Analyzing and Improving Floating-Point Compression for Application Monitoring Time Series

Konstantin Urbanides
konstantin.urbanides@dynatrace.com
Dynatrace
Linz, Austria

Markus Weninger
markus.weninger@jku.at
Institute for System Software
Johannes Kepler University Linz
Linz, Austria

Otmar Ertl
otmar.ertl@dynatrace.com
Dynatrace Research
Linz, Austria

Abstract

High-volume telemetry from application monitoring systems presents significant data-storage and processing challenges. While lossless floating-point compressors (e.g., Gorilla, Chimp, and Elf) are effective, their performance is typically evaluated on academic benchmarks from domains such as finance or meteorology. We show that these benchmarks do not reflect application monitoring data, which is often highly repetitive and contains long sequences of non-varying values. This "dataset bias" leads to suboptimal performance when production data mismatches an algorithm's core assumptions. This paper tackles this research-to-practice gap with three contributions. First, we analyze a large-scale application monitoring dataset, quantifying the "dataset bias" and identifying high redundancy as its key characteristic. Second, we propose a multi-layered compression strategy combining record-level run-length encoding (RLE) with XOR-based compressors, yielding ≈23% compression ratio improvements and ≈34% speedups over standalone algorithms. Third, we develop a novel, lightweight heuristic that dynamically selects a compressor for each data block based on its statistical profile. This adaptive approach achieves near-optimal compression (within 2% of the optimal ratio) and further improves compression ratio by over 13% compared to the best single RLE-enhanced compressor, with negligible computational overhead. We conclude that data-driven, adaptive strategies are superior to universal algorithms for compressing domain-specific monitoring data.

CCS Concepts

• **Information systems → Data compression**.

Keywords

lossless compression, floating-point compression, lightweight compression, application monitoring time series

ACM Reference Format:
Konstantin Urbanides, Markus Weninger, and Otmar Ertl. 2026. Analyzing and Improving Floating-Point Compression for Application Monitoring Time Series. In *Proceedings of SIGMOD (SIGMOD-Companion '26).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD-Companion '26, Bengaluru, India
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern software systems generate exponential volumes of time series data, creating significant challenges for efficient storage and real-time analysis [26]. Time series are fundamental to application monitoring and system observability, capturing critical metrics from CPU usage to application-specific indicators [10]. These metrics are essential for diagnostics and performance optimization. Floating-point values are the standard representation for this data, as they can capture a wide range of values spanning many orders of magnitude. However, their standard 64-bit IEEE 754 representation introduces two major inefficiencies: it often provides far more precision and range than is semantically required, and its complex bitwise layout is notoriously difficult for compression algorithms to handle efficiently [1].

Observability platforms, such as Dynatrace, continuously ingest time series data from thousands of services. This data flows through a pipeline for processing and storage, where streaming-based compression is crucial. Data is compressed as it arrives, typically using only a limited context of recent values. In this environment, efficient compression is not a mere optimization; it is a fundamental necessity to build scalable and cost-effective observability platforms.

XOR-based compressors such as Gorilla [42], Chimp [31], and Elf [30] are well-suited for this streaming task, offering rapid processing of floating-point values. However, a critical gap exists between their benchmarked performance and their practical utility in production systems. This gap stems from a fundamental problem: the academic benchmarks used to evaluate these algorithms, often from domains such as finance or meteorology, feature constantly fluctuating values and do not reflect the data characteristics of application monitoring environments.

Our analysis reveals that application monitoring data has fundamentally different characteristics: it is often highly repetitive, containing long sequences of identical, non-varying values (e.g., a static configuration setting). This discrepancy stems from applying algorithms designed for other domains to application monitoring. Consequently, they may not perform optimally in Dynatrace's production systems, as the real-world data differs from their original design assumptions. Furthermore, production requirements extend beyond compression ratio; frequent data access for dashboards and queries demands high-speed decompression.

This paper tackles this research-to-practice gap by addressing the following question: How can we design a compression strategy tailored to the unique characteristics of application monitoring data, while balancing the competing demands of compression ratio, speed, and real-time applicability? We argue that the most effective solution is not a single algorithm, but a multi-layered approach.

Our work thus encompasses the following contributions:

(1) We analyze real-world monitoring data to uncover structural and behavioral differences compared to common floating-point benchmarks, quantifying the "dataset bias" (Section 3).
(2) We evaluate the performance of existing XOR compression algorithms such as Gorilla and Elf and propose various new variations thereof (Section 4).
(3) We propose a layered compression approach that combines lightweight techniques, such as record-level run-length encoding (RLE), with specialized compressors to efficiently exploit the data's high redundancy (Section 5).
(4) We develop a novel, data-driven heuristic method for dynamically selecting the most suitable compressor for each time series block based on its statistical profile, achieving near-optimal compression with minimal overhead (Section 5.4).

2 Background

Although general-purpose compressors such as LZ4 [11], Zstd [12], Snappy [17], and gzip [27] offer good compression ratios, they are often unsuitable for high-throughput streaming scenarios common in our domain of application monitoring. Thus, in this section, we present the common basis for floating-point compressors used for streaming data, bitwise XOR operations, followed by an overview of the most prominent lossless streaming-based compressors: Gorilla, Chimp, and Elf.

2.1 XOR-based Compression

Previous-value-based compression methods for floating-point data [29–31, 42] are built on the observation that consecutive values in a time series are often highly similar. These algorithms use the previous value as a predictor for the current one, eliminating the computational overhead of complex predictive models. The difference (or delta) between the current value v_t and the previous value v_{t-1} is captured efficiently using a bitwise XOR operation: $XOR_\Delta = v_t \oplus v_{t-1}$. As shown in Figure 1, the resulting delta for IEEE 754 double-precision floating-point values is conceptually partitioned into three components: a sequence of leading zeros, a block of center bits that represent the actual difference between the values, and a sequence of trailing zeros. By definition, the block of center bits begins and ends with a '1' bit. When two XOR'd values are identical, their bitwise XOR delta is 0. This all-zero delta naturally has 64 leading and trailing zeros, resulting in 0 center bits.

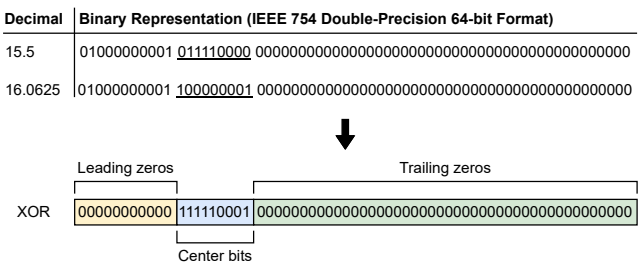

Figure 1: Conceptual partitioning of an XOR delta into three components: leading zeros, center bits, and trailing zeros.

2.2 Gorilla

Gorilla [42], introduced by Facebook in 2015, is the de facto standard for XOR-based compression and is used in prominent time series databases such as InfluxDB[1], Prometheus[2], and TimescaleDB[3]. Gorilla's design is based on the assumption that consecutive values are frequently identical or, if not, the structure of their XOR delta (i.e., the number of leading and trailing zeros) remains stable.

Gorilla's compression scheme for any value after the first (which is stored uncompressed) is a 3-case decision tree, as illustrated in Figure 2:

(1) **Identical Value ($XOR_\Delta = 0$).** If the current value is identical to the previous one, the XOR result is zero. This is assumed to be highly common and is thus encoded efficiently with a single control bit: '0'.
(2) **Different Values ($XOR_\Delta \neq 0$).**
   (a) *In-Window Compression.* The algorithm checks if the new delta fits within the *window* established by the previous delta. This condition is met if the new delta has at least as many leading zeros and at least as many trailing zeros as the previous delta. If it fits, the algorithm writes a '10' control sequence, followed only by the center bits of the new delta.
   (b) *New Window Creation.* If the delta does not fit into the previous *window*, a new *window* is created. The algorithm writes a '11' control sequence, followed by metadata for the new *window*: the leading zero count (6 bits), the center bit count (6 bits), and finally the center bits themselves. This new *window* is then used for subsequent *in-window* checks.

Gorilla's primary strength is its 1-bit encoding for identical values, making it exceptionally efficient for data with high redundancy and long, non-varying sequences. This baseline version of Gorilla, which uses 6 bits to encode the leading zero count, is referred to as "Gorilla (6 bits)" throughout this paper.

2.3 Evolutions of Gorilla

Gorilla's success inspired a family of related algorithms. These extensions serve as direct responses to Gorilla's core assumptions, introducing novel techniques to handle different data patterns and address perceived inefficiencies in specific scenarios.

2.3.1 Chimp and Chimp$_{128}$. Chimp [31] was designed based on a contrasting observation: that identical consecutive values are rare, while high bitwise similarity is common. This leads to a fundamentally different encoding scheme that de-prioritizes the identical-value case, requiring a 2-bit flag ('00'). Chimp's primary branching logic is based on the number of trailing zeros, not an initial check for identity. Its key innovation is a 3-bit quantized encoding for the leading zero count, which both halves the metadata cost compared to Gorilla's 6 bits and increases the probability of matching the previous leading zero count, enabling a more efficient encoding case.

[1] https://github.com/influxdata/influxdb
[2] https://github.com/prometheus/prometheus
[3] https://github.com/timescale/timescaledb

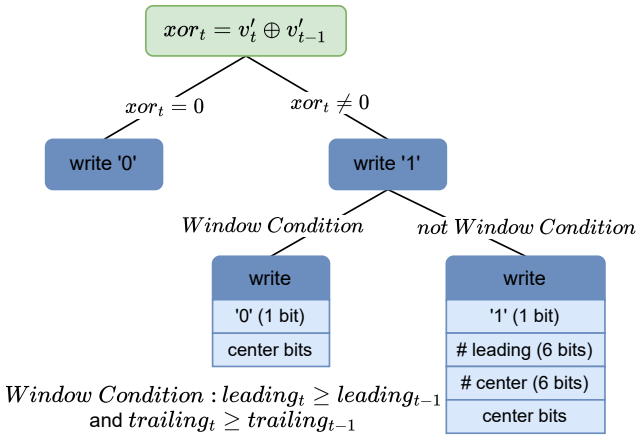

Figure 2: Gorilla's three-case decision tree, distinguishing between identical values ($XOR = 0$) and different values that are either in-window or require a new window.

Chimp$_{128}$ extends this by introducing a 128-value in-memory ring buffer. Instead of only comparing against the immediately preceding value, it searches this buffer for the historical value that produces the most compressible XOR delta. This trades a 7-bit pointer overhead—used to store the position of the best match—for the potential to find a much better (or identical) non-consecutive value, which can significantly improve compression on data with recurring patterns.

2.3.2 Elf and Elf+. Elf [30] introduces a different philosophy: preemptive preprocessing. Elf's core mechanism is 'erasing', where it intentionally discards (zeros out) the least significant, often noisy, bits of the mantissa before performing the XOR operation. The goal is to create values that, when XORed, are more likely to produce a delta with a large number of trailing zeros. This requires storing metadata to ensure the original value can be losslessly restored. Elf also employs its own XOR compression scheme tailored for these 'erased' deltas, which, like Chimp, de-prioritizes the identical-value case (using a 2-bit '01' flag) and adopts the 3-bit quantized leading zero count.

Elf+ [29] is an incremental enhancement to Elf. It does not alter the core XOR logic but adds optimizations to the erasing step. Specifically, it introduces new encoding cases that check if the decimal significand count remains stable between consecutive values. If this is the case, this metadata can be encoded more efficiently, further reducing storage overhead.

3 Dataset Analysis

The performance of a compression algorithm is not absolute; it is critically dependent on the characteristics of the data it is applied to. Before evaluating the compressors, we must first establish a deep understanding of our dataset's structure, properties, and statistical profile. This analysis is the foundation for our subsequent findings, as it reveals the specific data patterns that the algorithms must handle.

3.1 Motivation

A critical gap exists between the theoretical performance of compression algorithms and their practical utility. Algorithms such as Chimp and Elf were designed and benchmarked against academic datasets that often feature constantly fluctuating values (e.g., financial data[4] or weather patterns [37–39, 41]). On these benchmarks, they can outperform Gorilla.

Our work is motivated by the need to evaluate these compressors on a large-scale, real-world application monitoring dataset from an industrial environment. We hypothesize that such data possesses fundamentally different characteristics than common benchmarks. This "dataset bias" means an algorithm's success in literature does not guarantee its applicability in a production setting.

3.2 Structure and Records

The dataset analyzed in this study consists of ≈292GB of uncompressed time series data, comprising 7,993 distinct metrics collected over a 24-hour period from internal operational services at Dynatrace.

3.2.1 Structure. The dataset is organized into the following key structures:

- **Metric.** A metric represents a specific system measurement, such as CPU utilization or I/O throughput.
- **Time Series.** A time series is the sequence of measurements for one metric from a single source (e.g., a specific host or service) over a 24-hour period. Our dataset contains over 63 million time series instances.
- **Time Series Block.** For processing, each continuous time series is segmented into discrete 4-hour blocks. A block is the primary unit of operation for compression and can contain up to 240 data points, corresponding to a one-minute sampling frequency.
- **Count Aggregation Record.** A defining feature of our dataset is the use of aggregation. Since a metric can be sampled more than once per minute (our tools enable customers to record data from any source as metrics in arbitrary frequency), measurements are consolidated into a single data point per minute [19, 20]. This *count aggregation record* provides a statistical summary and contains four 64-bit values:
  - **Count:** The number of measurements aggregated.
  - **Minimum:** The smallest value in the interval.
  - **Maximum:** The largest value in the interval.
  - **Sum:** The total sum of all the measurements in the interval.

This structure presents a unique compression challenge, as each data point or record contains four values, not one. Figure 3 details the components of a single time series from the dataset.

The time series block is our fundamental unit for compression evaluation. Each block of complex count aggregation records can be conceptionally thought of as four conceptual subseries, one for each value: a 'count series' (all `count` values), a 'min series', a 'max series', and a 'sum series'. Each subseries can then be compressed individually.

[4] https://zenodo.org/record/3886895, https://github.com/influxdata/influxdb2-sample-data

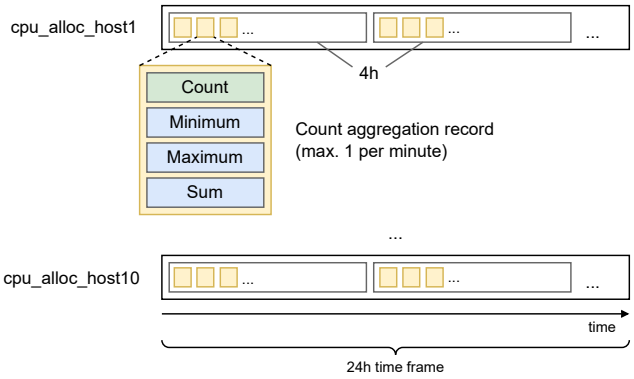

Figure 3: The structure of time series with its blocks and count aggregation records.

3.2.2 Record Encoding. To mitigate the storage overhead associated with these records, we employ an encoding strategy adapted from Moser et al. [19, 20]. This technique exploits redundancies within each record based on its `count` value before floating-point compression is applied. This is achieved by categorizing each record into one of three schemes, as illustrated in Figure 4.

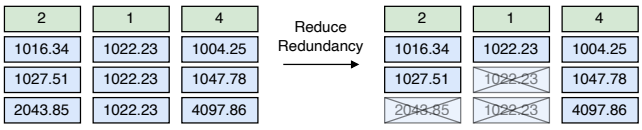

Figure 4: Our encoding strategy eliminates redundant values (e.g., removing the `sum` when `count == 2`).

- **Count == 1:** The `min`, `max`, and `sum` values are inherently identical. We only store the `min` value, discarding the other two.
- **Count == 2:** The `sum` is always equal to `min` + `max`. We store `min` and `max`, but discard the redundant `sum` since it can be reconstructed during decompression.
- **Count > 2:** No simple mathematical tricks exists, so all components (`min`, `max`, `sum`) are retained.

This encoding strategy significantly reduces the number of floating-point values that must be passed to a XOR compressor. During decompression, the omitted values can be reconstructed using the `count` value by applying the same scheme utilized for compression. The `count` values, which are integers by nature, are compressed using a variable-length integer encoding scheme. This scheme is specifically optimized for a count of 1, representing it with only a single bit. When evaluating the algorithms in Section 4, this record encoding is always applied as a preliminary step. In this work, we focus solely on the compression of these records. The compression of the timestamp associated with each record is outside of the scope of this paper and will not be discussed.

3.3 Core Findings of Dataset Analysis

Our analysis reveals that our application monitoring dataset deviates significantly from typical compression algorithm benchmarks in its structure, redundancy, and bitwise profile.

3.3.1 Block Characteristics and Data Redundancy. The defining characteristic of the dataset is the prevalence of non-varying sequences, where values remain constant over long periods. This high degree of data redundancy is the single most important factor influencing compression performance. An analysis of the `min`, `max`, and `sum` values within blocks reveals that, on average, only ≈13% of values in any given block are unique, meaning they appear only once. This indicates that the vast majority of values are repeated.

This redundancy manifests in two dominant block-level patterns:

- **Non-Varying Block:** Every count aggregation record within a 4-hour block is bitwise identical to its predecessor. The `count`, `min`, `max`, and `sum` values remain constant for the entire block (see Figure 5). A remarkable 64% of all time series blocks in our dataset fall into this category.
- **Varying Block:** Any block not classified as 'non-varying' is a 'varying block'. In this pattern, at least one record differs from the others, introducing bitwise changes that the compression algorithms must handle. These blocks, accounting for the remaining 36% of the dataset, represent the more complex compression scenario. While they often contain shorter runs of identical records, they are defined by the presence of at least one fluctuating value.

This pattern of non-varying blocks is fundamentally different from the fluctuating data found in financial or scientific benchmarks. It is likely caused by tracking (rather) non-dynamic metrics (such as the size of a session cache) or static configuration values as metrics. For example, we found metrics regularly tracking the version numbers of dependencies that (understandably) do not change throughout a single run (and would even stay constant across multiple runs as long as the dependency is not updated).

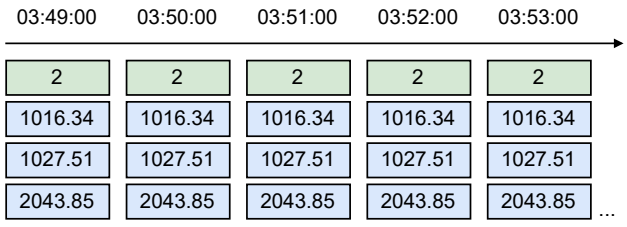

Figure 5: Example of a non-varying block, where all components of the count aggregation record, including the `count`, remain constant across the sequence.

3.3.2 Data Density and Distribution. The distribution of record counts per time series block reveals two key trends. First, a majority of blocks (over 40%) are densely populated, reaching the maximum possible count of 240 aggregation records. This corresponds to the highest sampling resolution of one record per minute for a four hour block. Second, a notable portion of blocks is sparse, containing significantly fewer records (on average ≈143) due to coarser sampling frequencies.

General-purpose compressors are often more efficient on larger data chunks. This does not directly apply to previous-value XOR compressors since they operate using only the delta between the current and previous value. Their performance is independent of block size and depends entirely on value-to-value similarity.

3.3.3 Value Characteristics and XOR Profile. The dataset's high redundancy directly impacts its bitwise profile, which is the critical factor for XOR-based compressors such as Gorilla [42]. These algorithms compress a value based on its bitwise XOR with the previous value. Our analysis reveals an exceptionally high frequency of identical consecutive values.

We find that over 82% of all consecutive value pairs in our dataset are identical. When two values are identical, their bitwise XOR result is 64 zero bits. As shown in Figure 6, this 82% rate of identical pairs is the dominant feature of our dataset's XOR profile, creating a massive spike at 64 trailing and leading zeros. Consequently, the leading and trailing zero distributions show strong correspondence. This behavior marks a fundamental distinction from typical time series benchmarks, where such uniformity is rarely observed.

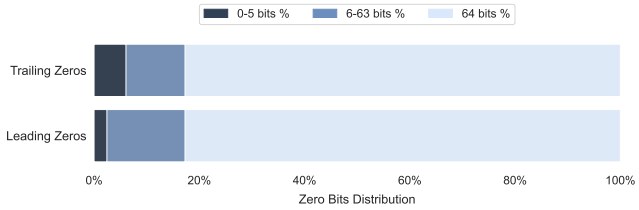

Figure 6: Distribution of the leading and trailing zero counts from the bitwise XOR of consecutive values. The bars show the percentage of XOR results that produce a specific number of zero bits (categorized as 0–5, 6–63, or exactly 64).

3.4 Quantitative Comparison to Benchmark Datasets

The development and evaluation of state-of-the-art algorithms such as Gorilla [42], Chimp [31], and Elf [30] have relied on a common set of academic benchmarks [4, 30, 31]. These benchmarks are typically composites of smaller datasets from diverse domains.

This composite benchmark includes data from:

- **Environmental Data:** Air pressure, temperature, wind speed[5], dew point measurements, dust and particulate size distribution from sources such as Meteoblue and NEON [37–39, 41].
- **Biological Data:** Infrared temperature of organisms and bird migration[6] tracking [40].
- **Economic Data:** Stock market prices[7] from Germany, the US, the UK, and Bitcoin prices[8].
- **Urban Data:** Temperature data from major cities[9].

While this diversity is suitable for testing general-purpose applicability, it does not reflect our specific domain. A quantitative comparison between our application monitoring dataset ("Dynatrace") and this composite benchmark ("Benchmark") reveals stark differences in key characteristics, as shown in Table 1. These differences have critical implications for compressor design.

[5]https://www.meteoblue.com/en/weather/archive/export/basel_switzerland
[6]https://github.com/influxdata/influxdb2-sample-data
[7]https://zenodo.org/record/3886895
[8]https://gz.blockchair.com/bitcoin/transactions/
[9]https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities

Data Redundancy. The most striking difference is data repetition. In our dataset, only 12.9% of floating-point values are unique per block (and only 8.3% of records are unique), indicating extreme redundancy. In contrast, the benchmark datasets are far less repetitive, with 45.1% unique values. We note that block definitions differ for this comparison: our analysis uses 4-hour blocks (up to 240 records), while the "Benchmark" datasets uses vectors of 1024 values.

XOR Profile. The bitwise profiles diverge starkly. Driven by the 82% identical-value rate (Section 3.3.3), our dataset exhibits exceptionally high average leading and trailing zero counts (55.1 leading zeros, 57.5 trailing zeros). The benchmark datasets, in contrast, average 23.8 leading and only 11.6 trailing zeros. This disparity highlights that the benchmark data is significantly less repetitive, contrasting sharply with the uniformity found in our dataset.

Volatility of Magnitude. When values do change, our dataset exhibits significantly higher volatility. The standard deviation of the IEEE 754 exponent in our data is 44.3, vastly higher than the stable 1.9 of the benchmarks. This high exponent volatility is almost entirely explained by a common pattern observed in our dataset: values frequently switching between 0 and 1. Due to the biased exponent in the IEEE 754 double-precision format, these two semantically close values have drastically different bitwise representations. The value 0 has an exponent of 0, while 1 has an exponent of 1023, and this massive jump explains the high standard deviation.

Structural Composition. Finally, the datasets differ fundamentally in structure. Our monitoring data consists of four component count aggregation records (`count`, `min`, `max`, `sum`), a richer format absent from the simple (timestamp, value) pairs found in the benchmarks.

Property	Dynatrace	Benchmark
Per Block		
Avg. Unique Values per Block %	12.9%	45.1%
Avg. Unique Record per Block %	8.3%	-
Std. Dev. of IEEE 754 Exponent per Block	44.3	1.9
Entire Dataset		
Avg. XOR Leading Zeros	55.1	23.8
Avg. XOR Trailing Zeros	57.5	11.6
Structure		
Count Aggregation Record	Yes	No

Table 1: Comparative analysis of key statistical properties between the application monitoring dataset ("Dynatrace") and conventional benchmark datasets ("Benchmark").

These profound differences in repetition, bitwise profile, volatility, and structure demonstrate that our application monitoring data represents a fundamentally different compression challenge. Consequently, the performance claims of algorithms benchmarked on standard datasets cannot be assumed to generalize to this domain. Our data's high redundancy (64% non-varying blocks) demands a strategy that efficiently handles long runs of identical values, while the high volatility of the remaining data requires a separate, robust encoding scheme. This "dataset bias" is precisely what justifies the need for the tailored, data-driven strategies we propose.

4 Evaluation of Compression Algorithms

Having characterized our dataset, we now evaluate the performance of the selected compression algorithms: Gorilla, Chimp/Chimp$_{128}$, Elf, and Elf+. The goal is to quantify how these state-of-the-art XOR compressors, along with variants of these implemented by us, handle the unique properties of application monitoring data identified in Section 3.

4.1 Compression Performance Metrics

We assess algorithm performance using two primary metrics:

- **Compression Ratio:** The unitless measure of size reduction, defined as the ratio of the uncompressed size to the compressed size. A higher ratio indicates better compression.

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size (Original Size)}}{\text{Compressed Size}}$$

- **Compression/Decompression Time (μs):** The average time in microseconds to process a fixed number of values. We report this per 1000 count aggregation records to provide a practical, record-oriented throughput measure. A lower time indicates faster performance.

Together, these metrics allow us to analyze the critical trade-off between storage efficiency and computational speed.

4.2 Gorilla Extensions

Given Gorilla's foundational role and its 1-bit flag for identical values—a pattern prevalent in our data (Section 3.3.3)—we focused on evaluating and extending it. We implemented the baseline Gorilla [42] and several new variants tailored to our dataset's properties.

Gorilla Optimized. This variant introduces two enhancements compared to the baseline Gorilla [42]:

(1) *Boundary Bit Elimination:* Leverages the fact that the first and last bits of a non-zero XOR delta's center bits are always '1'. These bits are inferred rather than stored, reducing the size of the center bit payload when creating new compression windows. The handling differs based on whether the number of center bits is 1, 2, or more than 2 as shown in Figure 7.

(2) *Adaptive Window Reset:* The baseline Gorilla can get stuck with an inefficiently large window. Our variant tracks the cumulative wasted space (window size minus required bits). If this regret exceeds a threshold (empirically set to 100 bits), the window is forcibly reset. This prevents sustained inefficiency on data with varying bitwise volatility.

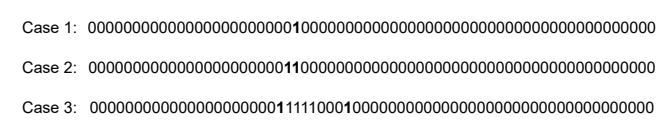

Figure 7: The three cases for boundary bit elimination in our Gorilla Optimized. The boundary '1' bits (bold) are inferred and not stored, only the center bits between them are stored.

Gorilla Reweighed. This variant challenges Gorilla's core assumption. It assigns the most efficient 1-bit flag ('0') to the *in-window* compression case, demoting identical values to a 2-bit flag. This makes it theoretically better for data with frequent, small, non-identical changes, that fall within the established compression *window*. As we will show it is poorly suited for our data when used as a standalone compressor.

Leading Zero Encoding Variants. We also tested modifications to Gorilla's 6-bit leading zero encoding for new compression *windows*. We implemented a variant using 5 bits with a cap ("Gorilla Limit"), where leading zero counts > 31 are treated as 31. The same applies to the "Gorilla Even" variant where the number of leading zeros are quantized to even numbers to be able to also only use 5 bits for storing the leading zero count. We also implemented "Gorilla Leading" a 3-bit quantized variant mirroring Chimp's approach, that halves the number of bits required to store the leading zero count. The sole distinction of these implementations from the baseline Gorilla is their more efficient encoding of the leading zero count. As our results will show, these variants offered negligible improvement.

4.3 Performance Evaluation

We evaluated all algorithms on storage efficiency and computational speed. Table 2 shows the compression ratios, segmented by the overall dataset, the 64% of blocks that are 'non-varying', and the 36% that are 'varying'.

4.3.1 Storage Efficieny. The results in Table 2 clearly demonstrate the "dataset bias" in action. The Gorilla family of algorithms decisively outperforms Chimp and Elf on our dataset. This is almost entirely due to their efficient 1-bit encoding for identical values, which allows them to achieve a remarkable 62.95 compression ratio on the 'non-varying' blocks.

In contrast, Chimp and Elf, which were designed for fluctuating data and de-prioritize the identical-value case, fail on this data. Their ratios of 47.25 and 40.28, respectively, on non-varying blocks show they are fundamentally misaligned with our data's primary characteristic. Even on the 'varying' blocks, the Gorilla variants remain superior.

The Gorilla Reweighed variant, which demotes the identical-value case, performs as poorly as Chimp on non-varying blocks (compression ratio of 47.25), confirming this design choice is wrong for our data. The leading-zero-encoding variants (Limit, Even, Leading) show negligible differences from the baseline, indicating that the 6-bit metadata cost is not a significant bottleneck.

Our Gorilla Optimized variant achieves the highest compression ratio (15.15) among all compressors. It matches the baseline's excellent performance on non-varying blocks while also achieving the best performance on the more challenging 'varying' blocks (6.67). This confirms our hypothesis that the baseline Gorilla can be improved with small, targeted optimizations for varying data.

The general-purpose compressor Zstd (Level 3) achieves the highest overall ratio (18.14). However, as shown next, this storage efficiency comes at an unacceptable computational cost for our use case. Moreover, comparing Zstd to previous-value based compressors such as Gorilla is not entirely fair. Zstd operates on large chunks of data, leveraging broader context for compression. In contrast, XOR compressors rely solely on the immediately preceding value (i.e., stream processing), which limits their compression capabilities.

Algorithm	Overall Dataset		Varying Blocks		Non-Varying Blocks		Compression Time (μs) per 1000 Records	Decompression Time (μs) per 1000 Records
	Compression Ratio	% of Best Compressions	Compression Ratio	% of Best Compressions	Compression Ratio	% of Best Compressions		
Gorilla (6 bits)	14.94	47.3	6.53	5.5	**62.95**	**70.5**	32.87	35.65
Gorilla Limit	14.96	48.6	6.53	9.1	**62.95**	**70.5**	32.27	33.23
Gorilla Even	14.79	47.7	6.45	6.6	**62.95**	**70.5**	32.04	32.61
Gorilla Leading	14.72	49.1	6.41	10.4	**62.95**	**70.5**	35.15	34.17
Gorilla Reweighed	14.27	7.8	6.55	21.4	47.25	0.3	**32.03**	**31.83**
Gorilla Optimized	**15.15**	**59.7**	**6.67**	**40.3**	**62.95**	**70.5**	34.16	32.62
Chimp	13.59	1.5	6.17	3.8	47.25	0.3	38.25	39.81
$Chimp_{128}$	9.54	3.8	5.44	10.2	17.21	0.3	155.67	150.96
Elf	13.55	25.6	6.38	18.5	40.28	29.6	58.32	37.98
Elf+	12.26	4.4	5.92	1.9	33.41	5.8	52.35	39.60
Zstd (Level 3)	18.14	–	7.07	–	474.52	–	97.14	72.45

Table 2: Overall compression ratio of all evaluated algorithms across the entire dataset. The results show the clear superiority of the Gorilla family, with Gorilla Optimized being the top-performing specialized compressor.

4.3.2 Computational Speed. Table 2 also presents the computational performance. For a real-time monitoring pipeline, low-latency compression and decompression are critical.

In terms of speed, the Gorilla variants are the clear winners. The baseline Gorilla (6 bits) (32.87 μs) and our Gorilla Optimized (34.16 μs) are exceptionally fast, and their decompression speeds are similarly low. This confirms their suitability for real-time query workloads.

In contrast, the other algorithms impose a significant performance penalty. Elf's compression time (58.32 μs) is nearly 80% slower than Gorilla's. Zstd (97.14 μs) is almost 3x slower, and $Chimp_{128}$ (155.67μs) is over 4.7x slower due to its expensive ring buffer search. These high-latency algorithms are not ideal for our high-throughput ingestion pipeline.

This analysis shows that for our use case, the Gorilla family provides the best balance of high compression ratio and elite computational speed. Gorilla Optimized, in particular, represents the best trade-off among specialized compressors, improving the compression ratio without compromising the low-latency performance required for real-time telemetry.

4.3.3 Best Compressions. While these evaluations are useful, an analysis based on block characteristics reveals where each algorithm excels. Figure 8 shows the percentage of blocks per block type for which each algorithm achieves the single best compression ratio. When multiple algorithms perform equally well, i.e., they exhibit the same compression ratio, they appear in multiple bars. Thus, each series (i.e., all blocks, varying blocks, non-varying blocks) sums up to more than 100%.

This fine-grained analysis confirms our aggregate findings. For the vast majority of non-varying blocks (Figure 8, medium blue bar), the Gorilla variants (Optimized, 6 bits, etc.) are the best performers, as they all compress identical values to 1 bit. However, a notable exception exists: for non-varying blocks that contain only a single record, Elf and Elf+ outperform Gorilla. This is an edge-case artifact: Gorilla stores the first value uncompressed (yielding no compression), while Elf's initial value encoding is more efficient.

For the more dynamic 'varying' blocks (Figure 8, light blue bar), the performance landscape is more fractured, but Gorilla Optimized is the clear winner. Our variant achieves the best compression for over 40% of all varying blocks, far exceeding any other algorithm. The second-best, Elf, is optimal for only ≈18% of these blocks. This confirms that Gorilla Optimized's high performance is not just an artifact of the non-varying data; it is genuinely superior at handling the more complex, fluctuating data in our domain.

In conclusion, our evaluation demonstrates that for our application monitoring dataset, Gorilla Optimized is the most effective standalone algorithm, offering the best overall balance of storage efficiency and computational performance.

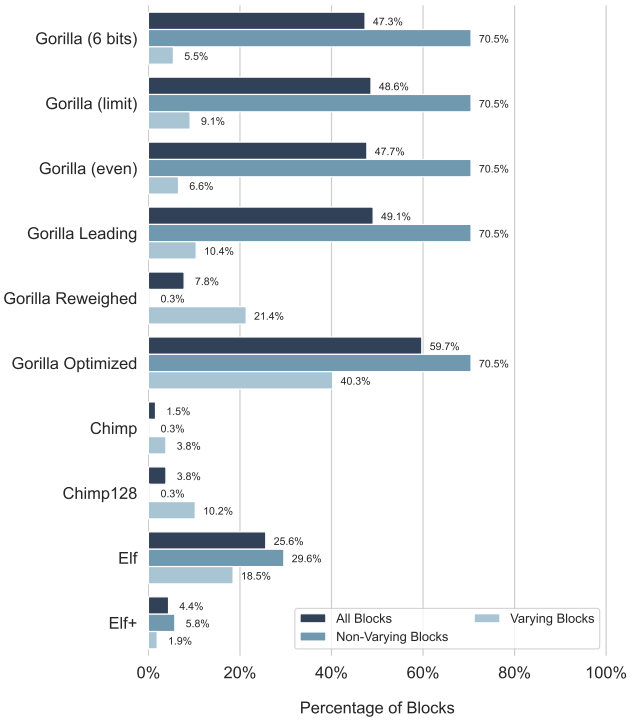

Figure 8: Comparison of how often which compressor performs best (number of time series blocks with highest compression ratio relative to the total number of time series blocks).

5 Optimizations

Our analysis in Section 4 demonstrated that Gorilla Optimized is the best standalone compressor for our data. However, the analysis also

showed that no single algorithm is optimal for all data patterns. This section introduces a series of data-aware optimizations that build upon our findings to further improve compression performance. We present a three-layered compression strategy based upon record-level run-length encoding and sophisticated redundancy reduction and evaluate the performance impact of these optimizations. We finally present a dynamic, heuristic-based framework for floating-point compression algorithm selection.

5.1 Record-level Run-Length Encoding (RLE)

Our dataset analysis in Section 3.3.1 revealed that 64% of all time series blocks are 'non-varying,' meaning every record in the block is identical. While Gorilla's 1-bit flag is efficient, it still expends bits for each identical record. This dominant characteristic suggests that a lightweight preprocessing step to handle these long, identical runs before the XOR compressor could yield significant gains.

Several of those lightweight techniques exist, such as frame-of-reference (FOR) encoding [16, 47, 52], delta encoding [3, 28, 45, 47], dictionary encoding [2, 6, 45, 47, 52], run-length encoding (RLE) [2, 45] and null suppression [2, 45]. We found that run-length encoding (RLE) is the ideal technique for our use case, as it replaces long sequences ('runs') of identical values with a single value and a count. For example, the time series `[5.31, 5.31, 5.31]` would be replaced by a pair in the form of `(count, value)`, i.e., `(3, 5.31)`. Given our data structure, we implemented RLE on the record-level. Instead of RLE-encoding individual values, our approach RLE-encodes the entire count aggregation record.

As shown in Figure 9, this method can reduce an entire non-varying block to just two components: the run count and a single copy of the record itself. This is vastly more efficient than Gorilla's encoding on floating-point value level.

This record-level RLE is also effective for 'varying' blocks, which often contain shorter, intermittent runs of repeated records. The record count is compressed using the same variable-length integer encoding as the `count` value, a scheme specifically optimized to represent counts of 1. Thus, in the worst case (a block with no consecutive identical records), this RLE layer adds only a negligible overhead of 1-bit per record. This provides large gains on redundant data with minimal risk on highly variable data.

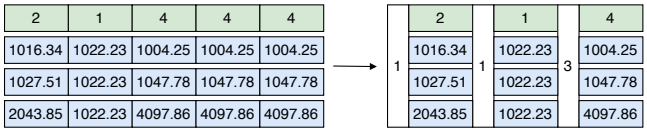

Figure 9: Run-length encoding (RLE) directly applied to count aggregation records

5.2 Extended Redundancy Reduction within a Record (ERRR)

This section introduces an optimization that builds upon the record encoding from Section 3.2.2. Our dataset analysis revealed that even within 'varying' records, a common pattern exists: in many cases, identical `min` and `max` values are present in our records, i.e., even if the measured values fluctuate within a 4-hour block, they often stay consistent within our 1-minute records.

Our extended redundancy reduction within records (ERRR), depicted in Figure 10, makes use of this observation in the following two ways:

- **Count == 2 and min == max:** For any record of this kind it means that the two aggregated measurements for that minute were identical. The base encoding (Section 3.2.2), dropping only the `sum` for records with a `count` of two, does not fully exploit this. In the extended version, we store only the `min` value and a control bit. This allows us to discard the redundant `max` value.
- **Count > 2 and min == max:** While the base encoding does not consider any optimizations for records with a `count` greater than two, we specifically taking into consideration cases where `min` equals `max`. This allows us to drop `max` and `sum` since `max` is equal to `min` and `sum` can be reconstructed as `count` × `min`. We verify this reconstructed `sum` is bitwise identical to the original `sum` before discarding the value, saving a second floating-point value.

This optimization significantly reduces the number of floating-point values that need to be passed to the XOR compressor, targeting redundancy within a single record.

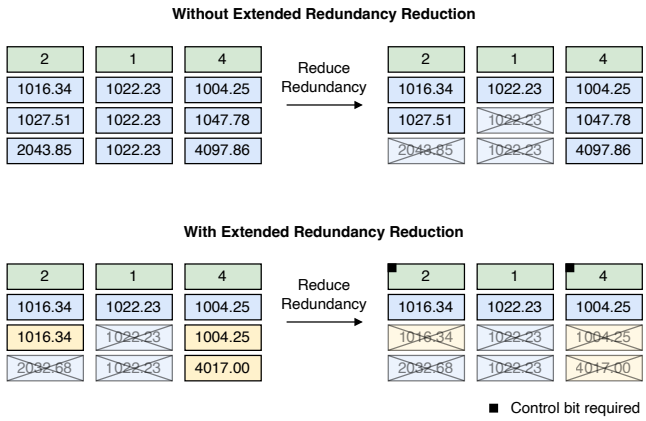

Figure 10: The extended record encoding strategy, which further reduces data by exploiting the common case where `min` and `max` values are identical.

5.3 Three-Layer Compression Strategy

We combined our optimizations from Section 5.1 and Section 5.2 into a modular, three-layer compression strategy:

- **Layer 1: Record-Level RLE.** Raw time series blocks are first processed by our record-level RLE (Section 5.1). This layer condenses long runs of identical records, drastically reducing the total number of records passed to subsequent layers.
- **Layer 2: Extended Record Encoding.** The unique records output by the RLE layer are then processed by the extended record encoding scheme (Section 5.2). This layer eliminates record redundancies within a record by extending the record encoding from Moser et al. [19, 20] by the `min == max` cases.

- **Layer 3: Integer and XOR Compression.** Finally, the integer values (i.e., RLE counts, and `count` values) are compressed using the variable-length integer encoding optimized for counts of 1. The control bits from the extended record encoding are handled separately, written bitwise in a defined order to the compressed stream. The remaining floating-point values (`min`, `max`, `sum`) are passed to a XOR compressor (e.g., Gorilla, Chimp, or Elf). Thanks to the preceding layers, the number of floating-point values passed to this layer is significantly reduced. Since this layer is the most computationally intensive—particularly when using advanced compressors such as Elf and Elf+—this reduction also helps minimize computational overhead.

This modular design (RLE → ERRR → XOR) is highly effective, as we will outline in the following sections. Layers 1 and 2 handle structural and record-level redundancies, significantly reducing the volume of data that must be processed by the computationally expensive Layer 3 XOR compressor.

5.3.1 Impact on Compression Ratio. Table 3 shows the powerful effect of applying our three-layer strategy (which we denote as *Layered <Algorithm>*). On average, it improves the compression ratio by ≈23% over the standalone algorithms (cf. Table 2). A particularly noteworthy result is the emergence of *Layered Elf* as top-performing combination (compression ratio of 18.60), surpassing even Zstd (18.14). This illustrates perfect synergistic effects:

(1) The RLE layer handles the 64% of non-varying blocks, where Elf is weak.
(2) This leaves Elf to compress only the more complex, 'varying' blocks, a task for which its erasing-based mechanism is highly effective.

This synergy—using a simple, fast layer for the common case and a complex compressor for the difficult case—is a key finding.

5.3.2 Impact on Compression / Decompression Time. Table 3 shows that the layered approach also delivers a substantial ≈34% average improvement in both compression and decompression speed.

This speedup is a direct consequence of the RLE layer. By drastically reducing the number of records that must be processed by the computationally expensive XOR compressors (Layer 3), the total workload is significantly reduced. This confirms that our layered strategy is a clear success, improving both storage efficiency and computational throughput.

5.4 Algorithm Selection Heuristic

Our results for the three-layer compression strategy (cf. Section 5.3) show that a single, "one-size-fits-all" layered approach can be highly effective and can even outperform Zstd. However, this is still a compromise. The 36% of 'varying' blocks are themselves highly diverse: some may be best suited for Layered Elf (e.g., noisy data), while others with stable, small changes may be better suited for Layered Gorilla Reweighed. This observation motivates a dynamic, per-block (i.e., 4-hour segment) selection strategy. We considered three paths:

(1) **Brute-Force:** Compress each block with every algorithm (e.g., all 10 algorithms and their *Layered* pendants) and choose that result with the highest compression ratio. This provides a theoretical optimum regarding compression ratio but is computationally infeasible, imposing an unacceptable runtime overhead on the ingestion pipeline by a factor corresponding to the number of considered algorithms.
(2) **Machine Learning:** Train a classifier (e.g., a decision tree [43, 44] or small neural network [35, 46]) to predict the best compressor from a feature vector (e.g., mean, std-dev, entropy, etc.) [33, 49]. This is a powerful alternative but introduces significant infrastructure and maintenance complexity (feature extraction pipelines, model serving, and continuous retraining to avoid model drift).
(3) **Lightweight Heuristic (Our Approach):** We propose a third path that achieves the adaptive benefits of an ML approach without the infrastructure cost. Our goal is to create a simple, data-driven "fingerprint" for each block and use it to select the best compressor from a pre-defined set. This method is fast and self-contained (no external models).

5.4.1 Our Lightweight Heuristic Framework. Applying our method involves four stages (illustrated in Figure 11), with stage 1–3 being performed offline and step 4 being performed online.

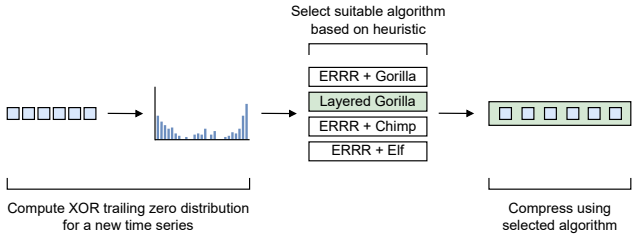

Figure 11: A high-level overview of the heuristic compression procedure, which involves computing a feature (XOR trailing zero histogram) and using it to select the optimal compressor from a predefined set.

1. Determine Optimal Compressor Set. First, we ran an offline brute-force analysis on our entire dataset to determine a relevant "target set" of compressors that perform best for certain blocks. We found that a small set of just seven algorithms (including our best-layered approaches such as Layered Elf, Layered Gorilla Reweighed, and specialized standalone compressors) accounted for 98.7% of all "best" compression results. This confirms we can achieve near-total coverage by reducing the search space from all 20 variants to just these 7.

2. Design a Predictive Feature ("Fingerprint"). The core idea of our heuristic is to use a feature that can "fingerprint" a block. Simple statistics (such as the average or standard deviation) are insufficient, as they average out the very distributional information our compressors rely on.

Via intensive testing and experimental exploration of our dataset, we finally settled for the 'trailing zero histogram' as our fingerprinting feature. This is a 65-bin array that counts the occurrences of each possible trailing zero count (from 0 to 64) found in the XOR deltas of a block's values. This histogram is the ideal fingerprint

Algorithm	Compression Ratio			Compression Time (μs) per 1000 Records	Decompression Time (μs) per 1000 Records
	Overall Dataset	Varying Blocks	Non-Varying Blocks		
Layered Gorilla (6 bits)	18.04 *(14.95)*	6.88 *(6.53)*	454.60 *(62.95)*	21.99 *(32.87)*	21.51 *(35.65)*
Layered Gorilla Limit	18.06 *(14.96)*	6.88 *(6.53)*	454.60 *(62.95)*	21.82 *(32.27)*	21.65 *(33.23)*
Layered Gorilla Even	17.83 *(14.79)*	6.79 *(6.45)*	454.60 *(62.95)*	21.80 *(32.04)*	21.54 *(32.61)*
Layered Gorilla Leading	17.74 *(14.72)*	6.76 *(6.41)*	454.60 *(62.95)*	24.37 *(35.15)*	23.52 *(34.17)*
Layered Gorilla Reweighed	18.44 *(14.27)*	**7.03** *(6.55)*	454.60 *(47.25)*	**21.12** *(32.03)*	**20.64** *(31.83)*
Layered Gorilla Optimized	18.33 *(15.15)*	**7.03** *(6.67)*	454.60 *(62.95)*	23.74 *(34.16)*	22.00 *(32.62)*
Layered Chimp	17.40 *(13.59)*	6.63 *(6.17)*	454.60 *(47.25)*	24.22 *(38.25)*	23.59 *(39.81)*
Layered $Chimp_{128}$	17.14 *(9.54)*	6.53 *(5.44)*	454.60 *(17.21)*	134.29 *(155.67)*	134.11 *(150.96)*
Layered Elf	**18.60** *(13.55)*	6.97 *(6.38)*	**1314.60** *(40.28)*	36.59 *(58.32)*	23.83 *(37.98)*
Layered Elf+	17.50 *(12.26)*	6.56 *(5.92)*	1237.93 *(33.41)*	30.53 *(52.53)*	23.65 *(39.60)*
Zstd (Level 3)	18.14	7.07	474.52	97.14	72.45

Table 3: Impact of the layered approach on the compression ratio of each algorithm. The results show a significant and consistent improvement across all compressors.

because it directly captures the bitwise behavior that XOR compressors exploit as shown in Figure 12:

- A massive spike in bin 64 (identical values) indicates an RLE-dominant block.
- Mass in the middle-zero bins (20–40) suggests stable, non-identical deltas, favoring compressors with good in-window performance such as Layered Gorilla Reweighed.
- Mass at low-zero bins (0–10) suggests noisy, volatile data, ideal for Layered $Chimp_{128}$.

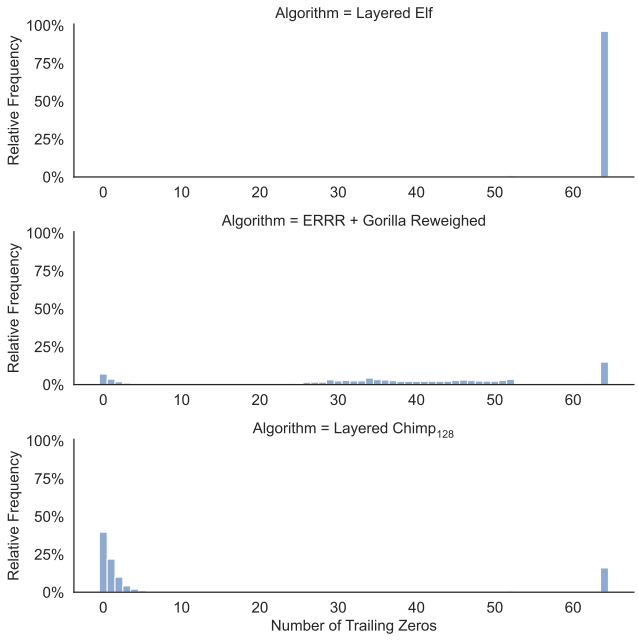

Figure 12: Three trailing zero histograms. Each represents the distribution of trailing zeros for the time series blocks the respective compressor performed best ("compressor profile").

3. Create Compressor Profiles. We use our dataset to create a "profile" for each of the 7 selected compressors which include ERRR + Gorilla Optimized, ERRR + Gorilla Reweighed , ERRR + $Chimp_{128}$, ERRR + Elf, Layered Gorilla Reweighed, Layered $Chimp_{128}$, and Layered Elf. A compressor's profile is its "ideal" fingerprint. We generate it by identifying all blocks in our dataset for which that compressor was the single best performer. We then aggregate (Z-normalize [24]) the 'trailing zero histograms' of all those 'best compressed' blocks. This results in 7 static "compressor profiles", each representing the prototypical data that a specific compressor is best-suited to handle.

4. Select Compressor by Matching Block to Best-Fitting Profile. The compressor selection process, which runs live as part of our ingestion pipeline, is now fast and simple:

(1) As the records of a new time series block arrive, we compute its 65-bin trailing zero histogram.
(2) We compare this new histogram to the 7 pre-computed, static compressor profiles. The comparison is done using Z-Normalized Euclidean distance [5, 15, 24]. This metric is crucial: by normalizing the histograms, we compare the shape of the distributions, not their absolute scale. This allows us to find the best-fitting profile pattern regardless of whether the block is sparse or dense (e.g., 60 vs. 240 records within a block).

The compressor whose profile has the smallest (i.e., *closest*) Euclidean distance to the block's histogram is selected. A few bits (e.g., 3 bits to identify 1 of 7) are prepended to the compressed stream to identify the chosen compressor, ensuring lossless decompression.

5.4.2 Evaluation. To properly evaluate our heuristic, we first partitioned our dataset by randomly selecting 20% of all time series blocks for a test set. The remaining 80% served as the training set, which was used to perform the offline step of generating the static compressor profiles. We generated these profiles twice, once for the 7 previously mentioned algorithms (and their variations), and once based only on the 3 compressors with the best combined coverage (ERRR + Gorilla Reweighed, Layered $Chimp_{128}$, and Layered Elf).

Set of Algorithms	Compression Ratio	
	Brute-force Approach	Heuristic Approach
Best Seven	22.04	21.57
Best Combined Three	21.84	21.23

Table 4: Comparison of the heuristic approach's compression ratio against the theoretical optimum (brute-force approach).

The heuristic's performance was measured using the unseen 20% test set. As shown in Table 4, our heuristic approach is exceptionally effective. For example, when adaptively selecting one of the 7 algorithms using XOR trailing zero profiles, it achieves a compression ratio of 21.57 that is within ≈2% of the theoretical brute-force approach (22.04) when run on the same test set.

This near-optimal result is a significant success for us. It represents a 13.7% storage improvement over using the best single-layered strategy (Layered Elf, which had a ratio of 18.60 compared to the now-achieved 21.57). Furthermore, our evaluation reveals that the heuristic approach is 5x to 15x faster than the brute-force approach and has a computational cost that is on par with running a single, standard compressor. The histogram is cheap to compute and the Euclidean distance comparison is trivial. This approach successfully combines the strengths of all compressors: near-optimal compression at a practical speed.

5.4.3 Practical Implementation. Our heuristic requires an entire block to be buffered in memory to compute its histogram. Considering streaming pipelines, we propose a two-stage strategy:

(1) **Ingest:** As data arrives, it is temporarily compressed using a fast, default compressor (e.g., ERRR + Gorilla Optimized or Layered Gorilla Optimized).
(2) **Storage:** Once the block is complete, the system decompresses this temporary version, applies the heuristic to select a more suitable compressor, and re-compresses the data for long-term storage. This approach provides low memory usage during ingest and optimal storage density at rest.

Other strategies could also mitigate this memory overhead. A partial block heuristic could select the compressor based on an initial subset of data (e.g., the first 60 records), which is then used for the remainder of the block. This reduces memory needs but risks a suboptimal choice if the initial subset is not statistically representative. A third approach, prediction based on the preceding block, would eliminate memory overhead entirely by using the previous block's profile to select the compressor for the current one. This is only viable if consecutive blocks are statistically similar, an assumption that warrants further investigation.

6 Discussion

The optimizations presented in this paper carry substantial practical implications. Our analysis is grounded in a large-scale, real-world application monitoring dataset, and we expect that data ingested into our production pipeline will exhibit similar characteristics (most notably high structural redundancy). The optimizations we developed are thus not merely theoretical. They represent a significant enhancement to our data compression, one specifically designed to tackle operational challenges of modern observability platforms.

Time series data is fundamental to application monitoring because it offers a simple and effective method for collecting recurring measurements. However, as modern software systems generate exponential volumes of this data, its storage becomes a significant challenge. In a cloud-native environment, this telemetry data incurs substantial and direct storage costs. These costs are compounded by the need for long-term data retention, where metrics are often stored for weeks, months, or even more than a year to support compliance requirements and long-term trend analysis. Over this extended period, the storage costs continuously accumulate.

Therefore, the performance gains demonstrated in this paper are critically important. The ≈23% improvement in compression ratio from our multi-layered strategy and the additional 13.7% improvement from the adaptive lightweight heuristic translate directly into tangible reductions in storage costs. For a large-scale platform such as Dynatrace, which ingests and processes massive volumes of telemetry data daily, these optimizations are a fundamental component for building a cost-effective and scalable system.

7 Related Work

The field of floating-point time series compression is subject to continuous development, driven by the need to minimize data volume and accelerate data retrieval speeds in data-intensive industries.

Benchmarks and Evaluation. Comprehensive benchmarks are critical for understanding the trade-offs between different compression techniques. FCBench [9], for instance, provides a cross-domain benchmark for lossless floating-point compression, evaluating both CPU and GPU-based approaches. It categorizes methods based on their input data types and predictor mechanisms. In particular, this benchmark highlighted the moderate performance of standard XOR-based schemes such as Gorilla [42] and Chimp_{128} [31] in a general-purpose setting, underscoring the challenge of comparing specialized XOR schemes with general-purpose algorithms. Critically, Hishida et al. [21] extended this analysis beyond simple throughput. Their framework investigates the subsequent impact of compression on query performance, evaluating algorithms on standard database operations (e.g., filters and aggregations) and machine learning queries (e.g., k-NN searches [23]) executed directly on the compressed data. Their work strongly reinforces the premise of our own: no single compression algorithm excels across all metrics. A technique that offers the best compression ratio may be suboptimal for query performance, highlighting the need for a balanced, domain-specific strategy.

Other Compression Techniques. Beyond the methods evaluated here, several other algorithms rely on XOR-based compression. For example, TSXor [7] utilizes a lightweight strategy targeting redundancy within a at most 127-value window, similar to Chimp_{128}, to combine effective compression with exceptionally high decompression speeds. Pongo [32] employs an erasing technique distinct from Elf [30], converting the binary fractional parts into decimals to efficiently encode values with short decimal sequences as integers. Camel [48] compresses the integer and fractional parts of a

floating-point number separately, rather than performing an XOR operation of the current value with its predecessor.

Furthermore, several novel techniques have moved entirely beyond the previous-value XOR model popularized by Gorilla [42] and Chimp [31]. Pcodec [34], for instance, is a non-XOR format that processes numerical data (both float and integer) using one of two preprocessing strategies: transforming values into uniformly distributed integers or enhancing value independence via delta encoding. This is followed by a general-purpose compressor such as LZ77 [50], LZ78 [51], BtrBlocks [25], or Zstd [12]. In addition, many columnar storage formats, such as Avro [13], ORC [22], Parquet [14], and Arrow [36], utilize optimized data layouts that allow general-purpose compressors to exploit data patterns more effectively.

Another category of compressors adaptively selects an appropriate compression strategy based on the characteristics of the underlying data. AFC [8] selects from four compression strategies based on observed data patterns, allowing it to adapt dynamically to changing data characteristics. ALP [4], an adaptive lossless floating-point algorithm optimized for vectorized execution. It achieves high speed and strong ratios by first identifying floating-point values that originate from decimal representations and losslessly encoding them as integers. It then applies vectorized compression directly to the remaining floating-point data, exploiting recurring patterns within these vectors. On the same datasets used in prior studies, ALP demonstrated superior speed and compression ratios compared to Gorilla [42], Chimp [31], and Elf [30] by better leveraging the data's intrinsic characteristics.

Machine Learning-Based Compression. Machine learning frameworks represent another significant research direction, often focusing on adaptive selection. LeCo [33] is an ML-based framework designed to automatically eliminate serial redundancy, acting as an intelligent preprocessor for lightweight lossless compression schemes such as frame-of-reference (FOR) [47], delta encoding [47], and run-length encoding (RLE) [45]. It stores prediction errors alongside the compressed values to ensure lossless decompression. Other models are more integrated. NeaTS [18] is a compression scheme that enables efficient random-access by approximating time series data using non-linear functions. While it can operate in a lossy mode with error guarantees, its lossless mode offers compression ratios comparable to other methods but with significantly faster decompression and random-access capabilities, making it ideal for real-time analysis. AMMMO [49] implements a sophisticated two-level compression model that leverages reinforcement learning to automatically tune parameters. It dynamically selects the most suitable compression scheme for each individual value, allowing for fine-grained pattern recognition. These approaches all demonstrate the power of adaptive, data-driven selection, a core principle that we apply in our own lightweight heuristic.

8 Limitation and Future Work

The findings of this study are subject to certain limitations. First, the dataset, while large, covers a single 24-hour period, which limits our ability to analyze long-term temporal patterns. Second, the data is sourced exclusively from internal services, and we cannot guarantee that data from other environments will exhibit the exact same statistical characteristics (even though, based on experience, we highly expect strong similarities). This specificity is also a core finding: the superiority of our heuristic is tied to Dynatrace's domain, i.e., working with application monitoring data, and the results are not intended to be universally generalizable. Finally, our evaluation did not include all recent algorithms, such as Camel [48], Pongo [32], and ALP [4]. We excluded Camel [48] and Pongo [32] from our evaluations because they diverge from the standard approach of applying XOR operations to the floating-point number as a whole. ALP [4] was also omitted because it operates on vectors of 1024 values, which fundamentally differs from the streaming paradigm utilized by the data processing pipeline used within Dynatrace. Because ALP [4] relies heavily on batch data to achieve its high performance, adapting it to a streaming context would likely eliminate its comparative advantages.

This work opens several avenues for future research. The proposed heuristic could be enhanced with more fine-grained features, such as time-weighted histograms or an adaptive RLE that compresses only parts of a record. Processing data in complete blocks facilitates the use of algorithms such as ALP [4], which exploit pattens within vectorized data. Even though our heuristic is already within 2% of the best achievable compression ratio using established XOR compressors, a natural and powerful extension would be to replace our rule-based heuristic with a full machine learning model. Such a model could be trained to learn more complex data patterns and predict the optimal compressor, with the ability to adapt over time as new data patterns emerge. Finally, we plan to expand our evaluation framework to include other state-of-the-art algorithms, such as Camel [48], Pongo [32], and ALP [4], and to investigate adapting our RLE-based approach to timestamp compression.

9 Conclusion

This work tackled the gap between the theory of time series compression and its practice in large-scale observability systems. We demonstrated that real-world application monitoring data as collected by Dynatrace exhibits fundamentally different characteristics than the common academic benchmarks used to design and test state-of-the-art compressors: high structural redundancy and a vast majority of identical consecutive values. This "dataset bias" renders the design assumptions of many algorithms, such as Chimp and Elf, suboptimal for this domain. Our contributions provided a series of layered, data-driven solutions. We first introduced an enhanced encoding for 'count aggregation records' and showed that the Gorilla algorithm family, with its 1-bit flag for identical values, is decisively superior for this data. We then demonstrated that a three-layer approach, prepending a record-level RLE to the stack, yields significant improvements in both storage (≈23%) and speed (≈34%). Our most significant contribution is a novel, heuristic-based framework for dynamic compressor selection. By using the XOR trailing zero distribution as a "fingerprint", our heuristic can match nearly every time series block to its optimal compressor. This adaptive strategy achieves near-optimal compression ratios—within 2% of the theoretical, brute-force maximum and representing a 12% storage reduction over the best single algorithm, while the computational cost is on par with a single compressor. We conclude that the most effective solution for real-world systems is not a single, universal

algorithm, but a data-driven, adaptive strategy that is tailored to the properties of its source domain. The optimizations presented in this paper yield tangible reductions in storage costs, particularly for large-scale observability platforms such as Dynatrace.

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. doi:10.1109/IEEESTD.2019.8766229

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 671–682. doi:10.1145/1142473.1142548

[3] Azim Afroozeh and Peter A. Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144. doi:10.14778/3598581.3598587

[4] Azim Afroozeh, Leonardo Kuffó, and Peter Boncz. 2024. ALP: Adaptive Lossless Floating-Point Compression. In *SIGMOD '24: Proceedings of the 2024 ACM SIGMOD International Conference on Management of Data*. ACM, 1–13. doi:10.1145/3626717

[5] Michael R. Berthold and Frank Höppner. 2016. On Clustering Time Series Using Euclidean Distance and Pearson Correlation. *CoRR* abs/1601.02213 (2016). arXiv:1601.02213 http://arxiv.org/abs/1601.02213

[6] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 283–296. doi:10.1145/1559845.1559877

[7] Andrea Bruno, Franco Maria Nardini, Giulio Ermanno Pibiri, Roberto Trani, and Rossano Venturini. 2021. TSXor: A Simple Time Series Compression Algorithm. In *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12944)*, Thierry Lecroq and Hélène Touzet (Eds.). Springer, 217–223. doi:10.1007/978-3-030-86692-1_18

[8] Haoyuan Chen, Liang Liu, Jingwen Meng, and Wanying Lu. 2024. AFC: An adaptive lossless floating-point compression algorithm in time series database. *Inf. Sci.* 654 (2024), 119847. doi:10.1016/J.INS.2023.119847

[9] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. 2024. FCBench: Cross-Domain Benchmarking of Lossless Compression for Floating-point Data. *Proc. VLDB Endow.* 17, 6 (2024), 1418–1431. doi:10.14778/3648160.3648180

[10] Giacomo Chiarot and Claudio Silvestri. 2023. Time Series Compression Survey. *ACM Comput. Surv.* 55, 10 (2023), 198:1–198:32. doi:10.1145/3560814

[11] Yann Collet. 2014. *LZ4 - Extremely Fast Compression.* Retrieved September 26, 2025 from https://github.com/lz4/lz4

[12] Yann Collet. 2015. *Zstandard - Fast real-time compression algorithm.* Retrieved September 26, 2025 from https://github.com/facebook/zstd

[13] Doug Cutting. 2010. *Apache Avro.* Retrieved March 8, 2026 from https://github.com/apache/avro

[14] Julien Le Dem and Nong Li. 2015. *Apache Parquet.* Retrieved March 8, 2026 from https://github.com/apache/parquet-format/

[15] Luong Van Do and Duong Tuan Anh. 2016. Some improvements for time series subsequence join based on pearson correlation coefficients. In *Proceedings of the Seventh Symposium on Information and Communication Technology, SoICT 2016, Ho Chi Minh City, Vietnam, December 8-9, 2016*, Huynh Quyet Thang and Nguyen Manh Hung (Eds.). ACM, 58–65. doi:10.1145/3011077.3011086

[16] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, Susan Darling Urban and Elisa Bertino (Eds.). IEEE Computer Society, 370–379. doi:10.1109/ICDE.1998.655800

[17] Google. 2025. *Snappy - A fast compressor/decompressor.* Retrieved September 26, 2025 from https://github.com/google/snappy

[18] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2024. Learned Compression of Nonlinear Time Series With Random Access. *CoRR* abs/2412.16266 (2024). arXiv:2412.16266 doi:10.48550/ARXIV.2412.16266

[19] Moser Herwig and Schwarzbauer Günter. 2025. Differential Encoding For Time Series With Complex Payload. https://patents.google.com/patent/US20250021577A1 U.S. Patent No. 20250021577A1, Filed July 3rd., 2024, Issued Jan. 16th., 2025.

[20] Moser Herwig and Schwarzbauer Günter. 2025. Differential Encoding For Time Series With Complex Payload. https://patents.google.com/patent/EP4492691A3 EU Patent No. 4492691A3, Filed July 10th., 2024, Issued Mar. 26th., 2025.

[21] Kaisei Hishida, Chunwei Liu, John Paparrizos, and Aaron Elmore. 2025. Beyond Compression: A Comprehensive Evaluation of Lossless Floating-Point Compression. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4396–4409.

[22] Hortonworks and Facebook. 2015. *Apache ORC (Optimized Row Columnar).* Retrieved March 8, 2026 from https://github.com/apache/orc

[23] Shrikant Kashyap and Panagiotis Karras. 2011. Scalable kNN search on vertically stored time series. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, Chid Apté, Joydeep Ghosh, and Padhraic Smyth (Eds.). ACM, 1334–1342. doi:10.1145/2020408.2020607

[24] Eamonn J. Keogh and Shruti Kasetty. 2002. On the need for time series data mining benchmarks: a survey and empirical demonstration. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*. ACM, 102–111. doi:10.1145/775047.775062

[25] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. doi:10.1145/3589263

[26] Alexandros Labrinidis and H. V. Jagadish. 2012. Challenges and Opportunities with Big Data. *Proc. VLDB Endow.* 5, 12 (2012), 2032–2033. doi:10.14778/2367502.2367572

[27] Seungyeon Lee, Jusuk Lee, Yongmin Kim, Kicheol Park, Jiman Hong, and Junyoung Heo. 2020. Efficient scheme for compressing and transferring data in hadoop clusters. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini (Eds.). ACM, 1256–1263. doi:10.1145/3341105.3374044

[28] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* 45, 1 (2015), 1–29. doi:10.1002/SPE.2203

[29] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, Songtao Guo, Ming Zhang, and Yu Zheng. 2023. Erasing-based lossless compression method for streaming floating-point time series. *CoRR* abs/2306.16053 (2023). arXiv:2306.16053 doi:10.48550/ARXIV.2306.16053

[30] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7 (2023), 1763–1776. doi:10.14778/3587136.3587149

[31] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proc. VLDB Endow.* 15, 11 (2022), 3058–3070. doi:10.14778/3551793.3551852

[32] Yufeng Liu, Yao Shen, Fenghua Zhang, and Feiteng Huang. 2025. Pongo: Efficient Lossless Floating Point Compression. *Cloud Computing and Data Science* (2025), 292–311.

[33] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1 (2024), 65:1–65:28. doi:10.1145/3639320

[34] Martin Loncaric, Niels Jeppesen, and Ben Zinberg. 2025. Pcodec: Better Compression for Numerical Sequences. *CoRR* abs/2502.06112 (2025). arXiv:2502.06112 doi:10.48550/ARXIV.2502.06112

[35] Warren S. McCulloch and Walter H. Pitts. 1990. A Logical Calculus of the Ideas Immanent in Nervous Activity. In *The Philosophy of Artificial Intelligence*, Margaret A. Boden (Ed.). Oxford University Press, 22–39.

[36] Wes McKinney. 2016. *Apache Arrow.* Retrieved March 8, 2026 from https://github.com/apache/arrow

[37] National Ecological Observatory Network (NEON). 2022. *2D wind speed and direction (DP1.00001.001).* Retrieved August 2, 2025 from https://data.neonscience.org/data-products/DP1.00001.001/RELEASE-2022

[38] National Ecological Observatory Network (NEON). 2022. *Barometric pressure (DP1.00004.001).* Retrieved August 2, 2025 from https://data.neonscience.org/data-products/DP1.00004.001/RELEASE-2022

[39] National Ecological Observatory Network (NEON). 2022. *Dust and particulate size distribution (DP1.00017.001).* Retrieved August 2, 2025 from https://data.neonscience.org/data-products/DP1.00017.001/RELEASE-2022

[40] National Ecological Observatory Network (NEON). 2022. *IR biological temperature (DP1.00005.001).* Retrieved August 2, 2025 from https://data.neonscience.org/data-products/DP1.00005.001/RELEASE-2022

[41] National Ecological Observatory Network (NEON). 2022. *Relative humidity above water on-buoy (DP1.20271.001).* Retrieved August 2, 2025 from https://data.neonscience.org/data-products/DP1.20271.001/RELEASE-2022

[42] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827. doi:10.14778/2824032.2824078

[43] J. Ross Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (1986), 81–106. doi:10.1023/A:1022643204877

[44] Lior Rokach and Oded Maimon. 2005. Decision Trees. In *The Data Mining and Knowledge Discovery Handbook*, Oded Maimon and Lior Rokach (Eds.). Springer, 165–192.

[45] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (1993), 31–39. doi:10.1145/163090.163096

[46] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117. doi:10.1016/J.NEUNET.2014.09.003

[47] Julia Spindler, Philipp Fent, Adrian Riedl, and Thomas Neumann. 2024. Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression?. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*. VLDB.org. https://vldb.org/workshops/2024/proceedings/ADMS/ADMS24_02.pdf

[48] Yuanyuan Yao, Lu Chen, Ziquan Fang, Yunjun Gao, Christian S. Jensen, and Tianyi Li. 2024. *Camel:* Efficient Compression of Floating-Point Time Series. *Proc. ACM Manag. Data* 2, 6 (2024), 227:1–227:26. doi:10.1145/3698802

[49] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-Level Data Compression using Machine Learning in Time Series Database. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1333–1344. doi:10.1109/ICDE48307.2020.00119

[50] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343. doi:10.1109/TIT.1977.1055714

[51] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536. doi:10.1109/TIT.1978.1055934

[52] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 59. doi:10.1109/ICDE.2006.150

Received 17 November 2025