_\ell < b_s$. We need to update V, E, M adding a_k in I. We have, $S_1 = S_1 - b_k \cdot (f_k - r_I) - b_s \cdot p_s + b_k \cdot (f_k -$

$r_I-1)+b_s\cdot(p_s+1)$, and $S_2 = S_2-(b_k\cdot(f_k-r_I))^2-(b_s\cdot p_s)^2+(b_k\cdot(f_k-r_I-1))^2+(b_s\cdot(p_s+1))^2$. We update, $V = \frac{1}{m}S_2\frac{1}{m^2}S_1^2$, and $E = E-\frac{f_k-r_I}{m}\log\frac{m}{f_k-r_I}-\frac{p_s}{m}\log\frac{m}{p_s}+\frac{f_k-r_I-1}{m}\log\frac{m}{f_k-r_I-1}+\frac{p_s+1}{m}\frac{m}{p_s+1}$. In order to update the mode, we update T. More specifically, we update the weight of b_s to be p_s+1 and we update the weight of b_k to be f_k-r_I-1. The mode M is updated by the top-weight in T. In the end, we update $r_I = r_I+1$ and $p_s = p_s+1$. Equivalently, we can update all the variables if $b_\ell = b_s$. The algorithm is also almost identical when a_k intersects the left endpoint of I.

From $W_{i,j}$, we construct the sorted array A in $O(m\log m)$ time. We initialize the bins, the interval I and the variables we use in $O(m)$ time. Then for every algorithm we make one pass over all items in A. For each item, we spend $O(1)$ time to update V and E and $O(\log m)$ time to update T for the mode M. Hence, for a window $W_{i,j}$ with m items we can find the optimum pivot in $O(m)$ time. Overall, we compute the optimum pivot for every window $W_{i,j}$ in $O(N^2\log N)$ time.

Putting everything together we have the next theorem. Let A be an array of m elements and let x be the window replacement ratio. There exists an algorithm to find the pivot that minimizes either the variance or the entropy, or maximizes the mode in $O(m\log m)$ time.

3.4.3 Ratio selection given the pivot

In this section we describe how to select the ratio x assuming that the pivot point p is given.

First, we define some additional useful notation. Let $H(p,x,A)$, $Var(p,x,A)$, $M(p,x,A)$ be the entropy, variance, and mode respectively for A if the ratio is x and pivot is p. In this subsection, p is fixed so we can skip it from the notation.

For every metric, we make an assumption that the value of the metric does not change drastically for two distinct but close values of x. This is a realistic assumption that we also observed from our experiments in real data (see also Section 3.5). Formally, for any $\gamma\in(0,1)$ and $\delta\in(0,1)$, a function g is called (γ,δ)-bounded, if $\frac{\max\{g(x,A),g((1+\gamma)x,A)\}}{\min\{g(x,A),g((1+\gamma)x+\delta,A)\}}\leq$

$1 + r(\gamma, \delta)$, where $g(\cdot, \cdot)$ is either the variance, the entropy or the mode, and $r(\gamma, \delta)$ is a small approximation factor that depends on γ and δ. If $\gamma = \delta$, we write $r(\gamma)$.

The main idea is to find an approximation of the best ratio by trying different values of x. First, we sort all points in A. Then, we discretize x's domain, i.e., the range $[0, 1]$, to the values $S = \{0, \delta, (1+\gamma)\delta, (1+\gamma)^2\delta, \ldots, 1\}$. We note that $|S| = O(\frac{1}{\gamma} \log \frac{1}{\delta})$. For every possible $s \in S$, we run the algorithm from the previous subsection to compute $g(p, s, A)$ (notice that we know both parameters pivot p and ratio s.). In the end, we return the the ratio $x \in S$ with the minimum $g(p, x, A)$ value if g represents the entropy or the variance, or the maximum $g(p, x, A)$ if g represents the mode.

Let x^* be the real optimum ratio for the function g. If $x^* \in [0, \delta]$ There exists a value $\hat{x} \in S$ such that $\hat{x} - x^* \leq \delta$, while if $x^* \in (\delta, 1]$ there exists a value $\hat{x} \in S$ such that $\hat{x}/x^* \leq 1 + \gamma$. In any case, for entropy and variance, it holds that there exists $\hat{x} \in S$ such that $g(p, \hat{x}, A) \leq (1 + r(\gamma, \delta))g(p, x^*, A)$. For mode, it holds that there exists $\hat{x} \in S$ such that $g(p, \hat{x}, A) \geq \frac{1}{1+r(\gamma,\delta)} g(p, x^*, A)$

We need $O(m \log m)$ time to sort A. Then using the algorithms from the previous subsection for every $s \in S$ we compute $g(p, s, A)$ in $O(m)$ time. Overall the algorithm runs in $O(m(\log m + \frac{1}{\gamma} \log \frac{1}{\delta}))$ time.

Let A be an array of m elements and let p be the pivot point. If the entropy, variance, and mode are (γ, δ)-bounded functions for $\gamma, \delta \in (0, 1)$, then there exists an algorithm to find an $(1 + r(\gamma, \delta))$-approximation of the optimum variance, entropy, or mode in $O(m(\log m + \frac{1}{\gamma} \log \frac{1}{\delta}))$ time.

3.4.4 Pivot and ratio selection

In this section we describe an efficient algorithm to find both the optimum x and p that optimize the data compression. We focus on optimizing the entropy; in the end we briefly describe how to extend for variance and mode.

We note that the algorithm of the previous subsection can be applied here, straightforwardly. Indeed, for every possible pivot point $p \in A$ we run the algorithm finding the best x given p. While, this algorithm returns a $r(\gamma, \delta)$ approximation, it is quadratic in the size of A, which is m. There are $O(m)$ possible pivot points and each execution of the previous algorithm takes $O(m(\log m + \frac{1}{\gamma}\log\frac{1}{\delta}))$. In total, such this algorithm runs in $O(m^2(\log m + \frac{1}{\gamma}\log\frac{1}{\delta}))$ time. This algorithm is not scalable in large tables. Next, we propose an efficient (near-linear time) randomized algorithm for finding the best pivot and ratio.

Similarly to the straightforward algorithm, our efficient algorithm tries all possible pivot points. For every pivot, we try all discrete values of ratios in S. However, for every pivot p and ratio $x \in S$, we estimate (with high probability) the entropy efficiently without visiting all elements in A.

Given a pivot p and a ratio x, the goal is to estimate the entropy $H(p, x, A)$. We use the ideas estimating the entropy from **????**. More specifically, as shown in **?**,we can get a multiplicative $1 + \gamma$ approximation of $H(p, x, A)$ for any p and any x, if we have a data structure $\mathcal{D}$ that can answer efficiently the following queries: i) Get a random sample a_h from A, ii) given an item $a_h \in A$, identify its bucket b_h, and iii) count the number of elements in a bin b_h, i.e., if I_h is the interval representing bin b_h then the data structure should return $|A \cap I_h|$. Given A, let P be the preprocessing time to construct $\mathcal{D}$ and Q be the query time to answer i), ii), iii) queries. Given p, x, we can get an $1+\gamma$ multiplicative approximation of $H(p, x, A)$ in $O(Q\frac{\log^2 m}{\gamma^2})$ time with probability at least $\frac{1}{m}$.

Our data structure $\mathcal{D}$ consists of a binary search tree $\mathcal{T}$ over A. Every element in A is stored in a distinct leaf node of $\mathcal{T}$, and every node u of $\mathcal{T}$ stores m_u the number of elements in the leaf nodes of the subtree rooted at m_u. We also store the minimum element a_1 in A. This tree has $O(m)$ space and it is constructed in $O(m \log m)$ time.

We show how $\mathcal{T}$ can be used to answer the queries i, ii, iii given a pivot p and a ratio x. For query i), it is known that the counters in the nodes of $\mathcal{T}$ can be used to return a

random sample of A in $O(\log m)$ time. Let v, w be the children of a node u. Then we return a point in the subtree of v with probability $m_v/(m_v + m_w)$, and a point from the subtree of w with probability $m_w/(m_v + m_w)$. For query ii), given a sample a_h we identify its bin b_h as follows. If $|p - a_h| \leq x\varepsilon$ then a_h has the same bucket with the pivot point p. Hence we return $b_h = \lfloor \frac{p-a_1}{(1-x)\varepsilon} \rfloor$. If $|p - a_h| > x\varepsilon$ we return $b_h = \lfloor \frac{a_h - a_1}{(1-x)\varepsilon} \rfloor$. The running time is $O(1)$. Finally for query iii) we use $\mathcal{T}$ to count the number of elements in bin b_h. Let b_p be p's bin. If $b_h \neq b_p$ then we define the interval $I = [b_h(1-x)\varepsilon, (b_h+1)(1-x)\varepsilon] \setminus [p - x\varepsilon, p + x\varepsilon]$. We run a counting query on $\mathcal{T}$ using the query interval I. The result is the number of elements in bin b_h. If $b_h = b_p$ then we define the interval $I = [b_h(1-x)\varepsilon, (b_h+1)(1-x)\varepsilon] \cup [p - x\varepsilon, p + x\varepsilon]$. We run a counting query on $\mathcal{T}$ using the query interval I. The result is the number of elements in bin b_h. In both cases, the running time is $O(\log m)$.

Overall, the query time of every query on $\mathcal{T}$ takes $O(\log n)$ time. Putting everything together, we conclude with the next theorem.

Let A be an array of m elements and let p be the pivot point. If the entropy, is a (γ, δ)-bounded functions for $\gamma, \delta \in (0, 1)$, then there exists an algorithm to find a multiplicative $(1+\gamma)(1+r(\gamma, \delta))$-approximation of the optimum entropy in $O(m \log m + \frac{\log^3 m}{\gamma^3} \log \frac{1}{\delta})$ time. The algorithm returns a correct result with probability at least $1 - 1/m$.

If we use the variance or the mode instead of entropy, we can still use the data structure $\mathcal{D}$ to estimate the optimum values using random samples. However, we cannot get the result with high probability as we had for the entropy. Instead, we return an approximation with confidence interval bounds.

3.5 Experiments

We conducted all of the experiments on an Intel NUC with a dual-core 2.30 GHz i3-6100U processor, 16GB RAM, and a 256GB SSD. All implementations were done in Python 3.11. All different compression techniques were applied to 4 different datasets generated from the

city of Chicago data portal. For SR and QTRC, Turbo Range Coder was chosen as the downstream compressor to encode the final codes into bytes **?**, whereas Gzip was chosen for QGzip.

3.5.1 Datasets

Our datasets are all from the city of Chicago data portal. The other four datasets were obtained from the city of Chicago which has its own data portal **?**. We downloaded data pertaining parks locations, energy consumption, Sidewalk Cafe permits and Array of things locations. Then we proceeded to rasterize the data into 1024x1024 blocks in order to maintain consistency among the experiments. Each of those files is 5MB in size approximately.

3.5.2 Baselines

Our baselines consist of both standard compression algorithms used for multiple file formats and scientific data compression algorithms used on research labs for large scale data storage. We briefly describe each of them below:

- **Identity Gzip (IdG)**: Lossless compression baseline; we apply Gzip to an array of numbers represented as floating point values.
- **Quantize Gzip (QGZ)**: We convert each floating point number to an integer according to a user-defined error threshold, thereby saving exponent and mantissa bits (see 2.2.2 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. We apply Gzip as the downstream compressor for the final step.
- **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) **?** to rearrange blocks of values into runs of the same symbol (i.e., integer), and then applies an arithmetic

- **Fixed-Rate Compressed Floating-Point Arrays (ZFP)**: It maps 4^d values in d dimensions to a fixed number of bits per block, which is chosen by the user.**?**

3.5.3 Main Experiments

The main experiments consist of comparing the compression ratio, compression latency and decompression latency of all baselines against our method. We evaluate the aforementioned metrics in all datasets previously described and on the following l_∞ error thresholds: (0.15,0.1,0.05,0.01,0.005,0.001). They are meant the encapsulate real world use scenarios.

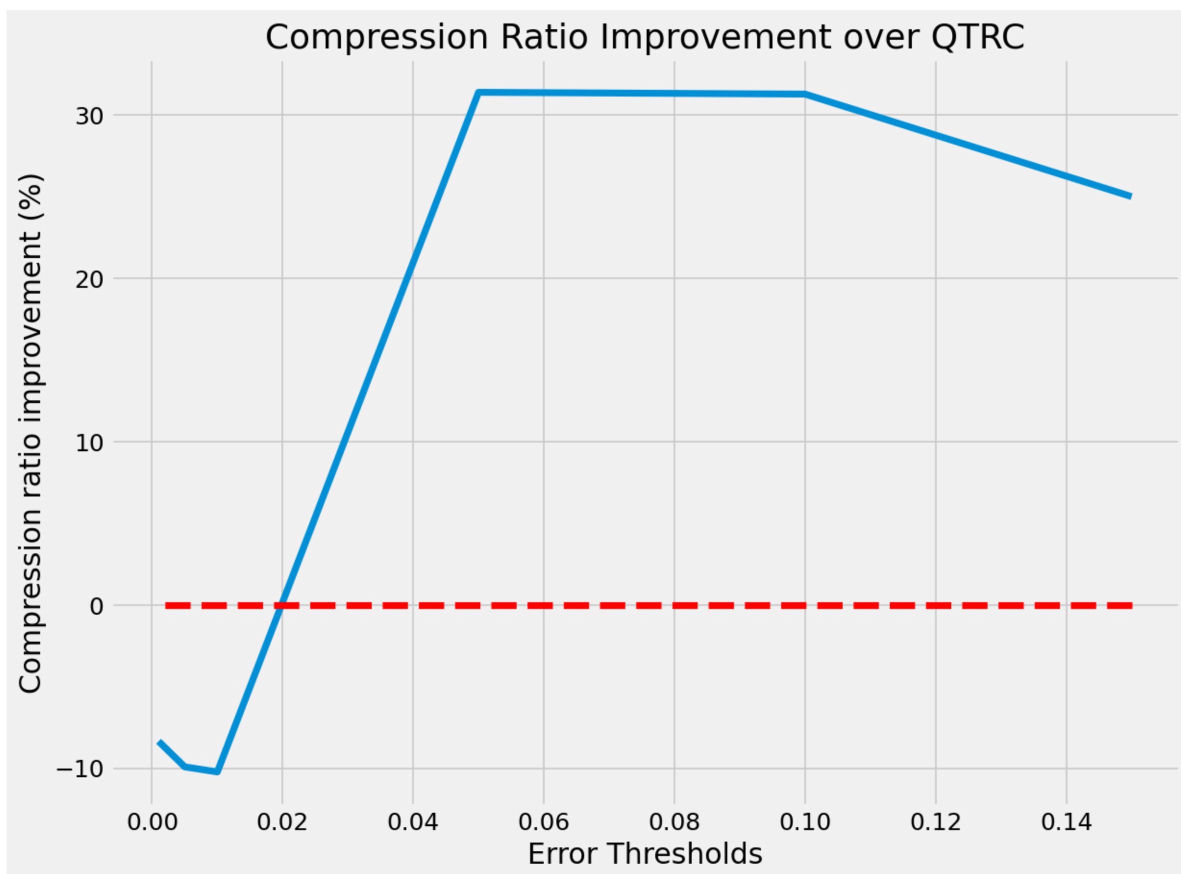


Figure 3.2: Compression ratio improvement over QTRC

We start by showing our method's improvement over QTRC which is the degenerate case of our solution when the window is set to use 0% of the l_∞ error budget. We notice on 3.2 that for errors larger than or qual to 0.01 we are able to generate significant improvements in compression ratio, upwards to 33% on a sample data set (park distance). Most of the improvement is derived from mapping neighboring points to the same quantization bucket when applying the window replacement, therefore increasing the run length on the subsequent downstream compressor.

On table 3.1 we see the results of our method and the competing ones. We notice that our method is able to produce the best compression ratio in most scenarios, achieving upwards to 35% improvements when compared to the second best baseline. It once again works more effectively on higher error thresholds, due to the size of the quantization buckets. One noticeable exception is the energy dataset, where our window method only improves upon the QTRC baseline on the higher error threshold 15%. That result is due to the higher variability of that dataset where neighboring units might have completely different energy consumption therefore mitigating the spatial effects of agglomeration.

We it comes to compression latency we observe a different trend. On 3.2 we notice that ZFP has compression latencies an order of magnitude faster than our method. That basically comes down to their highly optimized C++ code and the simpler operations performed during the block assignments and error allocation. When compared to QTRC we observe a slight increase in compression latency, but in the worst case scenario it adds roughly 15% to the already low latency baseline.

Finally when it comes to decompression latency we observe values on par to the highly optimized ZFP and similar (if not identical) to QTRC. Our method does not add any steps to QTRC when it comes to decompression, so it essentially conducts the same number of operations. Since we do not make any estimations during the process, we are able to match even ZFP when retrieving the data in its original format.

Data	ϵ	**RS**	**QTRC**	**GZip**	**QGzip**	**ZFP**
Park	0.15	0.0152	0.0190	58.77	0.0383	12.39
	0.1	0.0164	0.0216	58.77	0.0434	13.26
	0.05	0.0261	0.0343	58.77	0.0693	14.14
	0.01	0.0997	0.0895	58.77	0.1486	16.01
	0.005	0.1661	0.1496	58.77	0.1541	16.99
	0.001	0.5013	0.4594	58.77	0.7447	19.20
Energy	0.15	0.0508	0.0512	33.93	0.0593	13.56
	0.1	0.0514	0.0492	33.93	0.0589	14.38
	0.05	0.0521	0.0533	33.93	0.0752	15.19
	0.01	0.0632	0.0612	33.93	0.098	16.82
	0.005	0.0737	0.0718	33.93	0.0826	17.64
	0.001	0.1729	0.1586	33.93	0.2197	20.63
SW	0.15	0.0145	0.0165	47.481	0.0405	13.37
	0.1	0.0154	0.0185	47.486	0.0443	14.18
	0.05	0.0176	0.0234	47.48	0.0602	14.99
	0.01	0.0607	0.0625	47.48	0.102	16.62
	0.005	0.1200	0.113	47.481	0.105	17.44
	0.001	0.3486	0.338	47.48	0.4086	19.57
AOT	0.15	0.0819	0.0823	3762	0.0797	14.07
	0.1	0.0842	0.0824	37.62	0.0751	14.89
	0.05	0.0842	0.0845	37.62	0.0953	15.72
	0.01	0.0910	0.0930	37.62	0.122	17.37
	0.005	0.1008	0.1018	37.62	0.0963	18.34
	0.001	0.2066	0.1924	37.62	0.2293	21.78

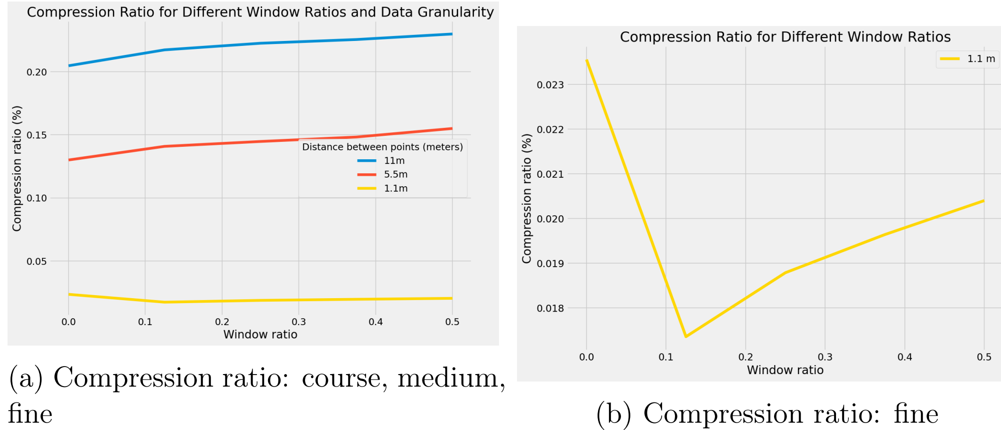
Table 3.1: Compression ratio for different error thresholds (%)

Data	ϵ	**RS**	**QTRC**	**GZip**	**QGzip**	**ZFP**
Park	0.15	0.1549	0.1318	0.3749	0.0200	0.0421
	0.1	0.1441	0.1312	0.3753	0.0229	0.0307
	0.05	0.1475	0.1460	0.3728	0.0249	0.0338
	0.01	0.1441	0.1325	0.3716	0.0489	0.0341
	0.005	0.1456	0.1354	0.3762	0.0679	0.0358
	0.001	0.1521	0.1403	0.3736	0.0548	0.0405
Energy	0.15	0.1597	0.1388	0.7155	0.0247	0.0594
	0.1	0.1453	0.1328	0.7203	0.0309	0.0341
	0.05	0.1501	0.1410	0.7187	0.0260	0.0374
	0.01	0.1483	0.1391	0.7652	0.0347	0.0378
	0.005	0.1476	0.1373	0.7186	0.0331	0.0413
	0.001	0.1515	0.1392	0.7190	0.0387	0.0463
SW	0.15	0.1536	0.1551	0.5998	0.0132	0.0344
	0.1	0.1422	0.1346	0.6044	0.0175	0.0352
	0.05	0.1464	0.1359	0.5975	0.0162	0.0343
	0.01	0.1497	0.1333	0.5977	0.0274	0.0384
	0.005	0.1397	0.1312	0.5992	0.0306	0.0383
	0.001	0.1451	0.1367	0.5969	0.0430	0.0418
AOT	0.15	0.1620	0.1320	0.5648	0.0332	0.0338
	0.1	0.1464	0.1380	0.5661	0.0394	0.0354
	0.05	0.1481	0.1395	0.5641	0.0307	0.0357
	0.01	0.1461	0.1349	0.6097	0.0439	0.0408
	0.005	0.1416	0.1322	0.5661	0.0380	0.0400
	0.001	0.1456	0.1353	0.5649	0.0421	0.0458

Table 3.2: Compression latency for different error thresholds (s)

Data	ϵ	RS	QTRC	GZip	QGzip	ZFP
Park	0.15	0.02602	0.0206	0.0780	0.01089	0.0245
	0.1	0.02039	0.0216	0.0733	0.01086	0.0175
	0.05	0.0202	0.0216	0.0833	0.0108	0.0182
	0.01	0.0207	0.0207	0.0647	0.0117	0.0205
	0.005	0.0212	0.0213	0.0717	0.0108	0.0217
	0.001	0.0283	0.0256	0.0727	0.0108	0.0240
Energy	0.15	0.0257	0.0201	0.0639	0.0116	0.0292
	0.1	0.0199	0.0207	0.0738	0.0109	0.0177
	0.05	0.0203	0.0204	0.0730	0.0109	0.0187
	0.01	0.0202	0.0203	0.0684	0.0108	0.0200
	0.005	0.0208	0.0212	0.0838	0.0117	0.0211
	0.001	0.0236	0.0235	0.0768	0.01090	0.0255
SW	0.15	0.0192	0.0179	0.0749	0.0075	0.0166
	0.1	0.0181	0.01802	0.0923	0.0075	0.0179
	0.05	0.0175	0.0188	0.0784	0.0092	0.0192
	0.01	0.0177	0.0194	0.0734	0.0076	0.0206
	0.005	0.0182	0.0186	0.0732	0.0074	0.0214
	0.001	0.0218	0.0220	0.0749	0.0075	0.0246
AOT	0.15	0.0269	0.0202	0.0557	0.0108	0.0178
	0.1	0.0206	0.0209	0.0750	0.0107	0.0188
	0.05	0.0201	0.0203	0.0694	0.0107	0.0186
	0.01	0.0206	0.0206	0.0589	0.0108	0.0207
	0.005	0.0207	0.0212	0.0628	0.0108	0.0223
	0.001	0.0235	0.0241	0.0679	0.0108	0.0270

Table 3.3: Decompression latency for different error thresholds (s)



(a) Compression ratio: course, medium, fine

(b) Compression ratio: fine

Figure 3.3: Breakdown of performance for several different data resolutions

3.5.4 Micro-benchmarks

We ran several different experiments in order to understand our compression method's most important hyperparameters: window size, pivot selection and error allocation. Additionally, we tested different flattening techniques to evaluate their corresponding performance. All experiments in this subsection were performed on the rasterized parks data set and the $L\infty$ error was set to 10%.

Raster Data Resolution

When rasterizing a data set, one needs to choose the appropriate resolution of the data that will be stored. In this context this is a trade-off between the precision and storage, as one increases the precision of the data being stored, there will be more points representing the same space and therefore it will require more storage space. Furthermore, we can always upsample from a finer grid to a coarse, but the other direction will involve an inference step with regards to the points being estimated. Therefore, whenever possible, we want to store data in the finer possible granularity in order to preserve information. We analyse the performance of our method on 3 different rasterized grids [1.1m, 5.5m, 11m] of the same data set. The values represent the distance between 2 points in the final data set.

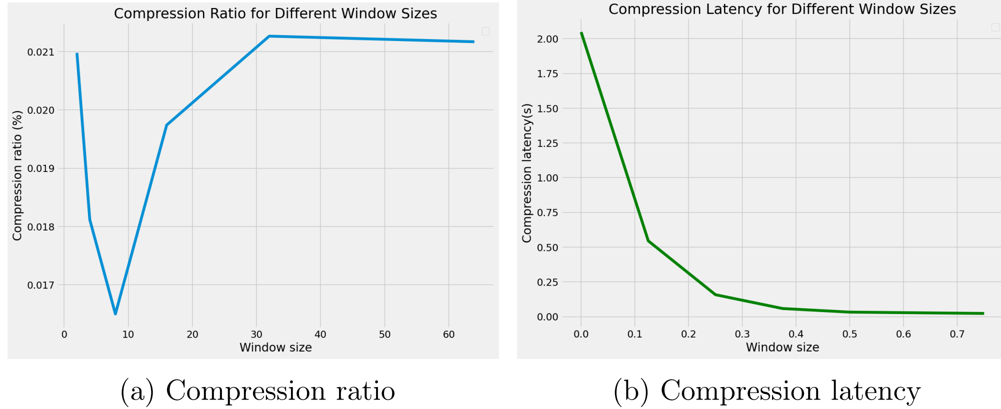
On Figure 3.3a, we can see that as we increase the spatial resolution of the data sets the compression ratio improves. That's an expected result, since the adjacent points will be more similar in values, therefore improving the results. When it comes to the effect of different budget allocations for the different granularity levels, we notice that the 11m and 5.5m data sets have their best compression ratio at the degenerate case ($w = 0\%$). The window method only improves the performance of the compression when the values are relatively smooth within their vicinity, which is the expected as we increase the number of points within an area. However, Figure 3.3b show that when the raster data is fine grained the gains are substantial, once again achieving a 24% improvement over the QTRC baseline. As the spatial resolution of datasets continues to improve due to advances in sensing technology, we would expect that HIRE would be able to exploit the increasingly fine-grained structure.

Window Size

A window represents the 2d section of the data being analyzed at each point in time as described. We tested different window sizes ranging from 2x2 to 64x64 to evaluate their effects on both the compression ratio and latency.

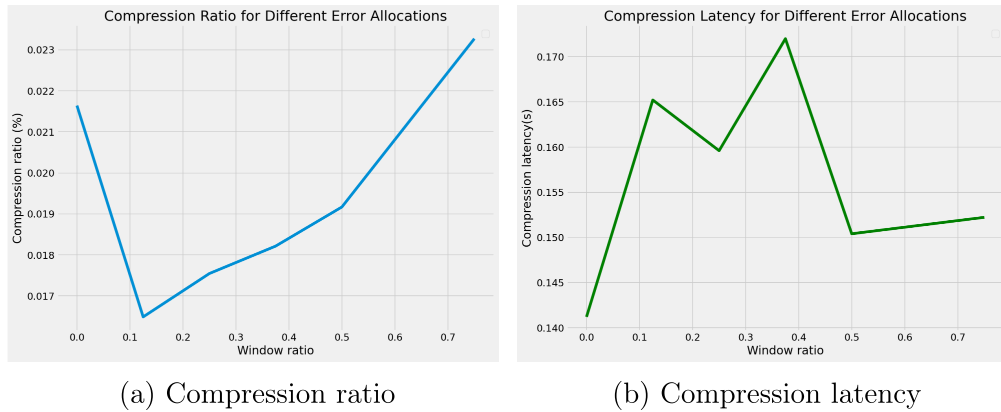
We notice in Figure 3.4a that the effect of the window size is not monotonic. It initially improves the compression ratio until it reaches its minimum at $w = 8$ and then it starts to get continuously worse. The optimal value seems to be data set dependent, where the more similar the neighboring data points are the greater the improvement of a larger window is. Once the window size surpasses the radius of similarity within the data the benefits subside.

On the other hand, the compression latency has the expected behavior with regards to the window size as shown on Figure 3.4b. The greater the window size, the faster the compression. One improvement that we aim to implement in the future is the use of parallelism with regards to the window's computations. If the windows are performing replacements in parallel we could have a balancing effect between the size of the window and the number of



(a) Compression ratio (b) Compression latency

Figure 3.4: Breakdown of performance for several different window sizes



(a) Compression ratio (b) Compression latency

Figure 3.5: Breakdown of performance for several different error allocations

possible parallel computations (up to a upper bound provided by the hardware).

Error Allocation

Error allocation is a vital parameter to consider in our method. As explained in **??** it determines how the total error budget ($L\infty$ error defined by the user) will be allocated to the window replacement method instead of the quantization. We evaluate the effects of allocating the following percentages of the overall budget to the window: [0,0.125,0.25,0.375,0.5,0.75]. In the following experiments we consider a total budget of 10% of $l\infty$ error, therefore once the window budget is chosen the quantization is its complement.

We can see on Figure 3.5a that the optimal choice of error to be allocated to the window

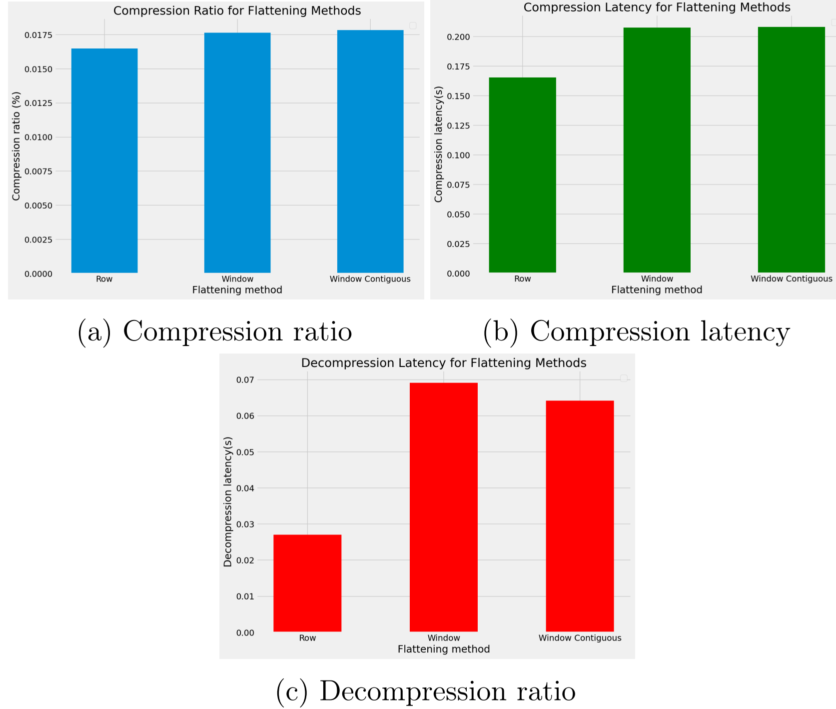
replacement is 12.5% of the total budget. The difference between the degenerate scenario with 0% o the error allocated to the window (QTRC baseline) and the best allocation corresponds to a 24% improvement in compression. Once again, the optimal choice will be data dependent, where the similarity between values will determine the best window ratio.

In Figure 3.5b we see that the compression latency is barely affected by the change in error allocation. This behavior is expected, since the computations involved are similar up to value replacements, where the more effective the allocation the more replacements are needed. However, those are barely noticeable in a modern computer architecture.

Flattening

Flattening corresponds to how the data is linearized in order for the downstream compressor to operate. Different flattening methods could affect how long the downstream compressor's run length is and therefore its performance. We tested 3 different versions: row, window and contiguous window. The row method flattens the data row by row sequentially, trying to maintain the run length along the horizontal axis. The window methods leverage the spatial relationship between adjacent data points by performing horizontal flattening within each window. The objective of the window methods is also to capture the redundancy introduced by the replacement algorithm, as well as the spatial characteristics of the dataset.

The difference between the window and window contiguous methods are the order of traversal through the input array. The window variant simply iterates through the windows from left to right and top to bottom, in a conventional pattern. On the other hand, the window contiguous method forces the order of traversal to only consider contiguous windows. If a window is on the edge of the array, the algorithm advances one window in the vertical direction and reverses the order of traversal in the horizontal direction. This ensures that every window is contiguous with respect to at least one other window when considering the ordering of the flattened array.

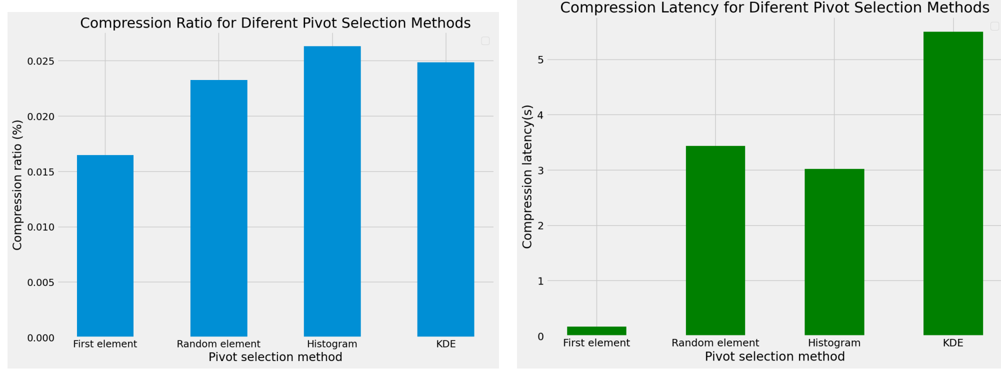


(a) Compression ratio (b) Compression latency

(c) Decompression ratio

Figure 3.6: Breakdown of performance for several flattening methods: row, window, contiguous window

In Figure 3.6 we see the effects of different flattening methods on the compression ratio, compression latency, and decompression latency. Although we might expect the window methods to perform better in terms of compression ratio, that is not the case on this dataset. In Figure 3.6a the row method, even though simpler, performs better at capturing the local similarities and extending the run length of the downstream encoder. While this result may seem surprising, at smaller window ratios, it is likely that the flattening method does little to magnify the effects of the replacement strategy. Hence, these conclusions may change at different combinations of window ratio and dataset. Furthermore, row flattening is faster both in terms of compression and decompression, as shown in Figures 3.6b and 3.6c.



(a) Compression ratio

(b) Compression latency

Figure 3.7: Breakdown of performance for several pivot methods

Pivot selection

There are several pivot selection methods that could be employed in our window method. We described 4 different ones in 3.3: first element, random element, histogram and kernel density estimate (KDE). Their corresponding effects on compression ratio are described in Figure 3.7a. We see that choosing the first element is the most effective choice even when compared to more complex estimation methods such as KDE. That is probably due to the other parameters, specially the window size. As we increase the window size, the effectiveness of more complex methods improves as both the statistical properties of the estimators (if we assume that the estimators are consistent, then as $n \to \infty$ the estimator converges to the real parameter) and the coverage of the window replacement increases inducing an improvement in the results.

When it comes to compression latency, the results are as expected. The more complex the method, the worst it performs in terms of latency. We notice on figure 3.7b that the choice the first element as pivot leads to significantly better results as the more complex algorithms perform much worse. In particular, the density estimation is the slowest due to its maximum likelihood procedure.

3.6 Conclusion

We presented RasterStore, an innovative framework to generate and compress rasterized data. We showed that traditional methods are not suitable to compress raster data due to their limited capacity to properly leverage spatial similarity within neighboring points or their lack of strong guarantees on error rates. We proposed a solution that extends Quantized TRC in order to replace similar points withing a window while preserving a user defined l_∞ error bound. Our experiments show that our method improves the compression ratio significantly while maintaining the decompression latency at excellent levels.

CHAPTER 4

GEOSPATIAL CASE STUDY: THE SOCIOME DATA COMMONS

4.1 Introduction

We present a case study that generated the interest in raster data compression. All the datasets used in the RasterStore paper had their origins in the following clinical research which also helps to motivate the necessity of a raster data compression framework.

Non-clinical aspects of life, such as social, environmental, behavioral, psychological, and economic factors, play significant roles in shaping patient health and health outcomes. These are broadly studied as Social Determinants of Health, which the World Health Organization defines as "conditions in which people are born, grow, live, work and age" and "fundamental drivers of [health] **?**." The design of such studies is fundamentally a data problem, where clinical patient data have to be integrated with other data sources to characterize a patient's life outside of their clinical interactions. We refer to the entirety of these non-clinical factors as a patient's "sociome." Due to the diversity of data sources and file types that sociome research has to consider, key bottlenecks in scaling such research to large patient populations include data integration**?**, data harmonization **?**, uneven data quality**?**, and statistical modeling of multimodal datasets**?**. Consequently, studies often focus on one factor, a composite index, or a set of highly related factors6, where potentially crucial nuances and interactions among factors can be lost.

Here, we report the design and implementation of the Sociome Data Commons (SDC). Leveraging the expertise of the Pediatric Cancer Data Commons **?**, we created a repository of pre-harmonized, geospatial sociome datasets that can be used in concert with clinical data to predict a variety of outcomes. To this end, we:

- Assembled and integrated publicly-available geocoded datasets about social, environ-

mental, behavioral, psychological, and economic exposures.

- Developed a data governance framework using a structured, standardized metadata model that conforms to FAIR (findable, accessible, interoperable, reusable) principles.
- Established a statistical methodology for analyzing sociome datasets of varying scope and quality, and for scaling and sustaining such analysis over large populations, environments and diverse data sources.

To evaluate the SDC, we performed a pilot use case to identify sociome factors associated with pediatric asthma exacerbations on the South Side of Chicago. Pediatric asthma was selected as it is a community priority, has well-documented social disparities, and is known to have many sociome influences, including housing and environmental conditions. In addition, clinical factors alone or models with limited variables have lacked sufficient predictive power for asthma outcomes.

4.2 Materials and Methods

In this manuscript, we use the following terms as defined below:

- Sociome Data Commons (SDC): A cloud-based repository of datasets characterizing a variety of local social, environmental, behavioral, psychological, and economic exposures.
- Metadata: Standardized descriptions of the content, quality, ownership, lineage, and scope of a dataset in the SDC.
- Model: A statistical or machine learning analysis that associates factors in the SDC to clinical outcomes derived from the electronic health record (EHR).
- Data Governance: The overall management and control of the assets in SDC, encompassing the policies, procedures, and frameworks that ensure data quality, accessibility to researchers, ethics, and privacy throughout its lifecycle.

- Generalizability: The ability for the SDC to store and serve multiple types of data and multiple types of models.
- Sustainability: The software infrastructure and organizational processes that govern the SDC will persist beyond initial pilot studies. The SDC is intended to be a persistent platform that can be meaningfully engaged by researchers across disciplines. It is built with the guiding principles described in Figure 4.1.

Sociome Data Commons Guiding Principles	
1. Governance	To ensure that researchers can ethically and accurately use the data stored in the SDC to construct a variety of exposure models.
2. Data Sustainability	To ensure that the accuracy and ease-of-use of the SDC does not degrade over time.
3. Dataset Inclusivity	To store, represent, and serve a variety of data types and structures that go beyond traditional health data.
4. Disaggregation	To focus on storing primary measurements of exposures and avoid derived indices.
5. FAIR principles	To ensure that all datasets and code comply with FAIR principles: Findable, Accessible, Interoperable, Reusable.
6. Multidisciplinary Management	To include clinicians, informatics researchers, computer scientists, social scientists, and community members in the research design.

Sociome Data Commons Datasets Add Dataset Data Dictionary Log in

Access The Sociome Datasets

The datasets below contain social, environment, and clinical factors relevant to patient outcomes.

Search for a dataset: Submit

Dataset Name	Description	Download
acs_income	Median household income from the American Community Survey.	Link
zillow_neighborhood	Data from Zillow.com describing house sale prices in a neighborhood.	Link
chicago_crime	Crime data from the City of Chicago data web portal.	Link
acs_poverty_index	A poverty index calculated through the ACS data	Link

Figure 4.1: SDC Guiding principles

4.3 Software Implementation

The SDC team has assembled and integrated diverse datasets into a simple, well-documented interoperable format. Researchers can use an application programming interface (API) to access these datasets directly from code or via an interactive website (Figure 4.1). The datasets are categorized with a structured, standardized metadata model that conforms to FAIR principles. These sociome datasets can be pulled into a protected enclave where they can be joined to clinical data (protected health information or a limited data set). Private (on premises) deployments can simplify privacy and security requirements, while a cloud-based multi-tenant solution could facilitate larger-scale collaborative research projects. Each dataset is documented with metadata describing its scope, quality, and units of measure. The

project utilizes a Python toolkit (that will be made available with an open-source license) to aid with common data integration and harmonization steps that researchers using the data might encounter. Researchers can identify the types of sociome factors they wish to investigate and easily build an integrated profile for a certain region.

4.4 Governance and Sustainability

The SDC establishes standards for dataset quality, dataset inclusion, metadata annotation, and data access. These standards will help promote trust in the included data and any derived conclusions. Novel contributions are presented in Figure 4.2.

Establishment of Data Documentation Standards	Each dataset in the SDC is annotated with a comprehensive data dictionary that adheres to FAIR principles. The structure of this dictionary is derived from the Data Documentation Initiative.[58]
Establishment of Data Quality Standards	The project establishes data quality norms to help researchers understand data veracity. Each dataset is assigned a data quality score on a scale of 1 (worst) to 5 (best). The score includes any errors in the dataset as well as missing, malformatted, or obvious outlier data using an error taxonomy developed in prior work.[59] Next, sampling biases in the dataset are evaluated to determine how representative the dataset is of the true underlying population (methodology described in detail in **Supplement 1b)**.
Mutli-disciplinary Research Review	The SDC is managed by a multidisciplinary team that includes clinicians, informatics researchers, computer scientists, social scientists, and community members. The SDC is designed to be both a data and code repository where research artifacts can be reviewed by a multidisciplinary team to ensure reliable, reproducible, and ethical research.
Differential Access	Where needed, authorization can be tailored to the user's role, allowing certain users access to different data, depending on privacy requirements.

Figure 4.2: SDC Standards

4.5 Asthma Pilot Methodology

We conducted a pilot use case of the SDC for demonstration and to test workflows. The pilot investigated sociome factors potentially related to pediatric asthma exacerbations. Clinical data was extracted from the University of Chicago Hospital's electronic health record (EHR) for all pediatric visits ($age < 18$). Data management and analysis on the extracted data occurred mainly in Python. 15 All clinical data (address history, demographics, diagnoses,

and encounters) were stored and analyzed on University of Chicago HIPAA-compliant compute and storage infrastructure. Some potentially-important clinical data, including allergy testing, asthma control test score, and overweight status, were not available at the time of this pilot. This study was approved by the University of Chicago BSD IRB, 21-1920, and a waiver of consent was granted for this retrospective study.

4.5.1 Geocoding

To adhere to privacy requirements for PHI, an on-site geocoder was preferred. To test and show robustness between available geocoder platforms (both cloud-based and on premises), a test was performed using 1,000 randomly-selected publicly-available Chicago addresses16 as well as systematic misspellings of Chicago's city hall address (a public landmark). We tested batch geocoding with Decentralized Geomarker Assessment for Multi-Site Studies (DeGAUSS, locally-hosted geocoding software) against industry standards: OpenStreetMap, GoogleV3 (both via GeoPy18), and the Census geocoder. DeGAUSS performed as well as GoogleV3 and the Census, and all outperformed OpenStreetMap.

4.5.2 Missing data

The level of missingness in the clinical data was low. Insurance was 5% missing, race/ethnicity 2%, and gender $< 1\%$. Missingness was resolved in two phases. First, by patient and sorted by date, values were filled forwards and backwards. Second, remaining missingness (2% for race/ethnicity, $< 1\%$ for insurance and gender) was resolved with multiple imputation.

4.5.3 Outcome definition

Visits for asthma and asthma exacerbations were categorized by the encounter text description including "asthma" and "exacerbation", respectively. Using text captured 2,010

additional asthma encounters (out of 3.3 million total visits, 2006-2021) than using ICD codes alone. Both terms were required in the visit text to qualify for a visit for an acute asthma exacerbation.

4.5.4 Spatial clustering

Spatial clustering is based on the assumption that "location matters." Events near each other are often related more than events far apart. Clustering combines geographic areas (here, census tracts) together to maximize similarity among the census tracts and maximize dissimilarity between the clusters. This clustering is performed to find meaningful spatial commonalities. Further, clustering reduces the size of large location datasets. For example, over 300 census tracts on the South Side of Chicago can be reduced to fewer than 10 spatial clusters. Different clusters can have different characteristics, summary descriptions and, especially of interest here, risk profiles. Clustering itself is an unsupervised learning problem in that the researcher does not know which areas will be grouped together (though the researcher does need to decide on the number of clusters). We used the skater algorithm21 in R22, which creates a connectivity graph of tracts' central points as well as edges from tracts' contiguous borders. The "cost" of each edge is established from the dissimilarity between neighboring tracts, and edges with greater costs are pruned. Statistical significance is assessed with Moran's I statistic, which measures spatial autocorrelation (observations at locations which are either contiguous or not). Spatial clustering was conducted with exacerbation visits as a proportion of all asthma visits and only for tracts with at least 10 visits. The exacerbation percentage was then mean centered and standardized. Since we did not know how many clusters would be meaningful, we output 5 different spatial clustering variables. These variables had different numbers of cluster assignments – between 3 and 7 – and were designed to be tested in our later model. That is, the model would determine whether 3 total clusters or up to 7 total were more important (described below).

4.5.5 Sociome Data Commons

For SDC datasets, a breadth of high-quality data types were chosen for this pilot study. All data were consistently aggregated at the census tract level to match the included American Community Survey (ACS) planning database and to avoid the modifiable areal unit problem of using different geographic levels in the same study. As available, data were reduced to years 2017-2019 to match the clinical data. Chicago crime was characterized via FBI guidelines as violent or not and rates for all crime, violent crime, and homicide (as a subset of violent crime) were created. Rates were also created for Chicago building code violations. ChiVes, the Chicago data collaborative and community mapping application, assembled a novel dataset including tree cover, biodiversity, summer PM (particulate matter) 2.5 estimates, traffic levels, and housing cost burden, among others. We developed a housing dataset, including building age and repair condition, sourced from the Cook County Tax Assessor's Office. The Environmental Protection Agency (EPA)'s Environmental Justice mapping and screening tool (EJ Screen) data was also included. Select variables in the ACS planning database are relative to the population (census tract averages, percentages, or median values). ACS average and percentages variables with less than a 10% range were also excluded, as variable data are needed to find meaningful differences. Census tract race/ethnicity distributions were also excluded to prevent overfitting on race. Reciprocal measures (e.g., percentage male and female) were reduced to one variable. To account for poverty and still identify other components of neighborhood conditions, we collapsed all ACS poverty-related variables together. Correlations were assessed relative to the percentage of persons living below the poverty level. The inclusion threshold was set at $|0.5|$. Both Pearson (linear relationships) and Spearman (monotonic) methods were used to maximize inclusion. Principal component analysis (PCA), which combines variables together with a linear orthogonal transformation, was conducted for feature reduction for the poverty-correlated variables (hereafter "poverty PCA").

4.5.6 Model

Data were modeled on the visit level and restricted to asthma visits. To avoid a UChicago-specific EHR artifact that occurred in late 2016 as well as COVID-19 pandemic complications, data were restricted to visits from 2017 to 2019. Because of sparse data outside of the South Side of Chicago, this pilot was limited to census tracts on the South Side Figures 4.3. Patient race/ethnicity and insurance variables were excluded: race/ethnicity to prevent overfitting on race as well as to permit later bias testing and insurance status because public health insurance is a proxy for poverty. All 5 spatial cluster variations were included to determine which provided the most information to the model.

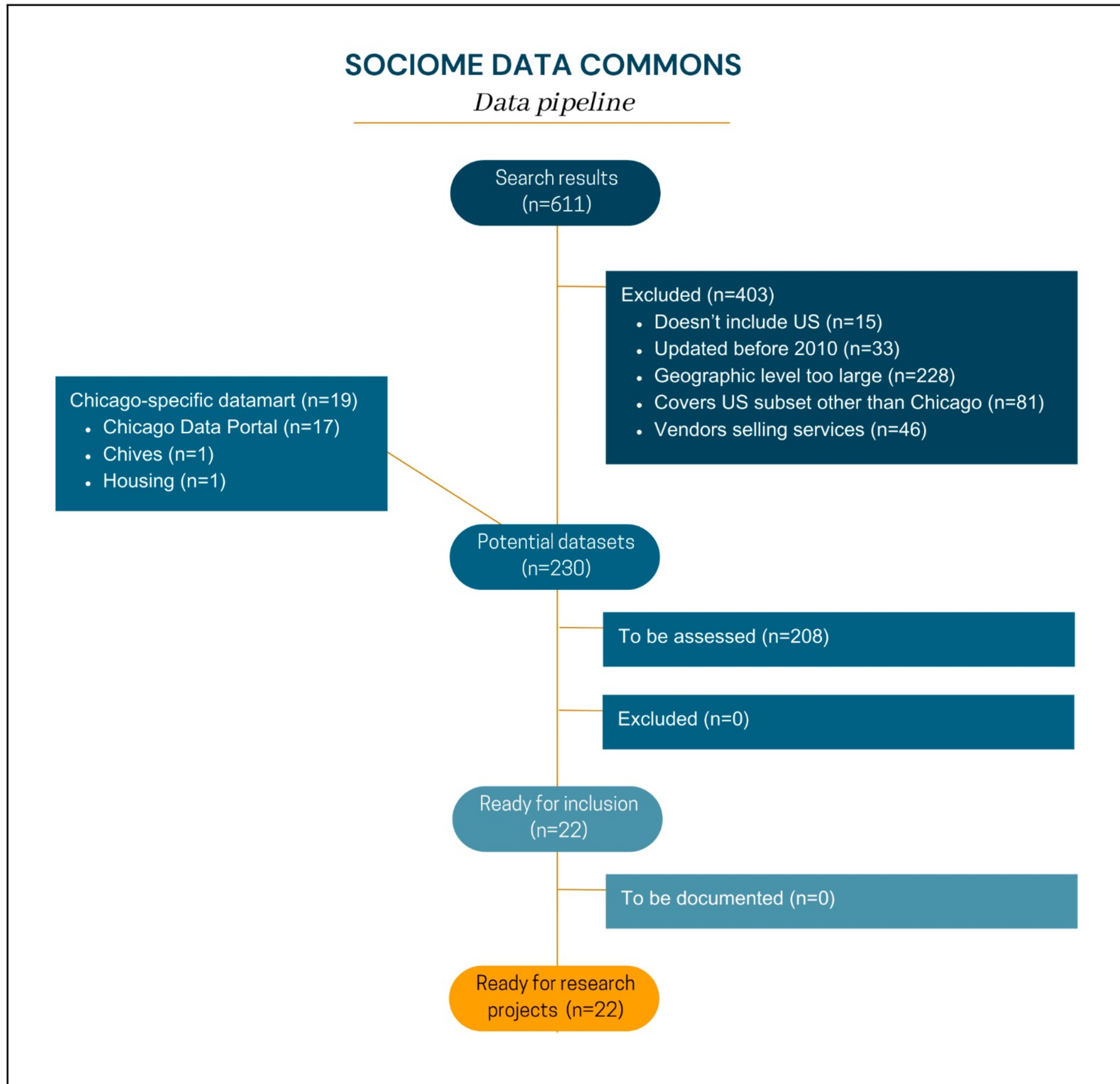


Figure 4.3: SDC Data pipeline

One nonlinear machine learning algorithm, a boosted decision tree, was piloted. Decision trees are non-parametric and do not make assumptions about the form of the data. This allows more flexibility and use of more complex datasets but also risks overfitting to the noise or randomness in the data. Boosted trees are an ensemble meta-algorithm, converting weak learning decision trees to strong ones in an iterative manner, such that each new tree improves upon the previous one. To assess overfitting, we split the data into train and test

sets. The model is designed ("fit") on train data only, and then solely "run" on the test data to predict outcomes. Model prediction accuracy, defined below, is compared between train and test datasets to determine if the model is overfitting on the initial train data. Weights were adjusted to account for outcome imbalance. Hyperparameters were optimized with Hyperopt, and manually adjusted after fit assessment with a 70/30 train/test split; with the final hyperparameters, shuffled 5-fold cross validation was conducted.

4.5.7 Evaluation

Metrics center on comparing model predictions to actual values. "Positive" indicates the outcome of interest, here, an asthma exacerbation, while "negative" indicates no exacerbation. "True" indicates that the actual outcome in the data matches the model-predicted outcome, while "false" indicates a mismatch between real and predicted outcomes. The metrics are: true positive (TP); true negative (TN); false positive (FP); and false negative (FN). Accuracy is the proportion of correct predictions ((TP + TN) / (TP + FP + TN + FN)). Test accuracy is the accuracy of the model on only the test dataset. Recall is the proportion of actual positives identified correctly (TP / (TP + FN)). Variable importances were determined by gain, which is the relative improvement in accuracy contributed by a particular feature. Gain provides a ranking of all variables in the model to indicate which are most important. It is vital to note that decision trees are not regression lines. If two variables are correlated, the decision tree will rank as more important whichever variable provides better accuracy. For example, we included 5 different spatial clustering variables to test which the model ranked highest, that is, which of the 5 provided the most gain. This served as variable selection for spatial clusters for future efforts. A baseline model included clinical and ACS data only. This was followed by a model with all SDC datasets and clusters to determine any improvement in predictive power. Further, the model would decide which of the 5 cluster variations provided the most gain to the model and, additionally, if clusters

were at all important relative to other variables.

4.5.8 Protocol testing

Two data-driven inclusion protocols were tested as future optional tools for users, and each used lasso-regularized logistic regression and added SDC datasets one-by-one to the clinical data (serially, not cumulatively). The first protocol assessed including full datasets via the AIC metric (Akaike information criterion; the lower the value, the better the model quality). The second protocol used lasso as variable selection. Each variable that reached statistical significance ($p<.05$) was included. Lasso regularization compresses coefficients to reduce bias and is a useful technique for variable selection.

4.5.9 Challenges

Challenges for this pilot include clinical data availability and the relatively low predictive power of variables ("signal") with the high variability of individual patients ("noise"). There is also a sampling bias in that the University of Chicago sees a distinct patient population – demographically, socio-economically, and geographically – which likely does not generalize to other populations.

4.6 Results

4.6.1 Sociome Data Commons

SDC Scope and Quality

The initial data repository consists of 22 total datasets and 375 individual metadata entries documenting their quality, provenance, and scope. Dataset content categories include environmental exposures (n=16), public safety (n=3), demographics (n=2), access and mobility

(n=2), property (n=1), and economic activity (n=1). The geographic levels include street addresses (n=14), census tract (n=7), census block (n=1), and latitude/longitude points (n=3). In the initial pilot use case, only high quality datasets were included. All 22 datasets to date have a data quality score of 4 or higher and 3 of the datasets have a score of 5. Furthermore, we found that datasets had varying geographic granularities. The fine-grained data, i.e., data at a finer precision than census tract, were in varying formats and scope. This requires significant efforts to harmonize and align with the other datasets using our software toolkit. In all, the datasets include 1906 total variables.

Usability Metrics

We evaluated the incremental cost of adding new datasets to the SDC by measuring the time required by a data engineering intern. These datasets had already been assessed by the team for relevance and quality. The metrics measure the time needed to use the automated harmonization software to reformat the data and enter the metadata into the commons. Over 13 random datasets, the average time to add to the Sociome Data Commons was 25 minutes with a wide standard deviation of 21 minutes.

CONSORT and Clinical Trial Ethos

We leveraged the deep expertise of the clinicians on the SDC team to develop a dataset inclusion pipeline that resembles a clinical trial flowchart. Datasets are assessed through a review process and progress through a number of stages until inclusion. Figure 4.3 shows the current status of this inclusion process.

4.6.2 Asthma pilot results

Clinical data

Using DeGauss, 93% of all clinical data were successfully geocoded from patient address to latitude and longitude, census block, and census tract. The other 7% of addresses were post office boxes (0.4%), non-address text (0.2%), and "imprecise" (6.9%), where "the address was geocoded, but results were suppressed because the precision was intersection, zip, or city and/or the score was less than 0.5."38 The table below is restricted to asthma visits among residents of Chicago between 2017 and 2019. The UChicago pediatric asthma patient population is 77% Medicaid and 89% non-Hispanic Black. Given the hospital's location and consequent patient catchment, restricting the data to the South Side of Chicago captured most asthma visits. However, population changes during the study period included reductions in the non-Hispanic Asian/Mideast Indian (78%) and White (81%) patients, while 97% of non-Hispanic Black and 92% of Hispanic patients remained, reflecting the segregation by race/ethnicity in Chicago. Spatial clustering of census tract exacerbations as percentage of asthma visits was significant (Moran's I = 0.5958, $p<.0001$) and also showed a dominant and geographically large cluster 1 where 39% of asthma visits were for exacerbations (Figure 4.4.C). In contrast, only 18% of visits were for exacerbations in cluster 5. The University of Chicago hospital itself resides within cluster 1, though the cluster extends far to the south and west (Figure 4.4.C). Restricting clinical data to patients within cluster 1, most asthma visits were retained. However, important changes include a dramatic reduction in the South Side Hispanic population 31%.

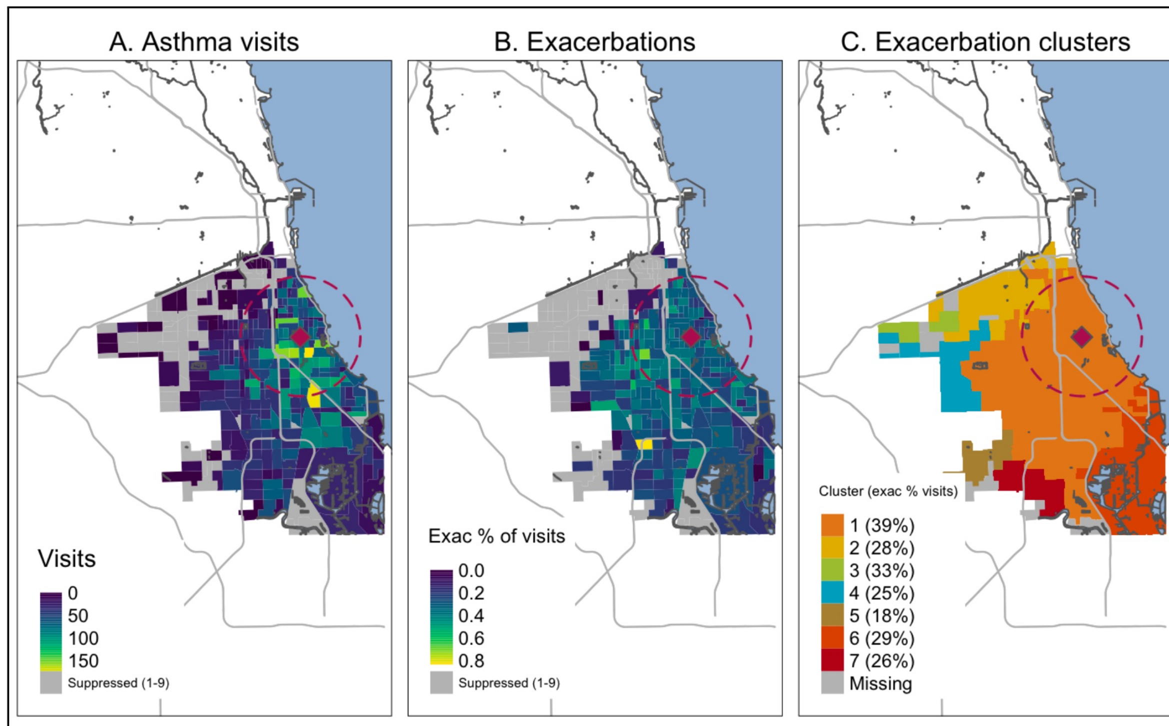


Figure 4.4: Clinical Maps

4.6.3 Sociome Data Commons datasets

For the poverty PCA, Spearman added four additional variables not included in the Pearson results, while Spearman alone would have excluded eight identified from Pearson. Twenty nine variables correlated with percent below the poverty level at |0.5| or above (see the supplement) and, along with percent below the poverty level, were mean centered and standardized. These variables were reduced to one PCA loading, which explained 56% of variance.

Model

A baseline boosted tree was run with only clinical and ACS data (which included the poverty PCA) with a test accuracy of 58%. With all datasets for the entire South Side, accuracy was increased to 62%, with recall for 1 (an asthma exacerbation) at 65%. Feature importance for the South Side revealed clear outliers for gain around 21 and again at 28. Most features

were at the lowest end of gain. Spatial clustering appeared twice in the top features. Seven clusters and the average age of housing provided the most gain in accuracy, followed by age at visit, proximity to Superfund pollution sites (marked for decontamination by the EPA), median rent, and the violent crime rate. The proportion of residential housing, urban flood susceptibility, and 3 spatial clusters followed in the top 10 variables. Given the geographic dominance of spatial cluster 1 and its high proportion of asthma exacerbations, we ran a model only on those patients residing within that cluster. Cross validated accuracy was 57%, accuracy was 61%, and recall for 1 was 60%. For cluster 1, there is a clear grouping of top feature importances at gain above 25. Otherwise, there is an accumulation of features around 8. The average age of housing units, the percentage of those under age 19 with no health insurance, the visit month, and the patient's age at visit were dominant variables in the model. These were followed by a second grouping of features in the top 10 of those 65 and over with no health insurance, housing cost burden, foreign born residents, median cost of rent, the poverty PCA, and several variables indicating a lack of health insurance.

Variable importances changed between the model including all of the South Side and the model restricted to cluster 1. Housing age (Figure 4.5.A) remained a top variable, as did patient age and median rent (Figure 4.5.B). However, moving from the full South Side to only cluster 1, urban flood susceptibility (Figure 4.5.C), the violent crime rate (Figure 4.5.D), and proximity to Superfund sites (Figure 4.5.E) left the top 10 important variables.Instead, several lack of health insurance variables and the poverty PCA (Figure 4.5.F) gained importance.

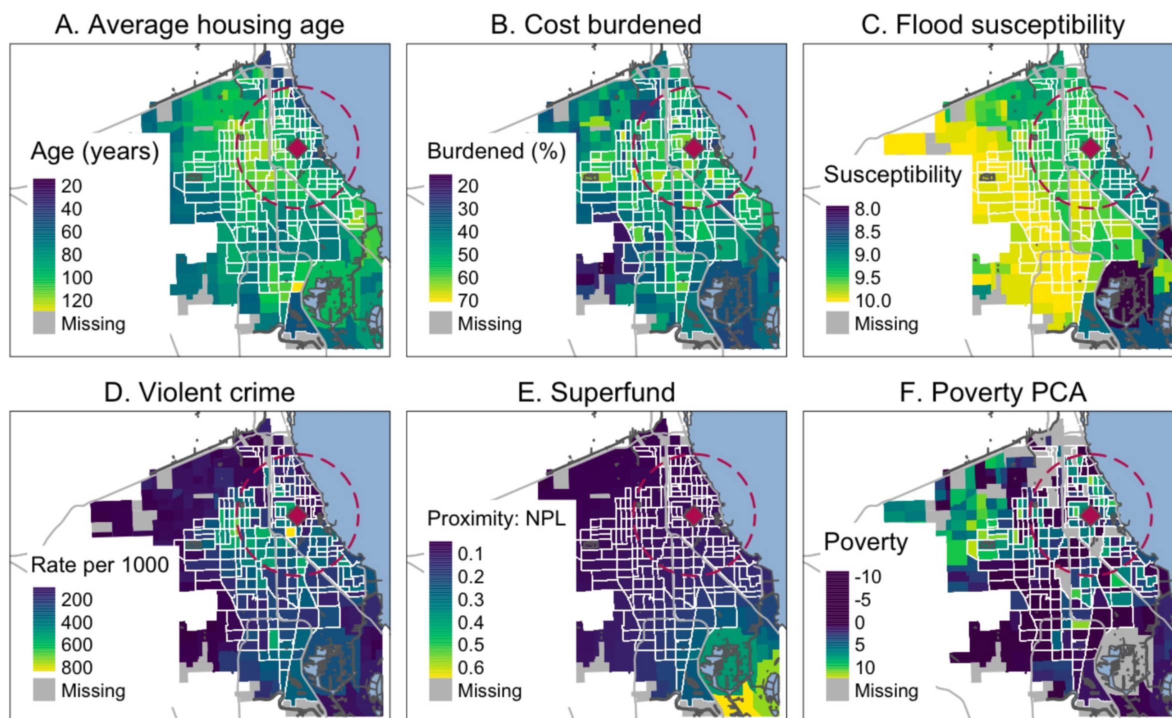


Figure 4.5: Sociome Maps

4.7 Conclusion

4.7.1 Platform discussion

he sociome data commons (SDC) has been purposefully designed to support and encourage extension of the platform into new data sets as well as the continued development, refinement, and adoption of SDC standards for dataset quality, dataset inclusion, metadata annotation, and data access / governance. The asthma pilot has served as the first driver use case for the SDC. Additional projects will be selected, in part for their ability to exercise and grow the capacity of the SDC to meet its ambitious goals. The purpose of this study is twofold: (1) to further the understanding of sociome factors in a variety of pathologies and (2) to understand principles of sustainable data commons design. The first purpose has parallels to other similarly-intentioned ongoing efforts. The National Neighborhood Data Archive (NaNDA) at the University of Michigan, the Health Equity Explorer (H2E) at Boston Medical Cen-

ter, Exposomics from the University of Utah, the City Health Dashboard from New York University, PopHR at the University of Tennessee, and SDOH and Place are among many groups aggregating important data expressly to enable researchers' use of sociome factors. We believe the second purpose - to understand the principles of sustainable data commons design - is unique to this project. The SDC is studying the design and implementation of such a data commons as a scientific problem including establishing quantitative metrics for assessing data quality, ease-of-use, researcher adoption, and sustainability. This manuscript contributes a framework for evaluating such criteria and we believe this to be an important contribution to this research area. This pilot and manuscript have been tailored to Chicago, but we are collecting national datasets, and can build additional location-specific datamarts as needed. However, local understanding provides critical context to properly using the data, and our group holds high knowledge of Chicago and available datasets. Notably, not all cities have tools such as the City of Chicago's Data Portal. In ongoing work, the team plans to make the entire SDC infrastructure publicly available and open source. It will include the software artifacts designed as a part of the project, such as the data harmonization and analysis code. It will further open-source the management infrastructure of how to host and serve these datasets. Furthermore, we will release policy templates for data governance and quality assurance. Once we finish integration of datasets, the SDC will serve as a reference implementation for data commons across a variety of social contexts of health research problems.

4.7.2 Analysis discussion

The pilot use case reported above was helpful in providing direction for our future SDC and analytic efforts. Spatial clustering of exacerbations, housing conditions, proximity to Superfund sites, violent crime, urban flood susceptibility, lack of health insurance, and the poverty PCA all contributed importantly to prediction of asthma exacerbation. Some of

these reflect current literature on risks for asthma, such as housing conditions and violence. Further, housing variables appeared in the top 20 variables, such as the building violation rate and mobile home percentage . Of course, these housing quality indicators are only proxy estimates for each individual patient's actual housing conditions and exposures, such as indoor air quality, first, second, or third-hand smoke exposure, and mold or pest exposure.47–50 Given a subset of patients' actual housing and indoor air quality, we could work to identify which, if any, SDC datasets could serve as a best proxy. The addition of personal exposure data might increase predictive accuracy, and the extent to which this occurs would inform how well (or not) generalized survey data like the sociome datasets perform in their stead. Proximity to Superfund sites merits further exploration of these and other pollution sites such as landfills and risk management plan sites. Exploring patient-level distance to these, rather than census tract estimates, is a next step. Notably, known risks such as PM2.5 exposure and traffic proximity48 did not appear in the top 10 variables, though they are in the top 20. Other findings, such as the violent crime rate and lack of insurance, also replicate the literature. Rarely-seen findings53 to be further explored and include urban flood susceptibility, which could indicate poorer housing quality and perhaps indicate susceptibility to damp housing and mold growth. Further exploration of flooding is needed. In this pilot, spatial clustering proved to be important. Model stratification was only conducted for the dominant cluster 1. A comparison of top variable importances demonstrates the promise of providing geography-specific risks, though further work is needed to clarify the reasons for the cluster differences, as well as a comparison model for the full South Side excluding clustering variables. Still, future work exploring cluster-specific models could inform geographically-tailored interventions.

4.7.3 Limitations

This study suffers from several data set limitations. For example, some probability surveys (such as NHANES54) contain important Sociome data, but are not publicly available at the census tract level. The models resulted in rather weak signals. While we anticipated that the importance might lie in an aggregation of multiple weak signals, predictive improvement is still needed. By choosing just a few datasets, we might not have yet included the most important datasets. We anticipate that broadening the range of sociome factors in an expanded SDC (which we are currently building) may increase model performance. The University of Chicago catchment area does not generalize to all pediatric asthma patients in Chicago. Variable data are needed to appreciate differences, and we need EHR data from other metropolitan Chicago health systems to provide greater patient heterogeneity. Efforts to expand the data are underway with our partners from the Institute for Translational Medicine. The spatial clustering of exacerbations might be affected by proximity to the UChicago hospital, as exacerbations are often urgent or emergency events. However, cluster 1 does extend far to the south and southwest of the hospital. Adding other hospitals' data should clarify clustering of exacerbations against hospital proximity. Analysis needs to be expanded. We piloted data as cross-sectional, and future efforts will use longitudinal methods and correct for this temporal bias, which will also enable causal exploration. Patient-level, rather than visit-level, analysis might also increase predictive performance, as many of the asthma exacerbation visits could have been from a particular subset of patients. By matching patients to specific locations in our housing dataset, we can move to sub-census tract metrics. Future efforts should include patient-specific distance to the nearest hospital and also investigate asthma exacerbations in more detail, for example, if a visit to the emergency department was needed. Modeling itself needs to be expanded. With machine learning, multiple models can be combined, leveraging the models' strengths and balancing their respective deficiencies. For this pilot study, we were missing data on many traditional

clinical phenotypes (disease severity, atopic status, comorbidities) and all biological (genetic, epigenetic) information about the pediatric asthma patients included in this study. As a next step, by obtaining clinical notes and using natural language processing, we aim to identify individual patients' levels of asthma control, an important clinical covariate. Asthma severity can also be assessed via medication prescriptions. If we are able to obtain biomarker data for select patients, that could allow identification of gene-environment interactions. The users were restricted to the project team with insights and guidance provided by the team. A robust governance framework will be crafted. In addition to the data governance discussed here, we will address appropriate use, responsible use, and community impact including a flexible regulatory module to meet varied and changing requirements. We will activate a multi-disciplinary work group composed of technical, research, and legal experts, ethicists, and community representatives.

In sum, the pilot study reported here demonstrates promise for future analyses of the complex interactions of the sociome and clinical health factors using the Sociome Data Commons. We expect that its further development, including accounting for dataset quality metrics, will facilitate the accounting for sociome factors in a wide range of clinical research topics, from analysis of response to cancer immunotherapy, to pandemic preparedness, to common complex diseases like diabetes mellitus, and more.

REFERENCES

What are the band designations for the Landsat satellites? | U.S. Geological Survey — usgs.gov. `https://www.usgs.gov/faqs/what-are-band-designations-landsat-satellites`. [Accessed 18-09-2023].

Cuneyt G Akcora, Yitao Li, Yulia R Gel, and Murat Kantarcioglu. 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*, pages 4439–4445, 2021.

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

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

Bruno Barbarioli, Gabriel Mersy, Stavros Sintos, and Sanjay Krishnan. Hierarchical residual encoding for multiresolution time series compression. *Proc. ACM Manag. Data*, 1(1), may 2023. doi:10.1145/3588953. URL `https://doi-org.proxy.uchicago.edu/10.1145/3588953`.

Carlo Batini, Cinzia Cappiello, Chiara Francalanci, and Andrea Maurino. Methodologies for data quality assessment and improvement. *ACM computing surveys (CSUR)*, 41(3):1–52, 2009.

Peter Baumann, Paula Furtado, Roland Ritsch, and Norbert Widmann. Geo/environmental and medical data management in the rasdaman system. In *VLDB*, volume 97, pages 25–29, 1997.

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

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

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

Clément Canonne and Ronitt Rubinfeld. Testing probability distributions underlying aggregated data. In *International Colloquium on Automata, Languages, and Programming*, pages 283–295. Springer, 2014.

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. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

Amit Chakrabarti, Graham Cormode, and Andrew McGregor. A near-optimal algorithm for computing the entropy of a stream. In *SODA*, volume 7, pages 328–335. Citeseer, 2007.

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

Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27(2):188–228, jun 2002. ISSN 0362-5915. doi:10.1145/568518.568520. URL `https://doi.org/10.1145/568518.568520`.

Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. Lfzip: Lossy compression of multivariate floating-point time series data via improved prediction. In *2020 Data Compression Conference (DCC)*, pages 342–351, 2020. doi:10.1109/DCC47342.2020.00042.

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

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

Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

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

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

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

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

Sudipto Guha, Andrew McGregor, and Suresh Venkatasubramanian. Streaming and sublinear approximation of entropy and information distances. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 733–742, 2006.

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

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

Ramón Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai Rulkov, and Irene Rodriguez-Lujan. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 157, 07 2016. doi:10.1016/j.chemolab.2016.07.004.

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

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

Fiodar Kazhamiaka, Matei A. Zaharia, and Peter D. Bailis. Challenges and opportunities for autonomous vehicle query systems. In *Conference on Innovative Data Systems Research*, 2021.

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

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

Eamonn J Keogh and Michael J Pazzani. Scaling up dynamic time warping for datamining applications. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 285–289, 2000.

Risi Kondor, Nedelina Teneva, and Vikas Garg. Multiresolution matrix factorization. In *International Conference on Machine Learning*, pages 1620–1628. PMLR, 2014.

Pavel Krasikov and Christine Legner. A method to screen, assess, and prepare open data for use. *ACM Journal of Data and Information Quality*, 2023.

David Krasowska, Julie Bessac, Robert Underwood, Jon C Calhoun, Sheng Di, and Franck Cappello. 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)*, pages 47–53. IEEE, 2021.

Sanjay Krishnan and Stavros Sintos. Range entropy queries and partitioning. In *27th International Conference on Database Theory (ICDT 2024)*, 2024.

Iosif Lazaridis and Sharad Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. *Acm sigmod record*, 30(2):401–412, 2001.

Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3): 261–296, sep 1987. ISSN 0360-0300. doi:10.1145/45072.45074. URL `https://doi-org.proxy.uchicago.edu/10.1145/45072.45074`.

Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1129–1141, 2021.

Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447. IEEE, 2018.

Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014. doi:10.1109/TVCG.2014.2346458.

Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. Decomposed bounded floats for fast compression and queries. *Proc. VLDB Endow.*, 14(11):2586–2598, jul 2021. ISSN 2150-8097. doi:10.14778/3476249.3476305. URL `https://doi.org/10.14778/3476249.3476305`.

Lacramioara Mazilu, Norman W Paton, Nikolaos Konstantinou, and Alvaro AA Fernandes. Fairness-aware data integration. *ACM Journal of Data and Information Quality*, 14(4): 1–26, 2022.

WHO Commission on Social Determinants of Health and World Health Organization. *Closing the gap in a generation: health equity through action on the social determinants of health: Commission on Social Determinants of Health final report*. World Health Organization, 2008.

John Paparrizos, Chunwei Liu, Aaron J Elmore, and Michael J Franklin. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1887–1905, 2020.

John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. Vergedb: A database for iot analytics on edge devices. In *CIDR*, 2021.

Emanuel Parzen. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics*, 33(3):1065 – 1076, 1962. doi:10.1214/aoms/1177704472. URL `https://doi.org/10.1214/aoms/1177704472`.

Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, aug 2015a. ISSN 2150-8097. doi:10.14778/2824032.2824078. URL `https://doi.org/10.14778/2824032.2824078`.

Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015b.

Navneet Potti and Jignesh M Patel. Daq: a new paradigm for approximate query processing. *Proceedings of the VLDB Endowment*, 8(9):898–909, 2015.

Powturbo. Turbo range coder, 2022. URL `https://github.com/powturbo/Turbo-Range-Coder`.

Murray Rosenblatt. Remarks on Some Nonparametric Estimates of a Density Function. *The Annals of Mathematical Statistics*, 27(3):832 – 837, 1956. doi:10.1214/aoms/1177728190. URL `https://doi.org/10.1214/aoms/1177728190`.

Julian Seward. bzip2 and libbzip2. *avaliable at http://www. bzip. org*, 1996.

John Shahid. Influxdb documentation, 2019.

B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. Monographs on Statistics and Applied Probability. Chapman & Hall, 1998. URL `https://books.google.com/books?id=fi0StQEACAAJ`.

Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Data compression for the exascale computing era-survey. *Supercomputing frontiers and innovations*, 1(2):76–88, 2014.

Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, page 127–140, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336314. doi:10.1145/2809695.2809718. URL `https://doi.org/10.1145/2809695.2809718`.

Andrea M Storås, Inga Strümke, Michael A Riegler, and Pål Halvorsen. Explainability methods for machine learning systems for multimodal medical datasets: research proposal. In *Proceedings of the 13th ACM Multimedia Systems Conference*, pages 347–351, 2022.

Samuel L Volchenboum, Suzanne M Cox, Allison Heath, Adam Resnick, Susan L Cohn, and Robert Grossman. Data commons to support pediatric cancer research. *American Society of Clinical Oncology Educational Book*, 37:746–752, 2017.

Gregory K Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992.

Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Accelerating generalized linear models with mlweaving: A one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment*, 12(7):807–821, 2019.

Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, jun 1987. ISSN 0001-0782. doi:10.1145/214762.214771. URL `https://doi.org/10.1145/214762.214771`.

Ramon Antonio Rodriges Zalipynis. Chronosdb: distributed, file based, geospatial array dbms. *Proceedings of the VLDB Endowment*, 11(10):1247–1261, 2018.

Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M Grulich, Ariane Ziehn, and Volker Mark. Nebulastream: Complex analytics beyond the cloud. *Open Journal of Internet Of Things (OJIOT)*, 6(1):66–81, 2020.

Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D Tonellot, Zizhong Chen, and Franck Cappello. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1643–1654. IEEE, 2021.