Beyond Compression: A Comprehensive Evaluation of Lossless Floating-Point Compression

Kaisei Hishida
Keio University
kaisei.hishida@keio.jp

Chunwei Liu
MIT CSAIL
chunwei@csail.mit.edu

John Paparrizos
The Ohio State University and
Aristotle University of Thessaloniki
john@paparrizos.org

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

ABSTRACT

Modern data-intensive applications generate vast amounts of floating-point data, essential for fields like databases and machine learning. While many compression techniques focus on space efficiency, there is a lack of benchmarks evaluating both compression and query performance, especially in areas like in-situ query execution on compressed data and machine learning tasks such as distance measurement and k-nearest neighbors (k-NN) in Retrieval-Augmented Generation (RAG) systems. This paper addresses this gap by evaluating popular lossless floating-point compression methods on three key factors: compression efficiency, database operations performance, and machine learning query performance. We implemented these techniques in Rust and integrated them into an open-source library for use with columnar engines. Our comparison highlights trade-offs between compression efficiency and query performance, showing that no single approach excels in all areas, and some methods trade off compression for slower performance.

PVLDB Reference Format:
Kaisei Hishida, Chunwei Liu, John Paparrizos, and Aaron J. Elmore. Beyond Compression: A Comprehensive Evaluation of Lossless Floating-Point Compression. PVLDB, 18(11): 4396 - 4409, 2025.
doi:10.14778/3749646.3749701

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lemolatoon/ebi.

1 INTRODUCTION

Database systems continue to be critical for large-scale operational (OLTP), analytical (OLAP), Internet of Things (IoT), and machine learning systems due to their ability to efficiently manage, compress, and query large volumes of structured data. For systems that store and query large amounts of data, compression is critical for managing storage footprint and efficient query performance [8, 9, 43]. Although data systems support diverse data types, floating-point values are increasingly critical, especially in machine-generated data and machine learning features. For example, sensor data from IoT devices often generate large amounts of floating-point data with precision determined by the sensor hardware, which is well suited for efficient compression [13, 47, 53, 54, 60, 79]. Similarly, the weights and embedding generated from machine learning models, stored as floating-point values, continue to grow each year. IoT devices alone are expected to generate approximately 90 zettabytes of data, underscoring the escalating scale of data production and the corresponding need for advanced compression methods [88]. Given the significant volume and growth of floating-point data, this paper focuses on the compression of such data in columnar formats, highlighting the need for advanced compression techniques to handle the unique challenges presented by floating-point data.

Compression techniques are classified by how they handle data loss. *Lossy* compression techniques can lose fidelity from the original uncompressed data. We focus on *lossless* compression, which does not lose any information after compression. However, within the database community, there are conflicting views on the exact use of the term. To reason about *lossless* data, you must think about how the floating-point values are produced or provided. While most processors handle real numbers using IEEE 754's floating-point representations [41], it tends to be just the best approximations to the original real numbers as is shown in Figure 1. We can define *lossless* either restoring IEEE 754's floating-pint representations or restoring the original real numbers with the given precision as shown in Figure 1. In this paper, we define a lossless floating-point compression technique as one that loses no information on a defined precision given by the schema. In other words, we assume an attribute defines the scale (min and max significant digits): how many digits should be retained after the decimal point. A compression scheme that never loses information at that fidelity is lossless.

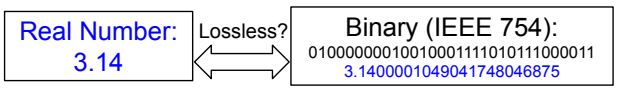

Figure 1: There is not always an exact mapping between real numbers and their binary encodings, even with the widely used IEEE floating-point format.

In the past decade, there has been a notable increase in the development of lossless floating-point compression methods for data systems, with a significant surge in activity over the past three years. Noteworthy examples include Gorilla [86], Buff [53],

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749701

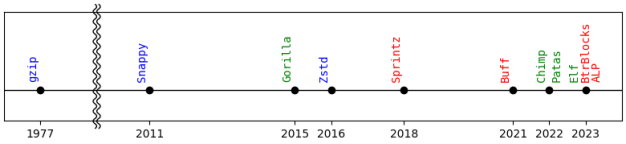

Figure 2: Timeline of lossless compression methods. Methods with blue are general-purpose, green are XOR-based, and red are quantization-based. Figure inspired by FCBench [23].

Chimp [52], Elf [50], and ALP [11], which are specialized methods for compressing floating-point time series data. In addition, many systems employ general-purpose (byte-oriented) compression methods such as Gzip [28], Snappy [39], and Zlib [36], treating floating point records as byte arrays. Figure 2 shows when these methods (along with an adapted integer compression technique) have been introduced and highlights the recent surge of database papers focusing on float compression.

While a recent study [23] thoroughly examined the effectiveness of floating point compression methods, we believe this study was not exhaustive for data systems researchers for the following reasons. First, this study has a high-performance computing (HPC) perspective and as a result focused on compression ratios, compression throughput, and decompression throughput, but did not evaluate data specific operations—including SQL workloads (e.g., filtering, aggregation) and emerging machine learning tasks (e.g., embeddings, feature stores)—that the community has long been optimizing [8, 9, 44, 56, 98]. Second, given the HPC orientation a heavy emphasis was placed on GPU-based techniques. Our belief is that GPUs are not widely exploited by database systems and are always present on general purpose computing devices (i.e., commodity servers, IoT devices, and Edge computing). Third, the study evaluated across various programming environments, including CUDA C/C++, Go, Python, Java, and Rust, and did not fully optimize some CPU methods. We unify methods into our framework with Rust, which eliminates overheads such as garbage collectors and leads to a fair comparisons.

A recent compression method [11] evaluates various lossless floating point compression methods; however, their evaluation primarily focuses on compression performance and does not adequately consider integrated query execution. In contrast, our benchmark framework evaluates both compression efficiency and query performance, including in-situ query execution.

Given the surge of database-centric float compression techniques, we believe that the community would therefore benefit from a short survey, comprehensive study, and benchmark with a unified framework and an emphasis on common database (including in-situ operations on compressed data [8]) and machine-learning operations. Our study provides not only the benchmark result with the unified setup, but also the library of experiments usable for future developments and research. Figure 3, which normalizes results where higher is proportionally better, demonstrates that there are tradeoffs between the proposed approaches. To evaluate the efficiency of floating-point compression techniques, we developed a library written in Rust: EBI[1]. This library is designed for the evaluation of floating-point compression and integration into existing data systems.

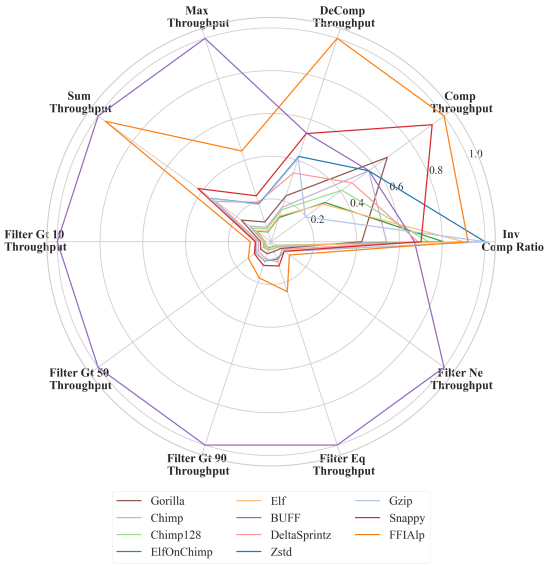

Figure 3: Radar Chart for All Metrics.

This paper is organized as follows: we review the background on floating-point representation, the columnar storage format and prior studies (Section 2). In Section 3, we explore the compression methods in detail. We then introduce the library developed for this work (Section 4) and proceed to the benchmark setup (Section 5). The results of the benchmarking are presented in Section 6, followed by a summary of the findings in Section 7.
The main contributions of this paper are:

- We provide an exact comparison of floating-point compression methods using a unified code base and file format, ensuring consistency across all experiments.
- A comprehensive study that evaluates both database query workloads and machine learning tasks—measuring performance and compression efficiency for a variety of floating-point compression techniques—applied to diverse datasets, including time-series data and ML embeddings.
- A complete file format and floating-point compression library that is provided as an open source[2] to facilitate further research or practical implementation.

2 BACKGROUND AND RELATED WORK

In this section, we provide a brief overview of floating-point representations and columnar formats. For more details on these topics, we refer the reader to canonical publications [8, 9, 38].

2.1 Floating Point Representation

To represent decimal values in processors, two popular representations are commonly used: fixed and floating-point representation.

[1]Ebi means "shrimp" in Japanese
[2]Available at https://github.com/lemolatoon/ebi

2.1.1 Fixed Representation. Fixed representation consists of the tuple $\langle sign, integral, fractional \rangle$. The *sign* bit indicates the sign, 0 for +, 1 for −. The *integral* bits represent the integral part of the decimal value. The *fractional* bits represents the fractional part of the decimal value. Thus, the decimal value V is represented as follows: $V = (-1)^{sign} \times integral.fractional$

The *integral* and *fractional* parts must have specific bit lengths. For example, when representing 3.14 with *integral* and *fractional* bit lengths of 2 and 6, respectively, *integral* and *fractional* will be 11 and 001110 respectively. The length of the *fractional* part is also referred to as precision in some contexts. A real-world example of fixed representation can be found in databases such as PostgreSQL, which provides the `numeric` data type [4]. This data type allows users to specify the bit lengths for both *integral* and *fractional* parts at the column level.

2.1.2 Floating-Point Representation. Floating-point representation consists of the tuple $\langle sign, exponent, significand \rangle$. The *sign* bit is the same as in the fixed representation. In floating-point representation, a base must be chosen. The decimal value V with the radix b is represented as follows.

$$V = (-1)^{sign} \times b^{exponent} \times significand$$

The de facto standard floating-point representation is IEEE 754. IEEE 754 defines multiple floating-point encodings for a given bit length of a decimal value. In this paper, we focus on 64-bit floating-point format, also known as *Double*. For *Double* or *binary64*, the radix is 2. The 64-bits representation consists of 1 bit for *sign*, 0 for +, 1 for −, 11 bits for the *exponent*, and 52 bits for *significand*. The *exponent* is biased by 1023. The *significand* is normalized, meaning it always starts with the bit 1, which is implicitly omitted in the bit representation. The decimal value V in IEEE 754 *binary64* will be represented as follows.

$$V = (-1)^{sign} \times b^{exponent-1023} \times 1.significand$$

A decimal value does not always have an exact floating-point representation. For example, consider 3.14. Since the IEEE 754 uses radix 2, it is not possible to represent 3.14 precisely in binary using limited *significand* bit length of 53. Approximately, 3.14 is represented as follows. Note that the *significand* part is shown in binary.

$$3.14 = (-1)^1 \times 2^{1024-1023} \times$$
$$1.1001000111101011100001010001111010111000010100011111$$

2.2 Column-Oriented Formats and the EBI Abstraction

OLAP engines rely on column-oriented formats, such as Apache Parquet, Apache ORC (on-disk), and Apache Arrow (in-memory) for physical data layout [33–35, 55, 92]. Despite their different names, the three formats share a common three-level hierarchy. A file (or in-memory buffer) is divided into *row batches*, called *row groups*, *stripes*, or *record batches*, respectively. Each batch contains one contiguous *column chunk* per attribute, and every chunk is further partitioned into *pages*. Lightweight encodings, such as dictionary, RLE, delta, and bit-packing, are applied first. Parquet and ORC may then compress each page or chunk with a general-purpose codec (e.g., Snappy or GZip) to reduce I/O volume, whereas Arrow omits this second stage to preserve fast zero-copy access semantics.

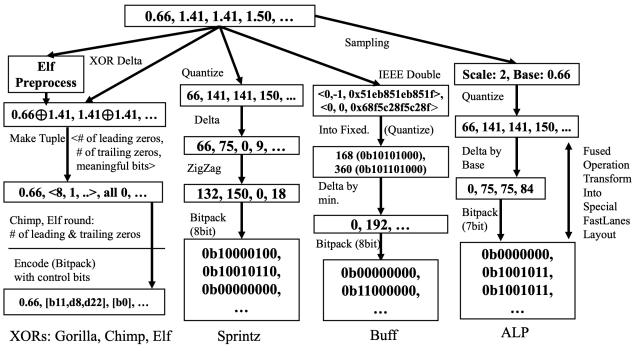

Figure 4: Floating-Point Compression Visualization

EBI preserves the familiar row-batch → chunk → page hierarchy and reuses the metadata fields expected by existing formats, while discarding control information, such as nested-schema bookkeeping, Bloom filters, and encryption keys, that is orthogonal to compression. An EBI file begins with a fixed-size header that records the selected floating-point codec and the chunk-size policy, continues with a sequence of independent chunks, and ends with a footer that stores per-chunk logical and physical offsets together with basic statistics. Each chunk is encoded by one floating-point compressor; no second-stage general-purpose codec is applied and no data cross-chunk boundaries, enabling true in-situ decompression. Because the chunk size can be specified as either a record count or a byte budget, a chunk can be surfaced as a Parquet page, an ORC stripe or chunk, or an Arrow buffer through a thin wrapper, leaving the payload untouched. This lean yet faithful abstraction lets us benchmark a wide spectrum of numerical compressors in a setting that aligns directly with mainstream columnar workflows.

3 FLOATING-POINT COMPRESSION

As noted in Section 1, we limit our study to non-GPU-optimized, lossless floating-point compression methods (shown in Figure 2). In this section, we classify these methods on their characteristics, and review each method in detail. Figure 4 illustrates examples of each compression method.

3.1 XOR Family Methods

The first category is the XOR family. These methods extensively exploit the XOR operation (⊕) on the bit representation of floating-point values.

3.1.1 Gorilla. The pioneering method of the XOR family is Gorilla [86]. Gorilla is designed for storing time-series floating-point values. Gorilla leverages two key observations: 1) consecutive records in a time series tend to be similar, and 2) the XOR result of similar floating-point values often contains many leading and trailing zeros. This principle underlies all XOR-based compression methods. For example, consider the values 24.3 and 23.3. Table 1 presents their IEEE 754 double representations along with their XOR result.

Table 1: XOR result of 24.2 and 24.3 in binary

Number	Binary Representation
24.2	0b0100000000011100000110011001100110011001100110011001100110011
23.3	0b0100000000011011101001100110011001100110011001100110011001101
24.2 ⊕ 23.3	0b0000000000000111101111111111111111111111111111111111111111110

In this case, 24.2 ⊕ 23.3 has 12 leading zeros, and 1 trailing zero. XOR family methods take advantage of these zeros. The bits between the leading and trailing zeros are called meaningful bits. 24.2 ⊕ 23.3 consists of 51 meaningful bits.

Specifically, for each record, Gorilla computes XOR with the previous record and encodes the XOR result. The detailed encoding process for XOR results is as follows:

(1) The first value is stored uncompressed using 64 bits.
(2) If the XOR result is zero (i.e., the value is identical to the previous record), store '0' in 1 bit.
(3) If the XOR result is non-zero, compute the number of leading and trailing zeros, then store '1' in 1 bit, followed by either (a) or (b):
    (a) If the meaningful bits fit within the length of the previously stored meaningful bits, store '0' in 1 bit, followed by meaningful bits, using the same encoding as the previous value.
    (b) Otherwise, store '1' in 1 bit. Store the the number of leading zeros in the next 5 bits, the length of meaningful bits in the next 6 bits, and the meaningful bits.

3.1.2 Chimp. Chimp [52] is another XOR family floating-point compression method inspired by Gorilla. Like Gorilla, Chimp utilizes the XOR operation, but it encodes the XOR result differently. Gorilla directly stores the number of leading and trailing zeros. Whereas Chimp rounds the number of leading zeros. For example, if the XOR result has fewer than 8 leading zeros, it is treated as having 0 leading zeros. If the number of leading zeros falls between 8 and 11, it is treated as 8 leading zeros. This rounding technique mitigates the fluctuations in the number of leading zeros.

With the rounding the number of leading zeros, Chimp also encodes the XOR result differently:

(1) The first value is stored uncompressed using 64 bits.
(2) For subsequent values, encoding is determined based on the trailing count, (a) or (b).
    (a) If the trailing count is greater than 6, store '0' in 1 bit, followed by further two branches:
        (i) If the XOR result is zero, store '0' in the next 1 bit.
        (ii) If the XOR result is non-zero, store '1' in the 1 bit. Store the leading count in the next 3 bits, and the center count in the next 6 bits. Finally, store the center bits.
    (b) If the trailing count is less than or equal to 6, store '1' in 1 bit, followed by further two branches:
        (i) If the leading count is the same as in the previous value, store '0' in the next 1 bit, then store the non-leading bits (meaningful bits and trailing 0s).
        (ii) If the leading count is different from the previous value, store '1' in the next 1 bit. Then, store the leading count in the next 3 bits, followed by the non-leading bits.

Chimp has a variant called Chimp128. Unlike Chimp, Chimp128 computes the XOR not with the previous record, but with a record from the last 128 records that produces the maximum number of trailing zeros. Although performing a linear search over all 128 previous records to find the one with the most trailing zeros is computationally expensive, Chimp128 optimizes this process using a lookup table. This table maps the last 14 bits of a record to the index of the most recent record with the same 14-bit trailing pattern. Chimp128 adopts a record from the last 128 records if and only if the candidate shares at least 13 trailing bits with the current record. Otherwise, it defaults to using the previous record as the XOR target, just like the original Chimp. Thus, the differences in XOR result encoding are as follows:

(1) The threshold of the trailing count is 6 in Chimp, but 8 in Chimp128, which affects the encoding process in (2a-b).
(2) If the trailing count is greater than or equal to 13, the XOR target index is stored in $log_2 128$ bits after control bits, modifying the behavior in (2-a-i) and (2-a-ii).

Patas [7] is a variant of Chimp introduced in a DuckDB [3] pull request. Patas stores the significant bits in a byte-aligned manner, enabling faster reading.

3.1.3 Elf. Elf [50] is another XOR family floating-point compression method that includes additional preprocessing steps. Elf introduces a preprocessing technique that generates more trailing zeros in each record by simply erasing the trailing bits. Elf employs a complex algorithm for erasing and restoring bits, enabling it to perform lossless compression. The Elf preprocessor works in conjunction with the internal XOR compressor as follows:

(1) Let the decimal precision of the record be n, and replace the trailing bits with zeros, effectively erasing them, as long as the difference from the original value remains within 10^{-n}.
(2) Encode and store the metadata α and β to restore the erased bits during decompressing.
(3) Encode and store the modified floating-point record using the internal XOR compressor.

Elf can potentially use the encoders from Chimp or Gorilla; however it also provides its own XOR result encoder. The Elf Encoder adopts leading zeros rounding, similar Chimp. The Elf Encoder encodes the XOR result as follows. Note that the trailing bits of each value have already been erased during Elf's preprocessing:

(1) The first value is stored uncompressed using 64 bits.
(2) If the XOR result is zero, store '01' in the next 2 bits.
(3) Otherwise, compute the rounded leading count, the trailing count and the center count, defined as $64 - \texttt{lead} - \texttt{trail}$. The process branches into three cases:
    (a) If the leading count is the same as in the previous value and the trailing count is ≥ to the previous trailing count, store '00' in the next 2 bits, and store the center bits using the information of the previous trailing count.
    (b) If the center count is ≤ 16, then store '10' in the next 2 bits, then store the leading count in 3 bits and the trailing count in 4 bits. Finally, store the center bits.
    (c) Otherwise, store '11' in the next 2 bits, then store the leading count in 3 bits and the center count in 6 bits. Finally, store the center bits.

While XOR based approaches carefully compress each record individually, they face challenges in [de]compression throughput due to the following reasons: 1) they involve complex control logic, leading to many branching during [de]compression, and 2) each record depends on the previous one for decompression, meaning they do not support random access.

3.2 Quantization Family Methods

Quantization-based floating-point compression methods quantize the floating-point values by multiplying them by an integer constant, converting them into integers, and apply integer compression techniques. At first glance, these methods may not appear *lossless*. However, under our definition of *lossless* in Section 1, the scale factor is provided by the scheme, allowing quantized integers to be reconstructed using only the scale information. We evaluated three quantization-based floating-point compression methods: Sprintz [13], Buff [53], and ALP [11]. In particular, Sprintz and Buff require a scale factor to quantize the dataset. Buff has a variant called Buff with outliers, which handles outliers that have excessively high precision to be quantized using the predefined scale factor. While ALP is lossless in the IEEE 754 representation, Sprintz and Buff are lossless in terms of decimal representations. ALP achieves IEEE 754 losslessness by incorporating scale detection through sampling and outlier handling. It is worth emphasizing that different techniques employ distinct quantization methods: Buff uses bitwise operations for quantization, while others apply mathematical scaling to quantize values. Additionally, Buff and ALP adopt SIMD-enabled decompression for improved performance.

First, we explain commonly used integer encodings, which serves as the building blocks for each compression method.

Bit-Packing is an encoding technique that stores each integer using the minimum number of bits necessary, based on the maximum value. When an unsigned integer is stored in n bits, the encoded value ranges from 0 to $2^n - 1$. Conversely, if the maximum integer is less than 2^n, all integers can be stored using n bits per value. This concept is known as bit-packing.

Zigzag maps signed integers to unsigned integers, ensuring that small absolute values remain small after encoding. For example, it maps 0 to 0, -1 to 1 and 1 to 2.

Delta reduces integer values relative to a reference value, making them smaller and suitable for bit-packing as an additional compression step. Delta encoding computes the difference of a value against some reference value, which can include fixed, per chunk, learned, the prior value, and others. For example, if the integers' value ranges from 100,000 to 100,128 and 100,000 is used as the reference value, the resulting differences will range from 0 and 128. If the same data type used to store both the original and delta-encoded values, this approach alone provides no compression. Therefore, delta encoding is often combined with bit-packing or run-length encoding (i.e., 200 contiguous zeros can be represented as (0,200)).

Dictionary encoding, employs a bijective mapping to replace attributes from a large domain, such as strings, integers, or floats, with a finite code domain, typically integers. This method enhances compression performance by converting large values into smaller integers, which can then be further compressed using integer encoding. Dictionary encoding is particularly effective when the input sequence has low cardinality relative to the number of records.

3.2.1 Sprintz. Sprintz [13] was originally designed as an integer compression method but can be applied to floating-point data through quantization. Sprintz utilizes a forecaster to predict the value based on the previous value and encode the error between the predicted value and the actual value. Note that Sprintz is compatible with various prediction models, including FCM and DFCM [87]. However, since Sprintz has demonstrated superior performance so we do not explicitly evaluate these older methods. This paper evaluates Sprintz with the Delta forecaster, which assumes that each value is identical to the preceding one. We refer to this variant as *DeltaSprintz*. The error sequence produced by the Delta forecaster is equivalent to the difference sequence. Sprintz further encodes the error sequence as follows: 1) apply zigzag encoding for each error to map signed integers to unsigned integers, and 2) perform bit-packing, determining the number of bits required based on the maximum value on the error sequence. Sprintz is designed for low memory usage and high decompression speed.

3.2.2 Buff. Buff [53] is a quantization-based floating-point compression method. Unlike traditional quantization, Buff scales values by 2^n instead of 10^n . Buff determines the number of bits required to preserve the specified precision. Rather then multiplying 2^n as a floating-point value, Buff computes its quantized integer using bit masking and bit shifting on the significant bits of the floating-point representation. Bitwise operations are more efficient than multiplication for this process.

If the value is 3.14 and the precision is 2, the number of bits required for the fractional part is 8, as determined by the paper [53]. In the IEEE 754 double format, the exponent bits for 3.14 are 1024, which corresponds to an exponent of 1 since the exponent in double is biased by 1023. During Buff's quantization process, the first mantissa bits are extracted, with a length equal to the exponent value plus the number of bits required for precision. The hidden bit is then added to the left, as illustrated in Table 2. This resulting quantized value can now be treated as an integer.

Table 2: Representations of 3.14

Exponent	10000000000 (1024 in decimal)
Mantissa (including hidden 1)	1.10010001111010111000010100...
Extracted Mantissa Bits ('1' + first 1+8 bits)	
Bits extracted from mantissa	11 00100011 (binary)
Integer from extracted bits	803 (decimal)

Buff applies additional integer encodings to the sequence of the quantized values. First, it performs delta encoding by subtracting the minimum value from all quantized results. Then Buff applies bit-packing to minimize the number of bits required for storage. Buff adopts a similar approach to BitWeaving [51] and ByteSlice [32], organizing bit-packed records in a byte-aligned format. Specifically, Buff splits each record into byte-sized segments. For example, if a record requires 17 bits, Buff splits it into 8 + 8 + 1 bits. Buff then stores records byte by byte, a method referred to as subcolumns. In the case of 17-bit bit-packing with n records: 1) Buff first stores the 8-bit subcolumns for records 1 to n. 2) It stores the next 8-bit subcolumns for records 1 to n. 3) Buff stores the remaining 1-bit subcolumns for the record 1 to n.

Buff enables in-situ query execution through its unique storage format. During decompression, it is possible to read only the first k bits to achieve the required precision. This allows for materialization with varying precision and provides significant speedups. By leveraging the base value from delta encoding, Buff enables faster sum calculations using integer arithmetic. Additionally, for maximum or filter queries, Buff improves efficiency by skipping unnecessary comparisons on already disqualified records. It achieves this by effectively managing qualified records after each subcolumn evaluation. SIMD instructions are also supported for materialization, aggregation, and filter operations.

Buff introduced the concept of just-enough precision for floating-point compression with in-situ query execution. It also proposed a variation of Gorilla, called gorillabd, which applies bounded precision to floating-point values before using Gorilla compression. This approach increseas the number of trailing zeros in the Gorilla XOR results, a technique later adopted in Chimp and Elf. Additionally, Buff employs sparse coding to handle outliers, though this feature is not evaluated in this work for the sake of simplicity.

3.2.3 ALP. ALP [11] leverages FastLanes [10] to achive significant performance benefits from SIMD instructions. ALP dynamically switches between the compression methods using adaptive sampling, where a subset of data is sampled to determine the most suitable method. The default ALP encoding consists of two steps: 1) ALP determines the scale factor for quantization based on sampling. 2) It uses FastLanes's FFOR to perform delta encoding and bit-packing in a fused, vectorized manner, processing data in chunks of 1024 records (equal to the vector width). ALP's quantization process is achieved by two-step scaling as follows where n represents the original floating-point value, e is the scale factor for the first step and f is the scale factor for the second step. The quantization process is defined as: $\texttt{ALP}_{enc} = \text{round}(n \times 10^e \times 10^{-f})$.

The dequantization process follows a similar approach, where d is the quantized integer: $\texttt{ALP}_{dec} = d \times 10^f \times 10^{-e}$.

Due to the excessively high precision of n, it is sometimes possible that $\texttt{ALP}_{dec}[d := \texttt{ALP}_{enc}] \neq n$. ALP handles these cases as exceptions, storing them separately, similar to Buff with outliers.

When adaptive sampling determines that the default ALP compression is not ideal for the dataset, ALP falls back to $\texttt{ALP}_{rd}$, a compression method designed for real doubles. $\texttt{ALP}_{rd}$ exploits the front bits similarity on double. First, it determines p, the position to split the double into two parts, as a result of the adaptive sampling. Then, it applies dictionary encoding and bit-packing to the left part, while applying only bit-packing to the right part.

3.3 Other Float Compression Methods

Additional floating-point compression techniques we do not evaluate in this paper. While we focus on CPU-based compression techniques, numerous GPU-based compression approaches have also been proposed [23, 89]. In 2023, FastLanes [10] was introduced as a building block for SIMD accelerated integer encodings, including Dict, FOR, Delta, and RLE. BtrBlocks [48] is another compression approach, which adaptively selects a compression scheme and encode values multiple times. BtrBlocks introduces PseudoDecimal as a floating-point encoding method that determines the scale factor for quantization on a per-record basis. Additionally, MOST [97] is a recent example of model-based lossy compression method for floating-point values.

3.4 General-Purpose Methods

General-purpose compression methods are designed for arbitrary byte arrays or encodings, not just floating-point data. In this paper, we evaluate Gzip [27], Snappy [39], and Zstd [24]. All three methods belong to the LZ77 family, encoding bytes using a sliding window over the input byte stream as their dictionary. Bytes are compressed by representing them as sequence information, which consists of: 1) position and length in the sliding window, and 2) a literal indicating the next bytes. The method used to encode this sequence information varies among Gzip, Snappy, and Zstd. Gzip encodes sequence information using Huffman encoding. Snappy does not apply further encoding; it directly stores the sequence information to prioritize high throughput. Zstd employs Huffman encoding for literal data, while using finite state entropy encoding for the remaining sequence information. This approach enables faster [de]compression throughput than Gzip. Note that these techniques are traditionally referred to as dictionary coders. However, we avoid this term to distinguish them from dictionary encoding used in columnar databases [9].

4 EBI FILE FORMAT AND LIBRARY

Our library EBI includes a corresponding columnar file format designed to integrate into existing data systems. It compresses data into chunks, as described in Section 2.2. The library provides low-level and high-level APIs, supporting benchmarking and integration into existing data processing frameworks. Released under the MIT license, EBI can be used freely for activities such as benchmarking, development, and further research. Three key motivations underpin the development of our new column-oriented file format and library: 1) existing file formats lack support for new floating-point compression techniques, 2) current file format libraries generally require users to decompress entire chunks before performing any operations [55], and 3) irrelevant metadata and complex control logic hinder the evaluation of compression performance. Our EBI file format and library aim to provide a simple yet complete file format for evaluating floating-point compression techniques. Additionally, the library is designed to be flexible enough to support in-situ query executions while offering high-level APIs.

4.1 EBI Library API

The EBI library provides a low-level API for manipulating chunks and a high-level API for interacting with the entire file. The low-level API is responsible for writing and reading individual chunks of encoded data. Since the chunk format depends on the chosen compression method, the low-level API maintains a consistent interface, while each compression method has a unique implementation. For compressing floating-point values, EBI provides a `Compressor` interface, with each evaluated compression method having its own `Compressor` implementation. Similarly, for decompression, EBI provides a `Reader` interface, where each method has a corresponding implementation that allows access to the data chunk in either compressed bytes or decompressed values. Certain

methods, such as `BuffReader` and `SprintzReader`, have specialized implementations because they support in-situ query execution directly on quantized values. Additionally, `BuffReader` is designed to efficiently interact with the subcolumns, ensuring proper query execution as described in section 3.2.2.

The EBI low-level API operates independently of the EBI high-level API and any specific EBI file format, making it suitable for development efforts that require the implementation of pure compression, decompression, and query execution.

The high-level API is responsible for handling the header, chunks, and the footer, managing `Write` and `Read` interface in Rust. Before performing actual compression or decompression, the high-level API interacts with the low-level API to convert an uncompressed buffer into a compressed buffer or vice versa, and writes the result at the appropriate position based on the format specification. The end-user interface of High-level API consists of `Encoder` and `Decoder`.

4.2 Evaluation Operators on EBI

Our evaluation framework leverages both the low-level and high-level APIs to perform unified evaluations and optimize in-situ queries when possible.

The framework is implemented using the `QueryExecutor` interface, with tailored implementations for each compression method. While the `QueryExecutor` interface provides a default method built on top of the `Reader` interface, it can be replaced with a specialized implementation for a specific compression method using the EBI low-level API. Note bm_filter is used for the bitmap filter. `QueryExecutor` has 6 methods:

(1) `filter(bitmap_filter, predicate): bitmap`
(2) `filter_materialize(bm_filter, predicate): float[]`
(3) `sum(bitmap_filter): float`
(4) `max(bitmap_filter): float`
(5) `min(bitmap_filter): float`
(6) `l2_norm(offset, target_vector): float`

With the lower-level query interface, the query logic is embedded within the chunk. The evaluation framework utilizes the `Decoder` to provide a high-level query interface. EBI provides the same high-level interface as its low-level counterpart, while also introducing additional functionalities.

(1) `materialize(bitmap_filter): float[]`
(2) `knn(target_vector, k): index[]`
(3) `matmul_cuda(target_matrix, shape): matrix`

Aggregations such as `sum`, which are available in the low-level interface, are executed on a per-chunk basis in the high-level interface. The materialize operation merges the decompressed results of each chunk. The `knn` function performs k-nearest neighbor search, which is implemented using the `l2_norm` primitive available in the low-level query API. The `matmul_cuda` function performs matrix multiplication by directly interfacing with the low-level API and leveraging cuBLAS for accelerated computation. Notably, the EBI design implicitly supports parallelization at the chunk level; however all evaluations in this work are conducted using a single thread.

5 BENCHMARK SETUP

The objective of this paper is to present a comprehensive benchmark of floating-point compression in data systems, with a focus on compression performance and efficiency, general database query performance, and performance in machine learning data.

Experiments are conducted on the two hardware setups. The GPU server is equipped with Intel(R) Xeon(R) Gold 6216 CPU, 192 GB RAM, 240 GB SSD, and Quadro RTX 6000. The other server is Intel(R) Xeon(R) Gold 6242 CPU, 192 GB RAM, 240 GB SSD.

All benchmark and compression techniques are natively implemented in Rust, with one exception. ALP [11] is the only compression method that uses a foreign function interface (FFI) between Rust and C++, as its implementation is highly optimized through the C++ compiler's auto-vectorization. We utilize autocxx [2], a wrapper library that automatically bridges the FFI between Rust and C++. In our case, the FFI overhead is minimal because C++ function calls are infrequent, being called only once per chunk containing 16,777,216 double values in our default setting. Additionally, the C++ function simply writes data to a buffer passed from Rust, keeping the interaction lightweight. Since both C++ and Rust are compiled to LLVM intermediate representation (IR) and subsequently optimized by the LLVM backend, their performance is directly comparable under equivalent compilation settings. We also believe that having one FFI-based implementation provides a flexible foundation for integrating future methods without requiring a full rewrite to Rust.

We use Rust 1.87 with the `--release` flag. ALP's C++ integration is via FFI and clang++ 18.0 is used internally. We utilized cuBLAS [1] with CUDA 12.6. We compiled the code for both AVX2 and AVX512 instruction sets by configuring the target CPU accordingly. Buff leverages hand-optimized functions tailored for AVX2, while ALP benefits from SIMD instructions auto-vectorized by LLVM.

We evaluated six operations on general floating-point datasets: compress, decompress, filter, filter-materialize, max, and sum. We also evaluated compression performance on several embedding datasets. To measure query workload performance, we executed two TPC-H queries, Q1 and Q6, with a focus on floating-point columns. We also used the UCR Time Series Classification Archive [26] to perform a 1-nearest neighbor (1-NN). A matrix multiplication operation was also evaluated on a GPU server using randomly generated floating-point data. For each configuration, we report the mean of five runs. Unless otherwise stated, all experiments were performed in memory with compiler optimizations targeting AVX2.

5.1 Evaluation Metrics

5.1.1 Compression Metrics. The compression metrics are used to evaluate a fundamental compression performance for each compression method. Compression ratio (CR) is a key metric for assessing the efficiency of a compression technique. A lower compression ratio indicates greater space savings. We use [de]compression throughput to quantify the speed at which a compression method compresses or decompresses a large set of values. Throughput represents the normalized elapsed time relative to the dataset size. A higher throughput indicates faster compression or decompression.

$$\text{CR.} = \frac{\text{Comp. Size}}{\text{Orig. Size}} \quad \text{Thrpt.} = \frac{\text{Orig. Size}}{\text{[De]Comp. Time}}$$

5.1.2 General Database Query Metrics. We use database operations filter, filter-materialize, max, and sum. Other operations, including min and average, can be derived from these primary operations, and are expected to have comparable computational costs; thus, they are not separately assessed. All queries are executed with EBI over compressed chunks. To evaluate late materialization [9], the filter takes a predicate as input and generates a RoaringBitmap [49], which is a space-efficient compressed bitmap. Filter-materialize first executes the filter operation and then materializes the records that satisfy the filter predicate. Max computes the maximum value across all the records, while sum aggregates the total sum of all record values. We adopt three filter predicates for evaluation: greater than the tenth percentile, greater than the ninetieth percentile, and equal to the median. The varying percentiles are intended to simulate different levels of selective queries. We conducted operator experiments under two orthogonal configurations: (1) SIMD target, comparing 256-bit AVX2 with 512-bit AVX512 and (2) I/O substrate, comparing on-disk execution with in-memory execution. For on-disk execution, the [un]compressed file is read from the filesystem after the cache is flushed prior to each operator evaluation.

We additionally include real-world queries derived from TPC-H Q1 and Q6, along with a simplified variant of Q1. The simplified Q1 query eliminates arithmetic expressions involving floating-point columns in order to evaluate in-situ query performance in a pratical setting. Each query is decomposed into two components: a non-floating-point part and a floating-point part. The non-floating-point component is executed using DuckDB [3], while the floating-point component is executed by EBI, which synthesizes the required computation from primitive operations.

For database queries, we evaluate only throughput. We measure both the end-to-end elapsed time and the breakdown of execution times. While the end-to-end execution time is directly measured by the benchmark, the segmented execution times are extracted through the instrumentation of our library.

To aggregate the elapsed time measurements across datasets, we compute query throughput as follows:

$$\text{Query Throughput [GB/s]} = \frac{\text{Original Size}}{\text{Query Elapsed Time}}$$

5.1.3 Machine Learning Query Metrics. Machine Learning (ML) tasks typically involve datasets with high precision and high entropy, which present significant challenges for compression. In this evaluation, we aim to assess the performance of state-of-the-art compression techniques on ML datasets. We examine the need for high precision for task performance by analyzing the impact of reduced precision on both compression efficiency and accuracy.

For evaluating compression on ML datasets, we used 2 embeddings generated using OpenAI's "text-embedding-3-small" [69]. For the ML query evaluation, we perform a 1-nearest neighbor (1-NN) classification task using the UCR Time Series Classification Archive [26]. 1-NN is a nearest-neighbor classification [25] that assigns an unlabeled data point the label of the closest previously classified data point. Given a set of train vectors with labels and an unlabeled target vector, where each vector represents a data point, the 1-NN classification proceeds as follows. First, it computes the distance between the target vector and each labeled training vector, then assigns the target vector the label of its nearest training vector.

The UCR Time Series Classification Archive contains 128 datasets from various domains, each containing train vectors and test vectors. Each test vector is intended to be classified into one of the labels assigned to train vectors. In this study, we store all train vectors in a single EBI file format. To evaluate performance, we measure the elapsed time for a single 1-NN operation using a single test vector. Since there are multiple test vectors, we obtain as many elapsed time measurements as there are test vectors. This elapsed time is then converted into throughput by dividing the original data size. In other words: `1-NN Throughput [GB/s] = (Original Size)/ (Elapsed Time for a single 1-NN Search)`

We also evaluate the performance of matrix multiplication operation on a GPU using CUDA and cuBLAS. In our experimental setup, the EBI format contains 40 matrices sized 4096 x 4096, each of which is multiplied with a target matrix of the same dimensions. This results in 40 matrix multiplications being performed against the target matrix.

5.2 Datasets

Table 3 presents all datasets evaluated in this paper classified with its type. For the general compression and database operator experiments, we use 13 time-series datasets and 17 non-time-series datasets spanning various domains. These datasets are sourced from the same data sources and domains as those used in the ALP paper [11], and include all datasets evaluated in the Chimp, Elf, and BtrBlock papers [48, 50, 52]. Due to high precision, Buff failed to compress on CMS/25, NYC/29, POI-lat, and POI-lon. Similarly, DeltaSprintz failed to compress on CMS/25, POI-lat, and POI-lon. In cases where a method fails to compress a dataset, we exclude that dataset from the evaluation for that method. We excluded these datasets whenever computing averages. For ML tasks, we primarily use the UCR Time Series Classification Archive [26]. To ensure the quantization is successfully done, datasets with precision higher than 9, we round each floating-point at the decimal point 10 to ensure their precision is at most 9. For the embedding datasets, we use AirBnB's property dataset [62] and arXiv [63], both embedded using OpenAI's "text-embedding-3-small".

Table 3: Datasets Evaluated. Top 2 rows are from ALP.

Type	Datasets
Time Series	Air-Pressure [64], Basel-temp [61], Basel-wind [61], Bird-migration [42], Bitcoin-price [46], City-Temp [91], Dew-Point-Temp [67], IR-bio-temp [66], PM10-dust [65], Stocks-DE [90], Stocks-UK [90], Stocks-USA [90], Wind-dir [68]
Non-Time Series	Arade/4 [5], Blockchain-tr [14], CMS/1 [5], CMS/25 [5], CMS/9 [5], Food-prices [95], Gov/10 [5], Gov/26 [5], Gov/30 [5], Gov/31 [5], Gov/40 [5], Medicare/1 [5], Medicare/9 [5], NYC/29 [5], POI-lat [40], POI-lon [40], SD-bench [94]
2D Time Series	*UCRArchive2018*'s 128 Datasets [26]
Embed.	Arxiv-Embed [63], Airbnb-Embed [62]
TPC-H	Generated with DuckDB v1.2.2 TPC-H Extension [6]

6 BENCHMARK RESULTS

We analyze benchmark results from four perspectives: compression performance, general database query performance, and machine learning query performance.

6.1 Compression Performance

We evaluate performance using three key metrics: compression ratio, compression throughput, and decompression throughput.

First, Figures 5 to 7 present the compression ratio, compression, and decompression throughput for all non-ML datasets. The boxplots in this paper use whiskers extending to 1.5 times the interquartile range (IQR). Dashed lines represent the mean values, while solid lines represent the median values. Figures 14 and 15 present the compression performance differences between time-series and non-time-series data. Decompression throughput is omitted due to its smaller difference and space constraints. These figures are results in-memory with an 256-bit AVX2 SIMD configuration.

These figures illustrate the trade-off between compression ratio and compression throughput. The top methods in terms of compression ratio are Gzip, Zstd, ALP, and Elf, in that order. ALP also achieves the highest compression and decompression throughput. While ALP provides the best balance between compression ratio and throughput, it is important to note that its high throughput is due to the use of efficient SIMD instructions. Although Zstd achieves relatively high compression and decompression throughput, Elf exhibits significantly lower throughput.

The XOR family compression methods exhibit relatively low throughput, including Elf, despite one of the best compression ratios. During decompression, these methods require access to the previous value to decode each record, which reduces throughput and eliminates random access.

While XOR family methods exhibit very low throughput, the byte-level general-purpose methods Snappy and Zstd achieve acceptable throughput. In terms of compression ratio, Snappy performs slightly worse, partly due to its algorithm's intentional limitation on the search window. However, particularly on non-time-series datasets, both Snappy and Zstd perform the second fastest. Buff, a quantization-based method, performs particularly well on time-series datasets while maintaining acceptable compression and decompression throughput.

Figures 11 and 12 show AVX2, on-disk throughput and Figure 13 show AVX512 in-memory throughput. On-disk decompression performance is influenced by read time, making compression ratio a critical factor. Particularly, Snappy is no longer faster than Zstd and Gzip due to poor compression ratio. On AVX512 experiments, only ALP increases throughput, attributed to its auto-vectorization-oriented design; however, the impact remains minimal due to the relatively high cost of I/O read operation.

From the perspective of compression performance, ALP is the best choice when SIMD instructions are enabled, as it achieves both high compression ratios and efficient compression and decompression throughput. Among other well-balanced methods, Zstd is a strong contender, offering a good balance between compression ratios and throughput, particularly for non-time-series data. For time-series data, Buff is a good choice, as it not only performs well in compression but also offers advantages in query execution, as discussed later. If compression ratio is the primary concern and throughput is less critical, Elf is a viable option, as it achieves one of the highest compression ratios.

Note that Gorilla was originally designed to store and analyze time-series data from system monitoring. As a result, this approach and its XOR derivatives are highly optimized for datasets that will result in many trailing zeros. This primarily occurs when repeating numbers are powers of two, including fractional data (i.e., 1/2, 1/4). To demonstrate how these methods work, we construct a synthetic dataset that uniformly generates runs up to 5 values long uniformly chosen a domain of $[2^4, 2^3, 2^2, 2^1, 1, 2^{-1}, 2^{-2}, \ldots, 2^{-6}]$. The synthesized dataset has 41.0 trailing zeros on average. Figure 18 shows the compression ratios on the synthetic dataset. Here, XOR-based methods and general-purpose compression methods, which exploit long repeated byte patterns, perform well, though they are not the best. While decimal values might be close to each other numerically, their actual binary representations can differ a lot, making such power-of-two-friendly datasets relatively uncommon in real-world scenarios. Notably, our non-ML datasets have 22.6 trailing zeros on average, much lower than this synthetic dataset.

6.2 Database Operator Performance

As this study focuses on floating-point compression for data systems, we evaluate query operator on compressed data. The operators assessed in this study include filter, max, and sum. For filter, we evaluate equality and greater than comparisons using each dataset's 10th, 50th, and 90th percentile values. Figures 8 to 10 show these results. Due to similar trends, results for the equality, 10th, and 50th percentiles are omitted; figures show throughput for max, sum, and selected filter queries. Figures 16 and 17 compare the query performance differences between time-series and non-time-series data. Filter throughput is omitted due to its consistent characteristic as Figures 16 and 17 and space constraints.

Buff features a specialized implementation for in-situ operator executions, allowing it to compute results without fully decompressing the data. Similarly, Sprintz can execute queries without requiring dequantization; however, it still needs to decode delta and zigzag encodings. Figures 8 and 9 show that Buff achieves significantly high throughput for filter greater than and max queries. This performance is particularly notable on time-series datasets, reflecting the same characteristics observed in its compression performance. Buff benefits from 256-bit AVX2 SIMD instructions and can efficiently skip unnecessary data due to its specialized format. For the sum operator, ALP achieves the highest throughput, followed by Buff. In contrast, XOR family methods also struggle with the sum operator due to their slow decompression.

With the exception of Buff, query throughput is generally proportional to decompression performances. For instance, decompression and sum throughput are highly correlated. ALP maintains high throughput, despite needing to fully decompress the data before computing the filter result. Sprintz achieves slightly better query performance, given its moderate decompression throughput. This advantage arises because Sprintz can perform filter comparisons directly on quantized values, reducing the overhead.

Figure 20 shows the execution time of the TPC-H Q1 query on the EBI side. While performance generally correlates with decompression throughput, Buff's performance varies depending on the query. As TPC-H Q6 exhibited similar relative performance, its results are omitted. The dotted line in Figure 20 represents a simplified version of Q1, which involves aggregation on a single column

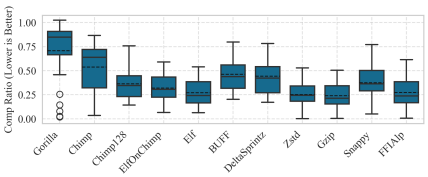

Figure 5: Compression Ratio

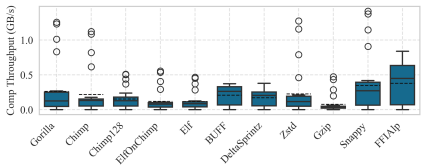

Figure 6: Compression Throughput

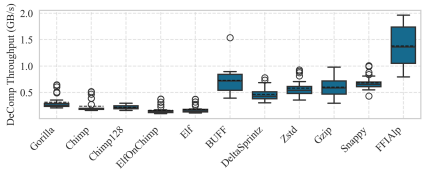

Figure 7: Decompression Throughput

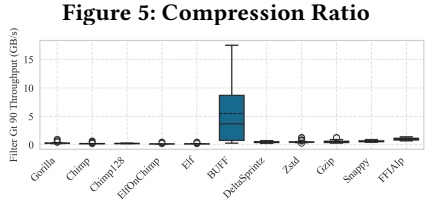

Figure 8: Filter > 90th %ile Throughput

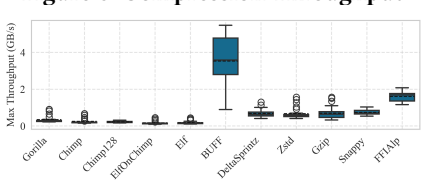

Figure 9: Max Operator Throughput

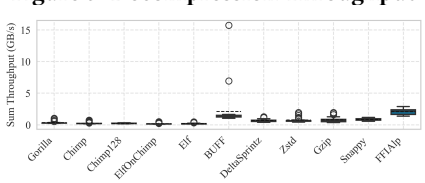

Figure 10: Sum Operator Throughput

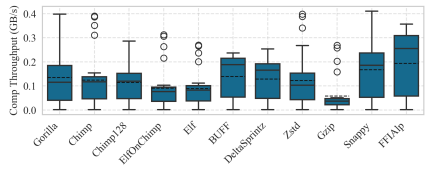

Figure 11: On Disk: Comp. Throughput

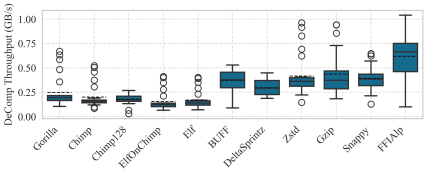

Figure 12: On Disk: Decomp. Throughput

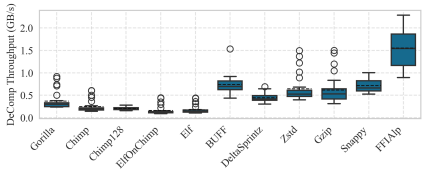

Figure 13: AVX512: Decomp. Throughput

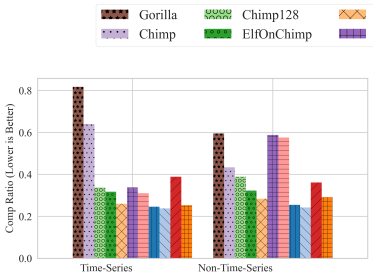

Figure 14: Compression Ratio

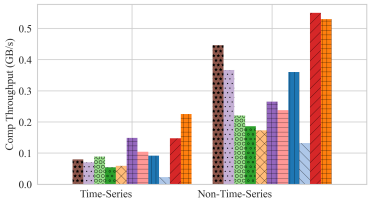

Figure 15: Compression Throughput

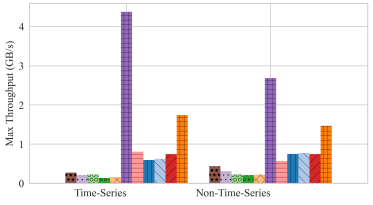

Figure 16: Max Operator Throughput

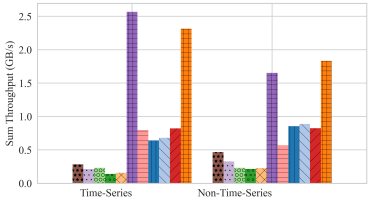

Figure 17: Sum Operator Throughput

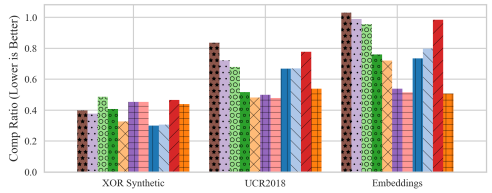

Figure 18: Mean Comp. Ratio on ML Datasets.

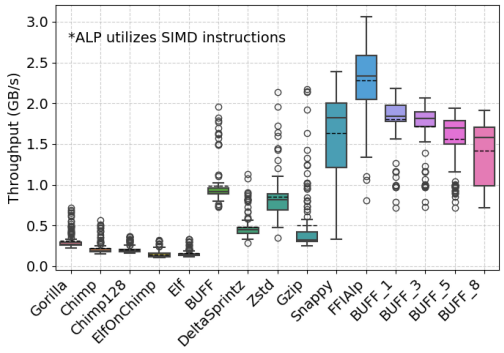

Figure 19: 1-NN with one target vector

without projection transformations. This enables Buff to perform in-situ query execution, making it the fastest in this case.

To summarize compression and database query performance results, Buff's optimizations for query operator led to a significant performance improvement, including in-situ executions of real-world TPC-H queries. In contrast, quantization-based and general-purpose methods generally outperform XOR methods, primarily due to the computational overhead of XOR and bitwise operations, which we further examine in the following section. For general compression performance, ALP achieves significantly superior results,

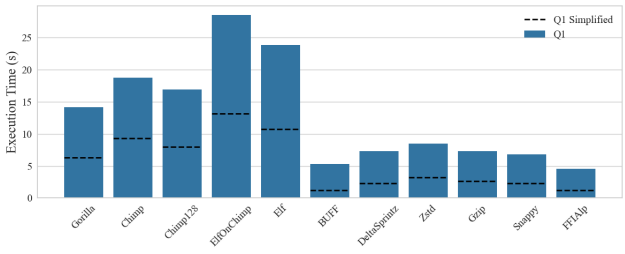

Figure 20: TPC-H Q1 Execution Times with Q1 Simplified Results with Horizontal Dotted Lines.

largely due to its effective use of SIMD instructions and a SIMD-optimized layout. Additionally, lightweight general-purpose compression methods such as Snappy and Zstd perform well, ranking just behind ALP in both compression and decompression throughput. Notably, Among non-time-series datasets, Zstd achieves the best compression ratio, second only to Gzip.

6.3 Profiling Methods

To analyze performance differences under our unified framework, we instrument EBI to record time spent on key tasks. Since not all algorithms align perfectly with their designated steps, or because we sometimes rely on external components or system calls, any other computational work is categorized under "Other Processing".

Figures 21 to 23 show the compression and decompression execution with times segmented by operation. The execution times are first normalized by dividing by the dataset size, and then the average is taken for each segment, presenting a runtime cost per bit unit. The I/O time across methods is consistent, except for decompression where the more effective methods read less data. XOR family methods use most of their time to calculate xors and perform bit-packing, which is a significant factor in their poor performance. These results also highlight that recent XOR methods improve their compression ratio at the expense of more costly bitwise operations. For a filter operation, the XOR family methods still suffer from most of their time on xor and bit operations. The reduced I/O and operations from Buff are in large part due to its ability to skip significant parts of the data. Except for Buff and Sprintz, all other compression methods exhibit the same time span for performing the compare operation. This consistency explains why the query performance is predominantly constrained by the decompression process.

6.4 Machine Learning Tasks Performance

Considering that nearest neighbor search is a fundamental building block for machine learning tasks [70], including clustering [12, 37, 75, 76, 82, 83, 85], classification [30, 74], similarity search [29, 31, 73, 78, 80, 81, 84, 96], and anomaly detection [15–22, 57–59, 71, 72, 77, 93], we explore a 1-NN task as a representative vector operation. We use the text embedding dataset for compression performance and the UCR Time Series Classification Archive for 1-NN. Our evaluation framework includes support for CUDA-optimized matrix multiplication operators. Given the foundational role of matrix multiplication in machine learning, we consider operations such as vector search for filtering and comparison, floating-point data compression (e.g., normalization of feature sets or embeddings), and matrix multiplication itself to be representative of typical database operations in machine learning workloads.

6.4.1 Compression Performance. In the UCR2018 dataset, the decompression throughput exhibited a trend similar to that observed in non-ML datasets. Additionally, Buff's lower-precision materialization improved throughput. This improvement was not due to an increase in the peak throughput but rather a rise in the minimum throughput, which led to a more consistent performance overall.

Figure 18 shows the compression ratio on the embedding dataset, which is increasingly popular for RAG tasks. Embedding data typically features high precision and high entropy, presenting significant challenges for compression. These compression ratios are inferior compared to those for general data. In particular, XOR-based methods struggle with its high entropy. Because the embedding has a precision of 10, the quantization-based family still compress the embedding data well.

For the compression ratio, there are no significant winners. Buff, DeltaSprintz, and ALP all have acceptable compression ratios of around 0.5 as the median value. For the compression throughput, Snappy is the fastest, but it struggles to achieve good compression due to its algorithmic design. As ALP remains the fastest, we confirmed Buff's lower-precision materialization improves its decompression performance.

6.4.2 1-Nearest Neighbor Performance on the UCR Archive. Figure 19 shows the throughput for classifying a single target vector. These results are similar to Figure 7 since 1-NN decompresses prior to the calculation. The reason why the throughput for 1-NN is higher than that of the decompression is while the decompression needs all the records to be stored in the vector, 1-NN needs only the current chunk to be stored as a decompressed form. Note that while the reduced precision of Buff improves performance, task accuracy is minimally impacted (less than 1% accuracy loss on average) – which aligns with research on low precision learning [99]. Overall, in 1-NN, since the decompression speed is a bottleneck, the decompression throughput is the most important factor for the 1-NN throughput. Supporting lower precision operations can significantly improve performance here.

6.4.3 Matrix Multiplication Performance on GPU. We conducted runtime profiling for this operator, as shown in Figure 24. The observed performance is proportional to decompression throughput. While the slower XOR-based methods suffer from expensive XOR operations, matrix multiplication operations account for a substantial portion of execution time in the other methods. These results highlight how the overall performance is influenced by both decompression and the subsequent operation. In general, when the subsequent operation is more computationally intensive than matrix multiplication, it dominates the runtime, resulting in similar overall throughput across methods.

7 CONCLUSION AND SUMMARY

In this study, we propose a comprehensive benchmark to evaluate floating-point compression methods with an 'apples-to-apples' comparison. Given the rising interest in systems support for floating-point data and compression, our open-source benchmark provides

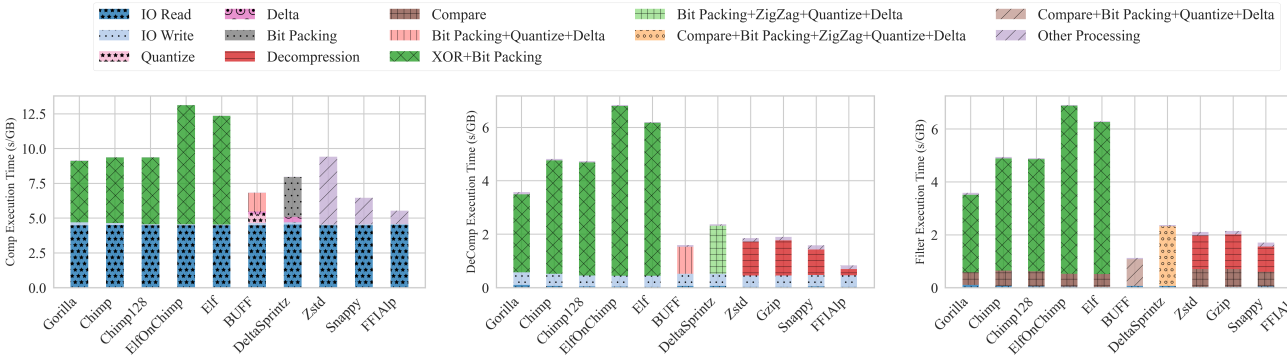

Figure 21: Compression Runtime Profiling, except Gzip which was too slow.

Figure 22: Decompression Runtime Profiling

Figure 23: Filter Greater than 90th Percentile Runtime Profiling

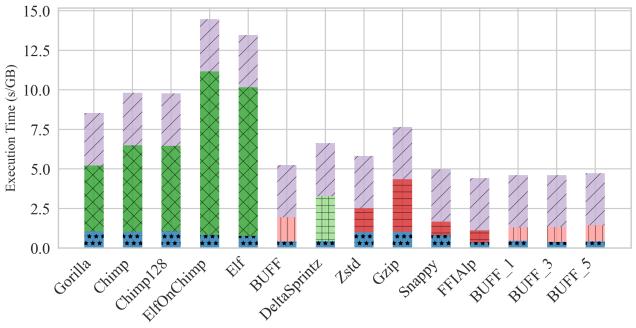

Figure 24: GPU Matmul Profiling

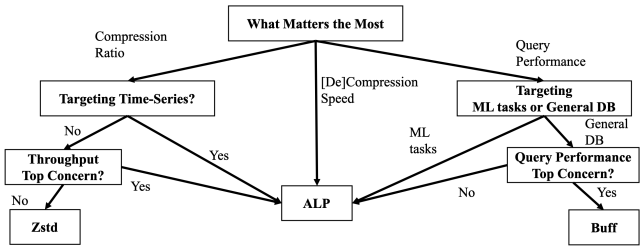

Figure 25: Decision Tree for Existing Compression Methods

a critical foundation for understanding trade-offs and enabling consistent comparisons across future work. Our results show that no single method dominates across all scenarios, as summarized in Figure 25 with a decision tree. We highlight the following findings.

XOR compression offers strong ratios but low throughput. While XOR-based methods such as Elf achieve top compression ratios, they suffer from slow decompression and lack in-situ query support. Recent gains in space efficiency often come at the cost of speed. Optimizing bit-level operations and integrating SIMD could improve performance in this family.

Decompression remains the main query bottleneck. Most methods (except Buff and Sprintz) require full decompression before query execution, making their performance proportional to decompression speed. Buff stands out with in-situ support and data skipping, offering significantly faster query execution. Extending such capabilities to other methods is a promising direction.

General-purpose methods perform well on non-time-series data. For non-time-series datasets, general-purpose compressors like Zstd and ALP show strong results. Zstd balances high compression ratio and throughput, while ALP leads in performance. These findings suggest general-purpose methods are viable when floating-point specificity is not critical.

SIMD and layout drive performance gains. ALP and Buff benefit from layout optimizations and SIMD instructions, enabling high [de]compression throughput. Future designs should explore combining layout-aware techniques with SIMD to boost both efficiency and flexibility, especially for XOR-style encoding.

Compression support for ML workloads remains limited. ML workloads are influenced by decompression speed and compute intensity, with embeddings often being high-entropy and hard to compress. Current methods lack efficient support for lossless ML data compression. Promising directions include ML-specific methods and tightly integrated in-situ decompression-computation models for lightweight inference.

Heterogeneous deployment scenarios demand broader evaluation. Applications span dashboard queries, ML tasks, and analytics, across cloud, edge, and hybrid environments. Hardware diversity—ranging from SIMD-less CPUs to GPU-accelerated systems—adds further complexity. Evaluating methods across these dimensions would require major codebase refactoring and infrastructure changes, which are beyond the scope of this paper. We leave such comprehensive exploration for future work, but believe that the results here demonstrate where trade-offs exist.

ACKNOWLEDGMENTS

This work was supported by the DARPA ASKEM program (award HR00112220042), the ARPA-H Biomedical Data Fabric project, grants from Liberty Mutual, Google, and Cisco Systems, and a Google DANI Award. Results presented in this paper were obtained using the Chameleon testbed [45] supported by the NSF.

REFERENCES

[1] The api reference guide for cublas, the cuda basic linear algebra subroutine library. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2025-07-10.

[2] autocxx — automatic safe interop between rust and c++. https://google.github.io/autocxx/. Accessed: 2025-07-10.

[3] Duckdb – an in-process sql olap database management system. https://duckdb.org/. Accessed: 2025-07-10.

[4] Postgresql: Documentation: 17: 8.1 : Numeric types. https://www.postgresql.org/docs/current/datatype-numeric.html. Accessed: 2025-07-10.

[5] Public bi benchmark. https://github.com/cwida/public_bi_benchmark. Accessed: 2025-02-08.

[6] Tpc-h extension – duckdb. https://duckdb.org/docs/stable/core_extensions/tpch.html. Accessed: 2025-07-10.

[7] [compression] patas compression (float/double) (variation on chimp). https://github.com/duckdb/duckdb/pull/5044, 2022. Accessed: 2025-07-10.

[8] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006.

[9] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[10] A. Afroozeh and P. Boncz. The fastlanes compression layout: Decoding > 100 billion integers per second with scalar code. *Proc. VLDB Endow.*, 16(9):2132–2144, May 2023.

[11] A. Afroozeh, L. X. Kuffo, and P. Boncz. Alp: Adaptive lossless floating-point compression. *Proc. ACM Manag. Data*, 1(4), Dec. 2023.

[12] M. Bariya, A. von Meier, J. Paparrizos, and M. J. Franklin. k-shapestream: Probabilistic streaming clustering for electric grid events. In *2021 IEEE Madrid PowerTech*, pages 1–6. IEEE, 2021.

[13] D. Blalock, S. Madden, and J. 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.

[14] Blockchair. Blockchair bitcoin transactions database dumps, 2025. Accessed: 2025-02-08. Downloaded blockchair_bitcoin_transactions_20250201.tsv.gz.

[15] P. Boniol, A. K. Krishna, M. Bruel, Q. Liu, M. Huang, T. Palpanas, R. S. Tsay, A. Elmore, M. J. Franklin, and J. Paparrizos. Vus: effective and efficient accuracy measures for time-series anomaly detection. *The VLDB Journal*, 34(3):32, 2025.

[16] P. Boniol, Q. Liu, M. Huang, T. Palpanas, and J. Paparrizos. Dive into time-series anomaly detection: A decade review. *arXiv preprint arXiv:2412.20512*, 2024.

[17] P. Boniol, J. Paparrizos, Y. Kang, T. Palpanas, R. S. Tsay, A. J. Elmore, and M. J. Franklin. Theseus: navigating the labyrinth of time-series anomaly detection. *Proceedings of the VLDB Endowment*, 15(12):3702–3705, 2022.

[18] P. Boniol, J. Paparrizos, and T. Palpanas. New trends in time series anomaly detection. In *EDBT*, pages 847–850, 2023.

[19] P. Boniol, J. Paparrizos, and T. Palpanas. An interactive dive into time-series anomaly detection. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5382–5386. IEEE, 2024.

[20] P. Boniol, J. Paparrizos, T. Palpanas, and M. J. Franklin. Sand in action: subsequence anomaly detection for streams. *Proceedings of the VLDB Endowment*, 14(12):2867–2870, 2021.

[21] P. Boniol, J. Paparrizos, T. Palpanas, and M. J. Franklin. Sand: streaming subsequence anomaly detection, 2021.

[22] P. Boniol, E. Sylligardos, J. Paparrizos, P. Trahanias, and T. Palpanas. Adecimo: Model selection for time series anomaly detection. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5441–5444. IEEE, 2024.

[23] X. Chen, J. Tian, I. Beaver, C. Freeman, Y. Yan, J. Wang, and D. Tao. Fcbench: Cross-domain benchmarking of lossless compression for floating-point data. *Proc. VLDB Endow.*, 17(6):1418–1431, May 2024.

[24] Y. Collet and M. Kucherawy. Zstandard Compression and the application/zstd Media Type. RFC 8478, Oct. 2018.

[25] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[26] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, Yanping, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML. The ucr time series classification archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/, October 2018. Accessed: 2025-07-10.

[27] L. P. Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[28] P. Deutsch et al. Gzip file format specification version 4.3. Technical report, RFC 1952, May, 1996.

[29] J. E. d'Hondt, H. Li, F. Yang, O. Papapetrou, and J. Paparrizos. A structured study of multivariate time-series distance measures. *Proceedings of the ACM on Management of Data*, 3(3):1–29, 2025.

[30] A. Dziedzic, J. Paparrizos, S. Krishnan, A. Elmore, and M. Franklin. Band-limited training and inference for convolutional neural networks. In *International Conference on Machine Learning*, pages 1745–1754. PMLR, 2019.

[31] J. E. d'Hondt, O. Papapetrou, and J. Paparrizos. Beyond the dimensions: A structured evaluation of multivariate time series distance measures. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*, pages 107–112. IEEE, 2024.

[32] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015.

[33] A. S. Foundation. Apache arrow. https://arrow.apache.org/. Accessed: 2025-07-10.

[34] A. S. Foundation. Apache orc. https://orc.apache.org/. Accessed: 2025-07-10.

[35] A. S. Foundation. Apache parquet. https://parquet.apache.org/. Accessed: 2025-07-10.

[36] J.-l. Gailly and M. Adler. Zlib compression library. 2004.

[37] A. Giannoulidis, A. Gounaris, and J. Paparrizos. Burst: Rendering clustering techniques suitable for evolving streams. *Proceedings of the VLDB Endowment*, 18(11):4054–4063, 2025.

[38] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991.

[39] Google. Snappy, a fast compressor/decompressor. — GitHub. https://github.com/google/snappy. Accessed: 2025-07-10.

[40] E. Hallmark. Points of interest poi database, 2019. Accessed: 2025-02-08.

[41] D. Hough. Applications of the proposed ieee 754 standard for floating-point arithetic. *Computer*, (3):70–74, 1981.

[42] InfluxData. Influxdb 2.0 sample data, 2025. Accessed: 2025-02-08.

[43] H. Jiang, C. Liu, Q. Jin, J. Paparrizos, and A. J. Elmore. Pids: attribute decomposition for improved compression and query performance in columnar storage. *Proceedings of the VLDB Endowment*, 13(6):925–938, 2020.

[44] H. Jiang, C. Liu, J. Paparrizos, A. A. Chien, J. Ma, and A. J. Elmore. Good to the last bit: Data-driven encoding with codecdb. In *Proceedings of the 2021 international conference on management of data*, pages 843–856, 2021.

[45] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[46] P. Kottarathil. Bitcoin historical dataset. https://www.kaggle.com/prasoonkottarathil/btcinusd, 2020. Accessed: 2025-07-10.

[47] S. Krishnan, A. J. Elmore, M. Franklin, J. Paparrizos, Z. Shang, A. Dziedzic, and R. Liu. Artificial intelligence in resource-constrained and shared environments. *ACM SIGOPS Operating Systems Review*, 53(1):1–6, 2019.

[48] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. Btrblocks: Efficient columnar compression for data lakes. *Proc. ACM Manag. Data*, 1(2), June 2023.

[49] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, Nov. 2016.

[50] R. Li, Z. Li, Y. Wu, C. Chen, and Y. Zheng. Elf: Erasing-based lossless floating-point compression. *Proc. VLDB Endow.*, 16(7):1763–1776, Mar. 2023.

[51] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.

[52] P. Liakos, K. Papakonstantinopoulou, and Y. Kotidis. Chimp: efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.*, 15(11):3058–3070, July 2022.

[53] C. Liu, H. Jiang, J. Paparrizos, and A. J. Elmore. Decomposed bounded floats for fast compression and queries. *Proc. VLDB Endow.*, 14(11):2586–2598, July 2021.

[54] C. Liu, J. Paparrizos, and A. J. Elmore. Adaedge: A dynamic compression selection framework for resource constrained devices. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1506–1519. IEEE, 2024.

[55] C. Liu, A. Pavlenko, M. Interlandi, and B. Haynes. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment*, 16(11):3044–3056, 2023.

[56] C. Liu, A. Pavlenko, M. Interlandi, and B. Haynes. Data formats in analytical dbmss: performance trade-offs and future directions. *The VLDB Journal*, 34(3):30, 2025.

[57] Q. Liu, P. Boniol, T. Palpanas, and J. Paparrizos. Time-series anomaly detection: Overview and new trends. *Proceedings of the VLDB Endowment (PVLDB)*, 17(12):4229–4232, 2024.

[58] Q. Liu, S. Lee, and J. Paparrizos. Tsb-autoad: Towards automated solutions for time-series anomaly detection. *PVLDB*, 18(11):4364–4379, 2025.

[59] Q. Liu and J. Paparrizos. The elephant in the room: Towards a reliable time-series anomaly detection benchmark. In *The Thirty-eight Conference on Neural Information Processing Systems*, 2024.

[60] S. Liu, T. Mangla, T. Shaowang, J. Zhao, J. Paparrizos, S. Krishnan, and N. Feamster. Amir: Active multimodal interaction recognition from video and network traffic in connected environments. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 7(1):1–26, 2023.

[61] meteoblue. Historical weather data from meteoblue, 2025. Accessed: 2025-02-08. Data retrieved for the period 2024-01-01 to 2025-02-08.

[62] MongoDB. Airbnb embeddings. https://huggingface.co/datasets/MongoDB/airbnb_embeddings, 2024. Accessed: 2025-07-10.

[63] MongoDB. Subset arxiv papers with embeddings. https://huggingface.co/datasets/MongoDB/subset_arxiv_papers_with_embeddings, 2024. Accessed: 2025-07-10.

[64] National Ecological Observatory Network (NEON). Barometric pressure (dp1.00004.001), 2021.

[65] National Ecological Observatory Network (NEON). Dust and particulate size distribution (dp1.00017.001), 2021.

[66] National Ecological Observatory Network (NEON). Ir biological temperature (dp1.00005.001), 2021.

[67] National Ecological Observatory Network (NEON). Relative humidity above water on-buoy (dp1.20271.001), 2021.

[68] National Ecological Observatory Network (NEON). 2d wind speed and direction (dp1.00001.001), 2025.

[69] OpenAI. Embeddings - openai api. https://platform.openai.com/docs/guides/embeddings. Accessed: 2025-07-10.

[70] I. Paparrizos. *Fast, scalable, and accurate algorithms for time-series analysis*. PhD thesis, Columbia University, 2018.

[71] J. Paparrizos, P. Boniol, Q. Liu, and T. Palpanas. Advances in time-series anomaly detection: Algorithms, benchmarks, and evaluation measures. In *SIGKDD*, 2025.

[72] J. Paparrizos, P. Boniol, T. Palpanas, R. S. Tsay, A. Elmore, and M. J. Franklin. Volume under the surface: a new accuracy evaluation measure for time-series anomaly detection. *Proceedings of the VLDB Endowment*, 15(11):2774–2787, 2022.

[73] J. Paparrizos, I. Edian, C. Liu, A. J. Elmore, and M. J. Franklin. Fast adaptive similarity search through variance-aware quantization. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2969–2983. IEEE, 2022.

[74] J. Paparrizos and M. J. Franklin. Grail: efficient time-series representation learning. *Proceedings of the VLDB Endowment*, 12(11):1762–1777, 2019.

[75] J. Paparrizos and L. Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1855–1870, 2015.

[76] J. Paparrizos and L. Gravano. Fast and accurate time-series clustering. *ACM Transactions on Database Systems (TODS)*, 42(2):1–49, 2017.

[77] J. Paparrizos, Y. Kang, P. Boniol, R. S. Tsay, T. Palpanas, and M. J. Franklin. Tsb-uad: an end-to-end benchmark suite for univariate time-series anomaly detection. *Proceedings of the VLDB Endowment*, 15(8):1697–1711, 2022.

[78] J. Paparrizos, H. Li, F. Yang, K. Wu, J. E. d'Hondt, and O. Papapetrou. A survey on time-series distance measures. *arXiv preprint arXiv:2412.20574*, 2024.

[79] J. Paparrizos, C. Liu, B. Barbarioli, J. Hwang, I. Edian, A. J. Elmore, M. J. Franklin, and S. Krishnan. Vergedb: A database for iot analytics on edge devices. In *CIDR*, 2021.

[80] J. Paparrizos, C. Liu, A. J. Elmore, and M. 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.

[81] J. Paparrizos, C. Liu, A. J. Elmore, and M. J. Franklin. Querying time-series data: A comprehensive comparison of distance measures. *Data Engineering*, page 69, 2023.

[82] J. Paparrizos and S. P. T. Reddy. Odyssey: An engine enabling the time-series clustering journey. *Proceedings of the VLDB Endowment*, 16(12):4066–4069, 2023.

[83] J. Paparrizos and S. P. T. Reddy. Time-series clustering: A comprehensive study of data mining, machine learning, and deep learning methods. *Proceedings of the VLDB Endowment*, 18(11):4380–4395, 2025.

[84] J. Paparrizos, K. Wu, A. Elmore, C. Faloutsos, and M. J. Franklin. Accelerating similarity search for elastic measures: A study and new generalization of lower bounding distances. *Proceedings of the VLDB Endowment*, 16(8):2019–2032, 2023.

[85] J. Paparrizos, F. Yang, and H. Li. Bridging the gap: A decade review of time-series clustering methods. *arXiv preprint arXiv:2412.20582*, 2024.

[86] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[87] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006.

[88] D. R.-J. G.-J. Rydning. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 2018.

[89] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1390–1403, New York, NY, USA, 2022. Association for Computing Machinery.

[90] spring. Financial data set used in infore project, June 2020.

[91] SRK. Daily temperature of major cities, 2019. Accessed: 2025-02-08. Data sourced from the University of Dayton.

[92] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[93] E. Sylligardos, P. Boniol, J. Paparrizos, P. Trahanias, and T. Palpanas. Choose wisely: An extensive evaluation of model selection for anomaly detection in time series. *Proceedings of the VLDB Endowment*, 16(11):3418–3432, 2023.

[94] I. Tesla. Ssd and hdd benchmarks dataset, 2022. Accessed: 2025-02-08.

[95] W. F. P. (WFP). Global food prices database (wfp), 2021. Accessed: 2025-02-22.

[96] F. Yang and J. Paparrizos. Spartan: Data-adaptive symbolic time-series approximation. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025.

[97] Z. Yang and S. Chen. Most: Model-based compression with outlier storage for time series data. *Proc. ACM Manag. Data*, 1(4), Dec. 2023.

[98] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du. Compressdb: Enabling efficient compressed data direct processing for various databases. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1655–1669, 2022.

[99] H. Zhang, J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International Conference on Machine Learning*, pages 4035–4043. PMLR, 2017.