Decomposed Bounded Floats for Fast Compression and Queries

Chunwei Liu, Hao Jiang, John Paparrizos, Aaron J. Elmore
University of Chicago
{chunwei,hajiang,jopa,aelmore}@cs.uchicago.edu

ABSTRACT

Modern data-intensive applications often generate large amounts of low precision float data with a limited range of values. Despite the prevalence of such data, there is a lack of an effective solution to ingest, store, and analyze bounded, low-precision, numeric data. To address this gap, we propose Buff, a new compression technique that uses a decomposed columnar storage and encoding methods to provide effective compression, fast ingestion, and high-speed in-situ adaptive query operators with SIMD support.

PVLDB Reference Format:
Chunwei Liu, Hao Jiang, John Paparrizos, Aaron J. Elmore. Decomposed Bounded Floats for Fast Compression and Queries. PVLDB, 14(11): 2586 - 2598, 2021.
doi:10.14778/3476249.3476305

1 INTRODUCTION

Modern applications and systems are generating massive amounts of low precision floating-point data. This includes server monitoring, smart cities, smart farming, autonomous vehicles, and IoT devices. For example, consider a clinical thermometer that records values between 80.0 to 120.9, a GPS device between -180.0000 to 180.0000, or an index fund between 0.0001 to 9,999.9999. The International Data Corporation predicts that the global amount of data will reach 175ZB by 2025 [44], and sensors and automated processes will be a significant driver of this growth. To address this growth, data systems require new methods to efficiently store and query this low-precision floating-point data, especially as data growth is outpacing the growth of storage and computation [38].

Several popular formats exist for storing numeric data with varied precision. A *fixed-point* representation allows for a fixed number of bits to be allocated for the data to the right of the radix (i.e., decimal point) but is not commonly used due to the more popular floating-point representation. *Floats* (or floating-point) allows for a variable amount of precision by allocating bits before and after the radix point (hence the floating radix point), within a fixed total number of bytes (32 or 64). For floats, the IEEE float standard [24] is widely supported both by modern processors and programming languages, and is ubiquitous in today's applications.

However, two main reasons result in floats not being ideal for many modern applications: (i) an overly high-precision and broader range that wastes storage and query efficiency; and (ii) not being amenable for effective compression with efficient in-situ filtering operations. For the former reasons, many databases offer custom precision format, typically referred to as a *numeric* data type. For example, consider Figure 1 that shows how precision varies for the IEEE float standard and sample application requirements on GPS and temperature datasets. The numeric approach allows for "just enough" precision but is not well optimized for efficient storage and filtering. Several methods have been explored for the latter reason, but due to the standard format, compression opportunities are limited for compression effectiveness, throughput, and in-situ query execution. This paper proposes a new storage format that extends the ideas of a numeric data type that supports custom precision but is optimized for fast and effective encoding while allowing to work directly over the encoded data.

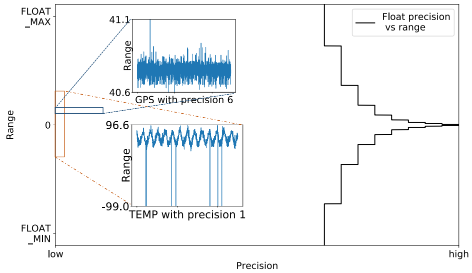


Figure 1: Many datasets only span a fixed range of values with limited precision, which is a small subset of broad float number range and precision.

To motivate our design, we first describe short-comings with popular compression techniques for bounded-precision floating data. Specifically, the recently proposed Gorilla method [40] is a delta-like compression approach that calculates the XOR for adjacent values and only saves the difference. Gorilla achieves compression benefits by replacing the leading and trailing zeros with counts. Gorilla is a state-of-the-art compression approach for floats, but it does not work well on low-precision datasets as low-precision does not impact float's representation similarity. In addition, Gorilla's encoding and decoding steps are slow because of its complex variable coding mechanism. Alternative float compression approaches leverage integer compression techniques by scaling the float point value into integer [9]. Despite their simplicity, these approaches rely on multiplication and division operations that are usually expensive and, importantly, cause overflowing problems when the input value and the quantization (i.e., scaling) factor are too large.

General-purpose byte-oriented compression approaches, such as Gzip [17], Snappy [52] and Zlib [21], can also be applied to compress floats. The input float values are serialized into binary representation before applying byte-oriented compression. These compression approaches are usually slow because of multiple scans looking for commonly repeated byte sub-sequences. Furthermore, full decompression is needed before any query evaluation. Dictionary encoding [8, 32, 46] is applicable for float data, but it is not

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476305

ideal since the cardinality of input float data is usually high, resulting in an expensive dictionary operation overhead. Note that we are only considering lossless formats or formats with bounded loss (e.g., configurable precision). Lossy methods (e.g., spectral decompositions [19] and representation learning [35]) may compress the data more aggressively but often at the cost of losing accuracy.

Decades of database systems demonstrate the benefit of a format that allows for a value domain that includes a configurable amount of precision. We take this approach, and to address the above concerns, integrate ideas from columnar systems and data compression for our proposed method, **B**o**U**nded **F**ast **F**loats compression (Buff). Buff provides fast ingestion from float-based inputs and a compressible decomposed columnar format for fast queries. Buff ingestion avoids expensive conditional logic and floating-point mathematics. Our storage format relies on a fixed-size representation for fast data skipping and random access, and incorporates encoding techniques, such as bit-packing, delta-encoding, and sparse formats to provide good compression ratios[1] and fast adaptive query operators. When defining an attribute that uses Buff, the user defines the precision and optionally the minimum and maximum values (e.g., 90.000-119.999). Without the defined min and max values, our approach infers these through the observed range, but at a decreased ingestion performance. Our experiments show the superiority of Buff over current state-of-the-art approaches. Compared to the state-of-the-art, Buff achieves up to 35× speedup for low selective filtering, up to 50× speedup for aggregations and high selective filtering, and 1.5× speedup for ingestion with a single thread, while offering comparable compression sizes to the state-of-the-art.

We start with a review of the relevant background (Section 2), which covers numeric representations (Section 2.1) and a wide range of compression methods (Section 2.2). In Section 3, we present Buff compression and query execution with four contributions:

- We introduce a *"just-enough"* bit representation for many real-world datasets based on the observation of their limited precision and range (Section 3.1).
- We apply an aligned decomposed *byte-oriented columnar storage* layout for floats to enable fast encoding with progressive, in-situ query execution on compressed data (Section 3.2).
- We propose *sparse encoding* to handle outliers during compression and query execution (Section 3.3).
- We devise an *adaptive filtering* to automatically choose between SIMD and scalar filtering (Section 3.4).

We evaluate Buff with its competitors in terms of compression ratio and query performance in Section 4, and conclude in Section 5.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce several numeric data representations, including the most popular float format. Then, we review several float compression approaches with a set of examples. Finally, we include popular query-friendly data structures that have influenced our compression format.

2.1 Numeric Data Representation

Numeric data representation is the internal representation of numeric values in any hardware or software of digital systems, such as programmable computers and calculators [48]. Many data representations are developed according to different system and application requirements. The most popular implementations include fixed-point (including numeric data type) and float-point.

Fixed-point uses a tuple $< sign, integer, fractional >$ to represent a real number R:

$$R = (-1)^{sign} * integer.fractional$$

Fixed-point partitions its bits into two fixed parts: signed/unsigned integer section and fractional section. For a given bits budget N, the fixed-point allocate a single bit for sign, I bits for the integer part, and F bits for the fractional part (where $N = 1 + I + F$). Fixed-point always has a specific number of bits for the integer part and fractional part. Figure 2 shows examples of fixed-point; as we can see from the "Fixed" column, the radix point's location is always fixed, no matter how large or small the corresponding real number is. Fixed-point arithmetic is just scaled integer arithmetic, so most hardware implementations treat fixed-point numbers as integer numbers with logically decimal point partition between integer and fractional part. Fixed-point usually has a limited range ($-2^I \sim 2^I$) and precision ($2^{(-F)}$), thus it can encounter overflow or underflow issues. A more advanced dynamic fixed-point representations allow moving the point to achieve the trade-off between range and precision. However, this is more complicated as it involves extra control bits indicating the location of the point. Fixed-point is rarely used outside of old or low-cost embedded microprocessors where a floating-point unit (FPU) is unavailable. Fixed-point is currently supported by few computer languages as floating-point representations are usually simpler to use and accurate enough.

Floating-point uses a tuple $< sign, exponent, mantissa >$ to represent a real number R:

$$R = (-1)^{sign} * mantissa * \beta^{exponent}$$

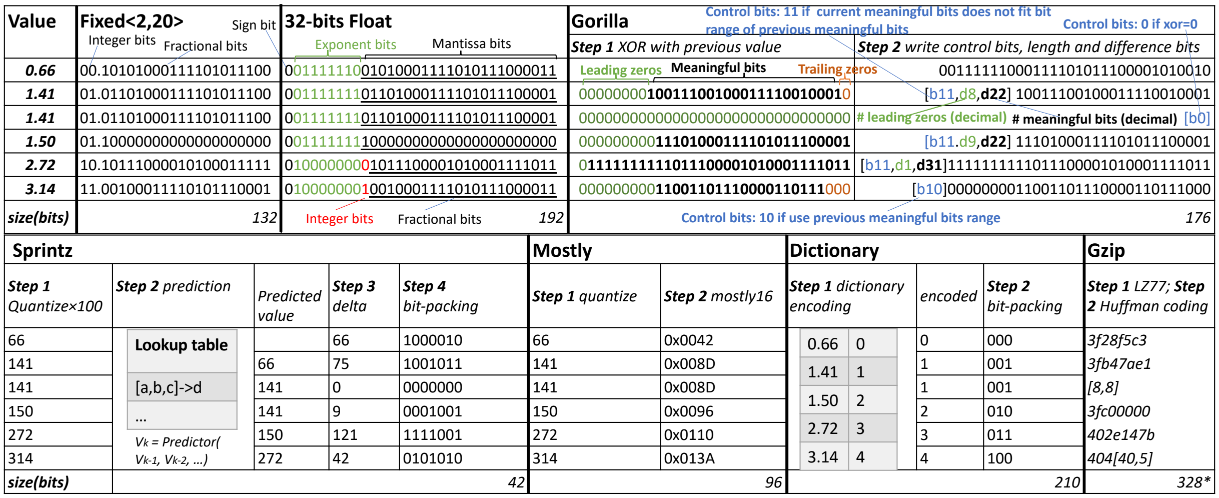
Instead of using a fixed number of bits for the integer part and fractional part, Float point reserves a certain number of bits for the exponent part and mantissa, respectively. The base β is an implicit constant given by the number representation system, while in our floating-point representation, β equals to 2. The number of significant digits does not depend on the position of the decimal point. For a given bit budget N, the floating-point allocates a single bit for sign, E bits for the exponent part, and M bits for the mantissa part (where $N = 1 + E + M$). The IEEE standard defines the format for single and double floating-point numbers using 32 and 64 bits, respectively. The exponent is encoded using an offset-binary representation with the zero offset by exponent bias:

$$R = (-1)^{sign} * 1.mantissa * 2^{exponent}$$

The IEEE standard specifies a 32-bits float as having: Sign bit: 1 bit, Exponent width: 8 bits (offset by 127), and Significand precision: 24 bits (23 explicitly stored). Figure 2 shows several examples of 32-bits float format; as we can see from the '32-bits float' column, the underlined bits correspond to the fractional bits, and the radix point is floating base on its represented value.

Numeric data type allocates enough bits for a given real number depending on the precision and scale of the input. Numeric data type can store numbers with a very large number of digits and perform calculations exactly. To declare a column of type numeric, we use the syntax: NUMERIC(precision, scale). The scale is the

[1] Compression ratio is defined as $\frac{\text{compressed\_size}}{\text{uncompressed\_size}}$.



Sign bit
Integer bits
Fractional bits
Exponent bits
Mantissa bits
Leading zeros
Meaningful bits
Trailing zeros
Control bits: 11 if current meaningful bits does not fit bit range of previous meaningful bits
Control bits: 0 if xor=0
# leading zeros (decimal)
# meaningful bits (decimal)
Control bits: 10 if use previous meaningful bits range

Value	Fixed<2,20>	32-bits Float	Gorilla *Step 1* XOR with previous value	 *Step 2* write control bits, length and difference bits
0.66	00.10101000111101011100	00111111001010001111010111000011		00111111000111101011100001010010
1.41	01.01101000111101011100	00111111101101000111101011100001	00000000**10011100100011110010001**0	[b11,d8,**d22**] 1001110010001111001000 1
1.41	01.01101000111101011100	00111111101101000111101011100001	00000000000000000000000000000000	[b0]
1.50	01.10000000000000000000	00111111110000000000000000000000	000000000**11101000111101011100001**	[b11.d9,**d22**] 11101000111101011100001
2.72	10.10111000010100011111	01000000001011100001010001111011	0**1111111111011100001010001111011**	[b11,d1,**d31**]1111111111011100001010001111011
3.14	11.00100011110101110001	01000000010010001111010111000011	000000000**11001101110000110111**000	[b10]00000000110011011100001101110000
size(bits)	*132*	Integer bits Fractional bits *192*		*176*

Sprintz *Step 1* Quantize×100	 *Step 2* prediction	 *Predicted value*	 *Step 3* delta	 *Step 4* bit-packing	Mostly *Step 1* quantize	 *Step 2* mostly16	Dictionary *Step 1* dictionary encoding		 *encoded*	 *Step 2* bit-packing	Gzip *Step 1* LZ77; *Step 2* Huffman coding
66	**Lookup table** [a,b,c]->d … *V$_k$ = Predictor(V$_{k-1}$, V$_{k-2}$, …)*		66	1000010	66	0x0042	0.66	0	0	000	*3f28f5c3*
141		66	75	1001011	141	0x008D	1.41	1	1	001	*3fb47ae1*
141		141	0	0000000	141	0x008D	1.50	2	1	001	*[8,8]*
150		141	9	0001001	150	0x0096	2.72	3	2	010	*3fc00000*
272		150	121	1111001	272	0x0110	3.14	4	3	011	*402e147b*
314		272	42	0101010	314	0x013A			4	100	*404[40,5]*
size(bits)	*42*				*96*		*210*				*328**

**For Gzip, we skip showing step 2 Huffman coding for space, and report the final encoded size.*

Figure 2: Compression examples for different approaches.

count of decimal digits in the fractional part, to the right of the decimal point. The precision is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point (e.g., 65.4321 has a precision of 6 and a scale of 4).

The Numeric data type is used in most database systems, e.g., MySQL, PostgreSQL, Redshift, and DB2, when exactness is required. However, the flexibility and high precision come at the cost of more control bits associated with each value. The storage cost of the Numeric type varies depending on the system implementation. PostgreSQL uses two bytes for each group of four decimal digits, plus 3 ∼ 8 bytes overhead for each value. In MySQL, Numeric values are represented in a binary format that packs nine decimal digits into four bytes. This makes the Numeric type inefficient for space. Besides, the query on Numeric data type is either by materializing to float point for approximate calculation or using its own exact arithmetic which is very expensive compared to the integer types or the floating-point types described earlier.

2.2 Compression for Decimal Numbers

Due to the popularity of float-point data, several compression techniques exist, including methods for float-point data, general-purpose methods, and migrated approaches from integer compression. We limit our discussion to lossless compression techniques.

Gorilla [40] is an in-memory time-series database developed by Facebook. It introduces two encodings to improve delta encoding: delta-of-delta (delta encoding on delta encoding, e.g., [101,102,103, 104] are encoded as [101,1,0,0]) for timestamps, which is usually an increasing integer sequence, and XOR-based encoding for value domain, which is a float type. In the XOR-based float encoding, successive float values are XORed together, and only the different bits (delta) are saved. The delta is then stored using control bits to indicate how many leading and trailing zeroes are in the XOR value. Similar compression techniques are also used in numerical simulation [25, 30, 31] and scientific computing [14, 43]. Figure 2 shows an example of Gorilla encoding with the first step calculating XOR, and the second step writing the control bits, number of leading zeros, number of meaningful bits, and actual meaningful bits. We use 32-bits to save space even though Gorilla was originally designed for 64-bits format. Gorilla is the state-of-the-art approach for float compression, which is widely used in many time series database systems, such as InfluxDB [3] and TimescaleDB [5].

However, Gorilla compression uses variable-length encoding, which means records are not aligned with their encoded bits. The encoded bits have to be decoded sequentially to reach the target records. Each value depends on its previous record; thus, for a target value, all previous values must be decompressed. Those features impede effective record skipping and random data access.

Sprintz [9] was initially designed for integer time series compression. Sprintz employs a forecast model to predict each value based on previous records via a lookup table, and then encodes the delta between the predicted value and the actual value. Those delta values are usually closer to zero than the actual value, making it smaller when encoded with bit-packed encoding. It is also possible to apply Sprintz to floats by first quantizing the float into an integer. As is shown in Figure 2, we first multiply input values by 100 and get a series of integers. Then we predict the next value based on the prediction model. In the third step, we calculate the delta between the prediction and actual value, and compress the delta with bit-packing encoding in the last step. Sprintz uses fixed-length for encoded values, which is easy to locate the target value. However, to decode the target value, we need to fully decode prior values, since previous values are still needed to predict the current value and then decode the value with compressed delta. Furthermore, the multiplication of decimal numbers is costly and potentially raises integer overflow issues during quantization.

Mostly encoding encodes the data column where its data type is more extensive than most of its stored values required. Mostly encoding is used for integer and numeric data type compression in the Amazon Redshift data warehouse [22]. For numeric values, Mostly encoding quantifies floats into an integer before applying compression. Most of the column values are encoded to smaller data representation with mostly encoding, while the remaining values that cannot be compressed are stored in their original form. Mostly encoding can be less effective than no compression when a high portion of the column's values cannot be compressed. Figure 2 shows an example of Mostly encoding. It first quantifies the float values and then applies MOSTLY16 (2-byte) for the given data as it needs 9 bits for each quantized value (we use hex to save space). This representation is less effective than bit-packing encoding that stores input value using as few bits as possible.

General-purpose byte-oriented compression encodes the input data stream at the byte level. Popular techniques, such as Gzip [17], Snappy [52] and Zlib [21] derive from the LZ77 family [54] that looks for repetitive sub-sequence within a sliding window on the input byte stream and encodes the recurring sub-sequence as a reference to its previous occurrence (first step in Figure 2). For a better compression ratio, Gzip applies Huffman encoding on the reference data stream (second step in Figure 2). Snappy only applies LZ77 but skips Huffman encoding for higher throughput. Byte-oriented compression treats the input values as a byte stream and encodes them sequentially. The data block needs to be fully decompressed before any original value can be accessed.

2.3 Query-friendly Storage Layout

Compression should not only keep the data size small but also support fast query execution. Prior work speeds up query performance by introducing a hierarchical data representation layout. Some works [29, 33, 42, 50] are orthogonal to data compression but are applicable to Buff to speed up the query performance.

DAQ [42] uses a bit-sliced index representation and computes the most significant bits of the result first then less significant ones. With bit-sliced index representation, it performs efficient approximation and provides efficient algorithms for evaluating predicates and aggregations for unsigned integers type. Column Sketch [23] introduces a new class of indexing scheme by applying lossy compression on a value-by-value basis, mapping original data to a representation of smaller fixed width codes. Queries are evaluated affirmatively or negatively for the vast majority of values using the compressed data, and check the original data for the remaining values only if needed. PIDS [27] identifies common patterns in string attributes from relational databases and uses the discovered pattern to split each attribute into sub-attributes and further compress it. The sub-attributes layout enables a predicate push-down to each attribute. And the intermediate result from the prior column could be projected to the following attributes efficiently with the aid of fast data skipping and progressive filtering.

BitWeaving [29] provides fast scans in a main-memory database by exploiting the parallelism available at the bit level in modern processors. It organizes the input codes horizontally or vertically into a processor word, allowing early pruning techniques to avoid accesses on unnecessary data at the bit level and speed up the scan performance. ByteSlice [20] is another main memory storage layout that supports both highly efficient scans and lookups. Similar to BitWeaving, ByteSlice assumes encoded binary as input. But instead of stripping the input code in bit units, ByteSlice decomposes the input code at the byte level and pads the trailing bits into a byte to achieve better read/write speed and be compatible with SIMD instructions. MLWeaving [50] is an in-memory data storage layout that allows efficient materialization of quantized data at any level of precision. Its memory layout is based on BitWeaving, where the first bit in a batch of input values is saved as a word, followed by the second bits of the same batch, and so on. Data at any level of precision can be retrieved by following a different access pattern over the same data structure. Using lower precision input data is of special interest, given their overall higher efficiency.

These techniques speed up query performance by progressively filtering out disqualified records and narrowing down the records that need to be parsed. They mainly process a query in their defined unit (either sub-attributes or bits) on their columnar data layout. This fine-grained columnar data layout is the foundation for Buff.

3 BUFF OVERVIEW

Many applications generate data that varies within a specific range with limited precision. Based on our discussion of state-of-the-art, none can leverage this feature to support efficient compression and query execution. To alleviate those deficiencies, we propose a novel compression method for floats. The compression works at two levels: eliminating the less significant bits based on a given precision (Section 3.1), and splitting float into the integer part and fractional part, then compressing those bits separately with two splitting strategies (Section 3.2). We then introduce sparse encoding to handle outliers to avoid compression performance deterioration (section 3.3). In the end, we describe query execution on our compressed data format (section 3.4). With these techniques, we achieve both good compression ratio and outstanding query performance.

3.1 Bounded Float

Computer systems can only operate on numeric values with finite precision and range. Using floating-point values as real numbers does not clearly identify the precision with which each value must be represented. Too small precision yields inaccurate results, and too big wastes computational and storage resources [18]. We target applications that need limited precision and data that falls within a finite range. Therefore, our proposed compression takes full advantage of the range and precision features of a given dataset.

3.1.1 *Bounded Precision.*

Float point data-type uses most bits for *mantissa*, which is not necessarily needed for many sensors or applications since many sensors usually provide limited decimal precision (e.g., the clinical thermometer has one place of precision 0.1 °F). However, the decimal precision is not aligned with those *mantissa* bits in a float due to the floating radix point in the format. Float format makes the most of its *mantissa* bits to get the best approximation for the given number. With 0.1 for example, all 52 *mantissa* bits are used to get its approximation (0.10000000000000000555) in IEEE double format. This mechanism is intended to support high precision for float format, but it impedes efficient float compression in most real-world cases. Those *mantissa* trailing bits provide far more extra precision than required, which leads to a huge bits change with a tiny number fluctuation. For example, Gorilla achieves good compression by first applying XOR to the current number with the previous one, then removing the leading and trailing zeros. The problem here is that *mantissa* bits are too sensitive to the number changes, making it difficult to leverage the trailing zero benefits. Suppose we know the precision of the incoming numbers. In that case, we can provide just enough precision support by eliminating the less significant *mantissa* bits. At the same time, we are still able to provide a guaranteed precision float for downstream analytic.

Table 1: Bounded float still keeps decimal precision

Real number		Sign	Exp(8 bits)	Mantissa(23 bits)	Actual value
3.14	binary	0	10000000	10010001111010111000011	3.1400001
	bounded-2	0	10000000	10010001100000000000000	3.13671875
					≈ 3.14

Table 1 shows float representation of a given value 3.14. As is shown in the binary row, the float format saves the closest approximation as 3.1400001. If we only care about two decimal positions after the decimal point, as is shown with *bounded-2* row, we can remove some trailing bits, while its value is still possible to approximate to the original value with two places of precision. The removal of insignificant bits saves a lot of bits when representing a number with a given precision. This is one motivation for Buff. To efficiently determine the number of fractional bits need for different required precision, we run brutal-force verification on all possible numbers under each precision requirement. We get the maximal factional bits needed for its corresponding precision, as shown in Table 2, which becomes the lookup table for bounded precision. We only provide up to 10 digit places precision here as it is good enough for most datasets. We can extend to higher precision, but it is less meaningful as commonly used float only has 52 bits for mantissa.

Table 2: Number of bits needed for targeted precision

Precision	1	2	3	4	5	6	7	8	9	10
Bits needed	5	8	11	15	18	21	25	28	31	35

3.1.2 Bounded Range. Another observation for most application scenarios is bounded range, The measured value is delimited by a lower and an upper measuring bound limit that defines the measuring span. The bounded range could result from either the measuring range of the instrument (e.g., clinical thermometer measuring range from 35°C to 42°C), the physical definition of measurand (e.g., CPU usage is defined between 0% to 100%), or based on other domain knowledge. The bounded range is another feature that is helpful when compressing the data since we only focus on the given range, and we can encode the number in a more concise but accurate manner. Buff can work with no provided ranges, but as our experiments show, it hinders ingestion performance as the compression and allocation needs to find the minimal and maximal observed value when encoding a set of values. With bounded precision and range, we can get rid of the less useful and redundant bits initially designed for higher precision and broader range. With those two techniques, we can get a condensed representation of the input data. In addition to the compression benefits, Buff also adopts float splitting to organize our compressed data better.

3.2 Float Splitting and Compression

The basic idea of float spitting is decomposing the input number into integer and fractional parts. This division separates the given number by radix point in a logical way and organizes each component into a different physical location. We can then apply efficient compression to those two parts, respectively. Bit splitting on fixed-point numbers is intuitively easy to do as its fixed position for the decimal point. However, the splitting is not easily applicable to float data directly, since float numbers are not aligned by the radix point. We extract both parts of a given float number with bit operations. First, we extract the exponent with a bit-mask to locate the decimal point. With the exponent value, we then bit-shift the float number and extract all left-side bits (including one implicit bit) as the integer part, and right-side bits as the fractional part. Figure 3 shows how this process works for a float value 23.1415 with scale 4 represented in 32-bit IEEE format. The exponent (marked with green font) is decoded as 4, which means the integer part has 5 bits (with a single leading 1 given implicitly by IEEE standard). We then ① extract 4 leading bits from the mantissa(marked with red) and add a leading 1 to get the integer part. Based on Table 2, we extract 15 following bits as fractional part for required precision **4**. ② With the range information either given by user or detected by our encoder, we offset the integer part by the minimal value to further condense the range space. This splitting is fast and efficient with only bit-wise operations and no *if* branches. It can be further improved if the user-provided data range has the same exponent value (e.g. all values fall within the same power of 2), in which case those float numbers have a fixed decimal point location, and float splitting could be conducted easier with direct bit-masking.

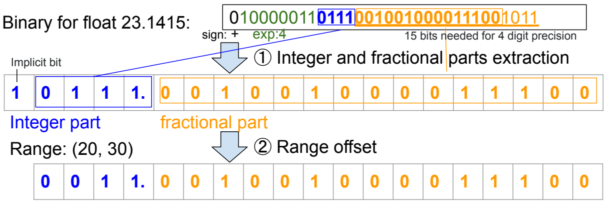


Figure 3: Extracting range bounded integer part and precision bounded fractional part for float number

The float splitting divides the given float numbers into two components and then compresses each component separately. We discuss its potential benefits on progressive filtering and data skipping later in Section 3.4.1. Here our analysis focuses on its compression efficiency in terms of compression ratio and reads/writes throughput. The encoded bit length of the integer part is determined by the range of input numbers, while the encoded bit length for the fractional part is determined by the actual precision required. We can compress both parts effectively with delta and bit-packing encoding for the integer part and bounded precision for the fractional part. The fixed-length encoding simplifies the encoding and decoding process and makes it possible for data skipping to be applied on the encoded bits directly, which improves performance.

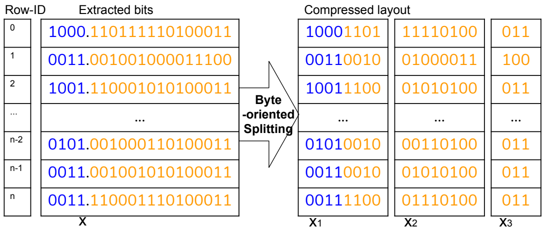


Figure 4: Byte-oriented columnar layout

However, float splitting introduces some overhead for the compression process. Byte-oriented reads/writes are more efficient than bit-oriented operations because of low-level data type boundary-crossing issues where the CPU needs to unpack and offset the bits to align them on byte boundaries. Assume we have m bits to read/write, this could be finished by $\lfloor \frac{m}{8} \rfloor$ times byte read/write, and at most one more bit read/write for remaining $m - \lfloor \frac{m}{8} \rfloor$ bits. Whereas float splitting divides m bits into x bits and y bits (where $x + y = m$), and this leads to $\lfloor \frac{x}{8} \rfloor + \lfloor \frac{y}{8} \rfloor$ times byte read/write, and at most twice bit read/write for remaining $x - \lfloor \frac{x}{8} \rfloor$ bits and $y - \lfloor \frac{y}{8} \rfloor$ bits. Additionally, the integer part usually takes less than one byte for many datasets because of the narrow value range for many measurands. To avoid slowness of bits read/write, we can always pad int bits with several following fractional bits to extend it into a byte. This keeps the fine-grained float splitting, but good read/write performance as well. If the goal is to save m bits anyway, we should split the bits in a more efficient way. This is the motivation of our advanced splitting version: byte-oriented float splitting.

Byte-oriented float splitting compression works similarly to the float splitting discussed previously. We first extract integer bits and just enough fractional bits that meet the precision requirement, then offset the intermediate value by minimal value before writing those bits in the byte unit separately. Each byte unit is treated as a *sub-column* and stored together. We store the remaining trailing bits of each record as the last sub-column, as is shown in Figure 4. We save the range and precision information as metadata along with compressed data to guarantee the decompression capability. With byte-oriented float splitting, compression and decompression are improved because of the fast byte reads/writes. We can also support custom variable precision materialization and achieve more efficient query execution because of a byte-oriented splitting layout.

3.3 Handling Outliers

In addition to variable precision materialization support brought by byte-oriented splitting, we can also handle outliers efficiently.

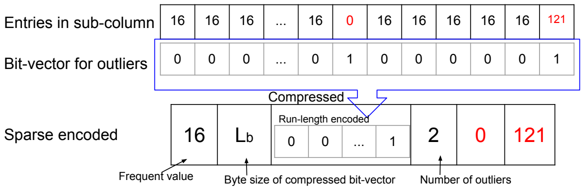


Figure 5: Sparse encoding condenses sub-column.

Outliers in the input data sequence are the data points that differ significantly from others. Outliers enlarge the value range significantly and further inflates the code space, which requires more bits to encode the input data sequence. For example, 99% of the input value can be represented by 1 byte, while 1% needs 4 bytes. Encoding all numbers in the same length inflates the code space to 4 bytes because of the 1% outliers. The code space inflating, if not handled well, will deteriorate the compression effectiveness and query efficiency. Outliers can be too large or too small that are far away from the common range. While in either case, the outliers usually only inflate the higher bits either with padding zeros or padding ones. When there are both large and small outliers, the padding bits can be any bit sequence between all zeros and all ones. This requires us to detect outliers and distinguish the most frequent value in the higher encoded byte. We first head sample the dataset and then apply Boyer–Moore majority voting algorithm [11] in each compressed byte unit to find the majority item (occurring more than 50%). If the majority item has a frequency greater than 90%, we get most frequent value and re-organize the corresponding byte column to achieve better compression by *sparse encoding* as shown in Figure 5: we keep the frequent value at the beginning and use a compressed bit-vector to indicate the outlier records, followed by outlier values in the current byte column. The outlier handling enables better compression performance and faster query execution with the aid of bit vectors. Higher byte columns can be skipped for most queries that target on some normal values only.

Sparse encoding requires a number of bits to be allocated per frequent value, to indicate if a value uses a frequent value or is an outlier. Compared with pure byte representation, sparse encoding further reduces the compressed data size if only there is a frequent item with a frequency greater than 1/8, assuming 1 bit overhead per record in the bit-vector. However, the outlier scheme comes with costs that special logic is required to handle outliers during compression and query execution. So we apply factor analysis on the frequency of frequent item through micro-benchmark with controlled outlier ratio. According to our experiments, sparse encoding achieves a significant query boost compared with pure byte representation when the frequency of frequent item is greater than 90%, which is the current default threshold in Buff.

3.4 Query Execution

In addition to the compression benefits, byte-oriented float splitting improves the query execution on the target float columns in the following aspects: byte-oriented data arrangement enables fast reading/writing and value parsing. Fixed-length coding helps effective data skipping to save CPU resources. The sub-column layout enables predicate push-down and progressive query execution. For efficient query execution directly on the encoded sub-columns, we apply query rewriting to decompose original query operators into a combination of independent query operators on sub-columns. Those translated query operators could be executed independently and combined to obtain the final results. If we evaluate each query operator on the sub-column one at a time, it is possible to skip some records that are determined to be qualified or disqualified already by previous sub-column query operators. Therefore, only a small amount of records need to be parsed and further checked.

3.4.1 Progressive Filtering.

3.4.1 Progressive Filtering. With byte-oriented float splitting compression, we can progressively evaluate the translated filter on each sub-column sequentially, and obtain the final results when all filters are finished. For a given predicate $x\ OP\ C$ where x is the attribute stored using Buff, OP is the operator, and C is the operand of this predicate. Instead of decoding each record and matching the predicate, we rewrite the filtering predicate on the target column into a combination of filtering predicates on its compressed sub-columns. Assume there are k sub-columns $x_1, x_2, x_3, ..., x_k$, we can get translated operands on those sub-columns as $C_1, C_2, C_3, ..., C_k$ and then apply filtering on each sub-column to obtain the final results [27].

To save computation in the filtering, we can avoid evaluating some records based on our current observations. During the progressive filtering process, all the intermediate results on each sub-column are shipped by a bit-vector, similar to late materialization

[49]. Two bit-vectors *RESULTS* and *TO_CHECK* are maintained to indicate the qualified rows and suspecting rows, respectively. The following filter only needs to further parse and check the suspecting rows in *TO_CHECK* bit-vector, then add qualified row numbers into *RESULTS* and suspensive row into *TO_CHECK* for the filter of the following sub-column. When the filtering level goes deeper, there will be fewer records that need to be checked. We can even early prune the filter execution in some special cases if there is no item in the *TO_CHECK* bit-vector. Even for the cases where early pruning is not happening, progressive filtering can avoid a lot of value parsing and check, especially valuable for the trailing bits sub-column where value parsing is more expensive. Progressive filtering is eligible for most filtering query executions. We describe its detailed implementation for *EQUAL* and *GREATER* predicate here. Other predicates such as *GREATER_EQUAL* or *LESS_EQUAL* can be achieved by logically combine existing predicates, *NOT_EQUAL* can be achieved by negating the *EQUAL* results.

Equality filtering predicate $x = C$ can be decomposed as $x_1 = C_1 \wedge x_2 = C_2 \wedge x_3 = C_3 \wedge ... \wedge x_k = C_k$ by query rewriting, which intuitively indicates the equality predicate could hold if and only if equality holds on all sub-columns. This can be formulated as:

$$x = C \iff \wedge_{i=1}^{k}(x_i = C_i)$$

The predicate push-down enables efficient progressive filtering on the sub-columns where we can execute the most selective predicate to maximize the filtering benefits - to filter out disqualified records as much as possible in the early phase. For filtering on the first sub-column, qualified rows are added into *TO_CHECK* bit-vector for filtering on the following sub-columns. Data skipping is used in the following evaluation, and only rows in *TO_CHECK* are parsed and further evaluated. Disqualified rows are removed from *TO_CHECK* bit-vector. In the last sub-column, all qualified rows are added into *RESULTS* bit-vector as a final result for the filtering predicate $x = C$.

With the previous example in Section 3.2, if given an equality filtering predicate $x = 23.1415$, we can extract the bits of bounded range and precision 0011.001001000011100 with the aid of range and precision in the metadata. Then we can rewrite the predicate as $x_1 = 00110010 \wedge x_2 = 01000011 \wedge x_3 = 100$. On the encoded file shown in Figure 4, progressive filtering starts with evaluating all values in the first sub-column x_1, and row 1 and $n-1$ are qualified with the first predicate, so they are added to *TO_CHECK*. On the second sub-column x_2, only row 1 and $n-1$ need to be checked, and row $n-1$ is disqualified, thus removed from *TO_CHECK*. On the third column x_3, only row 1 is evaluated and qualified, thus row 1 is added into *RESULTS* bit-vector returned as a final result.

Range filtering predicate $x > C$ can be decomposed as a combination of $x_i = C_i$ and $x_i > C_i$ after query rewriting. For example, $x_1 > C_1$ is sufficient to induce $x > C$. While $x_1 = C_1$ indicates further evaluation is needed for x_2. Then we proceed to sub-column 2 predicate evaluation. Similarly, if $x_2 > C_2$, we have $x > C$. Otherwise if $x_2 = C_2$, we proceed to sub-column 3. The evaluation is repeated until all sub-columns are processed. It can be formulated as:

$$x > C \iff \vee_{i=1}^{k}[\wedge_{j=1}^{i-1}(x_j = C_j) \wedge (x_i > C_i)]$$

For the first step, all records are parsed and evaluated. Only rows in *TO_CHECK* bit-vector from the previous step are decoded and evaluated in the following evaluation step. There are two categories of records generated for each step: qualified rows (where $x_i > C_i$)) are added to *RESULTS* bit-vector and unsettled rows (where $x_i = C_i$)) are added to a new *TO_CHECK* bit-vector. On the last sub-column, all qualified rows are added to *RESULTS* bit-vector, which is returned is a final result for the filtering predicate $x > C$. During any filtering step, an early stop is activated when the *TO_CHECK* bit-vector is empty and all following bits are skipped. *NULL* is returned for equality filtering, and the intermediate *RESULTS* bit-vector is returned for range filtering in this case. We introduced cold start filtering cases where all records are potentially qualified. However, our filtering operators can be applied to the cases where intermediate query results are provided in the form of bit-vector, by using given bit-vector as *TO_CHECK* for start sub-column.

3.4.2 Progressive Aggregation. Byte-oriented float splitting compression also facilitates some aggregation queries with the aid of techniques mentioned above [42]. Query execution could be improved either by data skipping for Min/Max queries or by transforming double operations into integer ones for Sum/Average queries.

Min/Max information is stored as metadata in the compressed byte array for each compressed segment. Therefore, efficient metadata lookup is sufficient to answer those queries. However, metadata lookup is not applicable for custom query scope predefined by a filter. Our compression approach also supports fast custom scope query execution, as encoded values are independently and neatly arranged in each sub-column level. Min/Max aggregation queries perform similarly with the equality query discussed previously. Starting with the first sub-column, the query executor finds the greatest/smallest value and puts all corresponding rows into *TO_CHECK* bit-vector. The query executor then proceeds to the next sub-column, evaluates all records corresponds to rows belonging to the previous *TO_CHECK* bit-vector, and generates a new *TO_CHECK* bit-vector if applicable. The min/max value is assembled in the end and returned as final results after all sub-columns are evaluated. A *RESULTS* bit-vector indicating the rows with min/max value is also returned along with query results. Min/Max queries are efficient with Buff because only very limited bits are evaluated during the query execution with the aid of progressive filtering and data skipping. In most cases, only a single row is qualified and added into *TO_CHECK* bit-vector for the first sub-column. Thus, the following sub-columns evaluation is merely skipping to the target row and assembling the specific single record, which is more efficient than decoding all bits and comparing the decoded values.

Sum/Average aggregation queries are very efficient with our compression approach with no full decoding needed. The byte-oriented splitting layout enables fast access to each byte/bits component corresponding to different precision. Rather than decoding every record into float value before applying the summation operation, value is accumulated on each sub-column in the form of small integer until the final summation of the intermediate sum results from each sub-column adjusted by its corresponding exponent and base value. The final summation is returned as *Sum* result or further processed to get the *Average* result. For example, on the encoded file shown in Figure 4, we first sum records in each sub-column, and get $sum(x_1)$, $sum(x_2)$ and $sum(x_3)$ respectively. Then we scale the summation with its corresponding basis. In this example, basis for column x_1 is 2^{-4}, x_2 is 2^{-12} and x_3 is 2^{-15}. Then the final sum result is $sum(x) = sum(x_1) * 2^{-4} + sum(x_2) * 2^{-12} + sum(x_3) * 2^{-15} + B \times N$

where B is the base value (min value) of this encoding block and N is the number of entries. We can further divide by N to get the average. The value is not fully decompressed to the original double value during the query execution, and all the execution is achieved with integer arithmetic, which improves the query performance.

3.4.3 Variable Precision Materialization and Aggregation. Reduced precision has emerged as a promising approach to improve power and performance trade-offs [15]. Customized precision originates from the fact that many applications can tolerate some loss in quality during computation, as in media processing and machine learning. The byte-oriented encoding layout also enables variable precision materialization in addition to full precision required by downstream analytic. For example, consider a system-agnostic query like *printf("%.2f", latitude) where AVG(temp)>90°F*. We can skip reading the trailing bytes/bits for *temp* column if we get *AVG(temp)>90°* with several leading bytes, and only several leading bytes of *latitude* are needed for two digits precision printing. For any reduced precision value read, Buff aligns the requested precision with the least byte boundary covering the required precision. For example, for a given compressed dataset with 4 bits for the integer and 18 bits for the fractional part (supports 5 digit precision at most), Buff materializes 2 bytes ($\lceil (4 + 11)/8 \rceil$) to support 3-digit precision query.

Buff also provides deterministic approximate aggregation query processing on data with reduced precision as suggested by DAQ [42]. Recall previous example in Section 3.2, on the encoded file shown in Figure 4, we can get deterministic approximate aggregation result by only reading the partial data. Similar to progressive aggregation execution in Section 3.4.2, we can get an sum approximation by reading the first sub-column as $sum(x) = sum(x_1) * 2^{-4} + B \times N$ with a deterministic lower bound $= sum(x_1) * 2^{-4} + B \times N$ where all trailing bits are zeros, and a upper bound $= sum(x_1) * 2^{-4} + (B + 2^{-4}) \times N$ where trailing bits are ones. Similarly for *avg* query, we get a deterministic bound by reading first sub-column as $[sum(x_1) * 2^{-4}/N + B, sum(x_1) * 2^{-4}/N + (B + 2^{-4})]$. As we read and process more data, we can provide a more accurate result.

3.4.4 Data Skipping. Data skipping is another technique used to enhance query performance. After scanning a sub-column, we know some records in the remaining columns do not satisfy the predicate, and we can skip them. The byte-oriented encoding layout enables efficient data skipping in the following aspects: byte-oriented layout guarantees efficient data reading and skipping without handling any cross-boundary issue. The fixed coding length for each component enables efficient skipping with simple bits length calculations. The trailing bits in the last component are mostly skipped based on the previous evaluation. In some cases, we can early prune the query execution and totally skip checking the following bytes and trailing bits if no records need to be further checked. Besides, data skipping enables fast records access for custom row-level scope queries, such as queries on a given time interval for time series data. Data can still be skipped efficiently without parsing the irrelevant entries.

3.4.5 Adaptive filtering. Different from prior works that use SIMD to accelerate query execution [20, 26, 41, 53], Buff uses a workload-adaptive execution strategy. When SIMD is available, Buff will use SIMD as much as possible. However, SIMD is not always the best when compared with the scalar progressive filtering. SIMD is usually good on low selective filtering, where a large percentage of records are evaluated. While progressive filtering is extremely efficient on high selective filtering tasks. For each sub-column, Buff will choose either the filtering strategy based on density ratio $r = \#candidates/\#records$. By default, SIMD execution is used for the first sub-column unless a sparse bit-vector is provided as input. For the following sub-columns, Buff uses SIMD execution if the density ratio is high and progressive filtering otherwise.

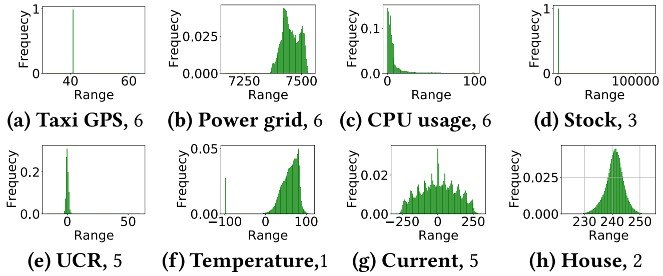
We perform a sensitivity analysis to find the crossover point. Progressive filtering takes more time as the density ratio increases and crosses over the constant SIMD cost at a density ratio 0.06, which is the current SIMD threshold used in Buff's adaptive filtering.

4 EXPERIMENTS

Compression approaches should keep the compressed data size smaller and support fast query execution, including fast filtering and aggregation queries. This section applies the compression techniques mentioned above on various datasets covering common applications with bounded float and reports both compression and query performance. In addition, we also include end-to-end complex query evaluation on Time Series Benchmark Suite (TSBS) [4].

All experiments were performed on servers with 2 Intel(R) Xeon(R) CPUs E5-2670 v3 @ 2.30GHz, 128GB memory, 250GB HDD, and Ubuntu 18.04. All our implementations and experiments are done in Rust 1.49.0 and we use AVX2_m256i for SIMD. To use ByteSlice for floats we re-implement ByteSlice in Rust, also implement Gorilla and Sprintz in Rust. For Gzip and Snappy, we use rust flate2 and parity-snappy libraries respectively. We apply Gzip level-9 in our experiments. All our implemented baseline achieve their claimed throughput (e.g., Snappy: *250MB/s* Gzip: *20-50MB/s*, Sprintz: *200MB/s*).

4.1 Datasets



(a) Taxi GPS, 6 **(b) Power grid,** 6 **(c) CPU usage,** 6 **(d) Stock,** 3

(e) UCR, 5 **(f) Temperature,**1 **(g) Current,** 5 **(h) House,** 2

Figure 6: Datasets have bounded range and precision

Our dataset covers various applications: stock data [1], taxi data from NYC (GPS) [6], DevOps monitoring (CPU) [45], city temperature (TEMP) [28], humidity [12, 13], power grid (PMU) [47], magnet motor (Current) [7], household electric power consumption (House) [2] and machine learning [34]. As most datasets are shared using a *txt* or *csv* format, we can easily extract the precision by counting the fractional digits. As we target a compression technique that would work with application code or a database, we evaluate an ingestion process that sends data in the IEEE 64-bit float format which is used by many applications and compression evaluations [14–16, 31, 40, 51]. Figure 6 shows the data range, distribution and scale (precision) for representative measurements of those datasets. The x-axis shows data range including all outliers if any. For GPS we use the latitude attribute. The data distribution varies on different measurements. Those datasets cover data with different precision with/without outliers. Many datasets have a very skewed distribution, where most records are clustered around a tiny area. As an example, most data points in NYC's GPS dataset fall into the city area with coordinate between *(40.702541, -74.007347)* and *(40.793961, -73.883324)*. The high decimal position with scale *6* guarantees submeter level precision of the data points. Similarly, all these measurements have limited range and bounded precision. In addition to those read world datasets, we also generate datasets from TSBS. Here, we use the IoT dataset with a scale 1000, which includes 9.1 billion data points about tracking information for trucks.

4.2 Optimized Baselines

During our evaluation, we found that the idea of bounded precision can improve Gorilla. Therefore, we provide an optimized version of Gorilla for comparison. In addition, we include fixed-point with bounded precision and ByteSlice variations for float as baselines.

4.2.1 Optimizing Gorilla. When we apply Gorilla on our datasets, we get a poor compression performance for most datasets. For datasets, such as CPU and Temperature, the compressed size is close or even larger than the original size. After detailed analysis, we notice that many data points in these datasets fluctuate over value 0 or ±2. There are two reasons why Gorilla is sub-optimal on datasets covering those values. First, float numbers fluctuating around 0 flips the sign bit (the leading bit). Also, when floats fluctuate around an absolute value 2, they flip the leading bit of exponent bits as exponent value 1 is encoded as 10000000 in a 32-bits float. Second, high precision floating numbers make the mantissa bits sensitive to the value change, especially for those less significant bits. For these two reasons, Gorilla fails to perform efficient trailing zero compression and performs poorly on these datasets.

To improve the compression performance of vanilla Gorilla, we have two different directions to work on:

- Leverage the leading zeros by offsetting the input numbers to avoid the "pitfall" value points.
- Use bounded precision float for the input numbers to eliminate the less significant bits in the mantissa part.

We verified those two solutions on our datasets. We found the former solution improves compression ratio performance slightly but at the cost of slowing down the compression throughput significantly, since we have to offset the value before we can apply gorilla encoding. However, the latter solution significantly improves compression performance. In the following evaluation section, we include the latter Gorilla version (GorillaBD). Nevertheless, GorillaBD only improves the compression performance and can do nothing to speed up the query execution because of the complex variable encoding/decoding mechanism inherited from Gorilla. This motivates us to devise a new compression for floats.

4.2.2 Bounded Fixed-point. As discussed in Section 2.1, fixed-point is not precision aware, since it uses every bit to approximate the value no matter what precision is needed. Additionally, fixed-point is not optimized for custom ranges, since it represents a range $-2^I \sim 2^I$. In order to include fixed-point into our evaluation, we implement bounded fixed-point (as *FIXED*) as a baseline with bounded precision applied to achieve just-enough precision and delta encoding used to offset the custom range. In FIXED, we scan the file and encodes the datasets with bounded fixed point representation.

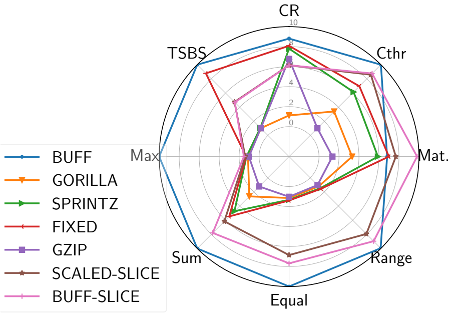


Figure 7: Compression performance overview (greater is better, CR: compression ratio, Cthr: compression throughput). Buff outperforms in query and compression throughput.

4.2.3 ByteSlice variations for float. ByteSlice is designed for an input code of a bit-packed integer, which is not directly applicable to float numbers. In order to apply ByteSlice on floats, a conversion from the integer code is needed to meet the ByteSlice input requirement. We can either borrow from Buff to extract the aligned bit representation of float (**BUFF-SLICE**) or scaling float into integer (**SCALED-SLICE**), and then bit-pack those code into ByteSlice. We compare against both ByteSlice variations.

4.3 Performance Overview

The radar chart of Figure 7 shows the overall performance of all float compression approaches above. We normalized the performance results into a range between 1 and 10 on each dimension uniformly, and then take the average of all workable datasets (Sprintz and SCALED-SLICE fail on some dataset which are not included here). ByteSlice variations do not provide a specialized aggregation solution, so we materialize before max and sum operation. For all metrics here, higher is better. As we can see, the state-of-the-art Gorilla approach does not work well on those datasets. Sprintz and FIXED perform well in terms of compression ratio and materialization performance on those datasets while performing poorly on other performance dimensions. ByteSlice variations are good on materialization performance but sacrifice compression ratio because of bit padding. Buff performs best in terms of query and compression throughput. In addition, it handles outliers data quite well and achieves a better compression ratio on outlier-included datasets (e.g., GPS, Stock dataset). For some query types, such as min/max and high selective equality filtering, Buff is up to 50× faster than other compression baselines. General-purpose approaches (e.g., Gzip and Snappy) perform poorly in these dimensions.

4.4 Compression Performance

We evaluate both the compression ratio and compression throughput on our datasets and benchmarks. A good compression must be fast and effective.

4.4.1 Compression Ratio. Figure 8 shows the compression ratio for each compression approaches. Gorilla compression performs poorly on most of our datasets as there are very limited repeated or similar adjacent values. GorillaBD works better than Gorilla since many trailing bits are formatted by bounded precision float, resulting in

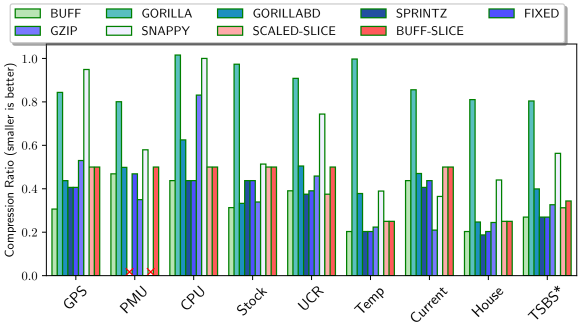


Figure 8: Compression ratio shows the Buff are comparable to the best and always better than vanilla Gorilla (Sprintz and Scaled-slice fails on PMU).

a better compression ratio. Buff performs similarly with Sprintz and FIXED since they leverage the bounded precision and range to map the numbers into almost the same finite encoding space. Value range and value precision determine the number of bits needed for the integer and the fractional parts. The slight compression ratio difference between Sprintz and Buff comes from the dividing of bits, which wastes some code space when the bits are not fully used. However, this brings a huge query benefits, which we will show in Section 4.5. In addition, as mentioned in Section 3.3, Buff can detect outliers and enables sparse encoding to compress the leading byte column further, resulting in a better compression ratio on GPS and Stock dataset compared with Sprintz and FIXED. ByteSlice variations pads the trailing bits for better query performance, so it is less effective than Buff, FIXED, and Sprintz on most datasets. The Byte-oriented compression approaches compress data by searching common sequences at the byte level, so Byte-oriented compression performs efficiently on temperature datasets because of the low cardinality brought by low data precision but performs poorly on datasets with less repeated values.

4.4.2 Compression throughput. In addition to the compression ratio, we also evaluate compression throughput. We count the time, including loading data from memory, applying compression, and writing back to memory. Figure 9 shows compression throughput for all float compression approaches. The leftmost bar corresponds to Buff with/without user-provided range stats. With ranges given by a user, our method can skip the range checking step in the float decomposing step and achieve higher compression throughput. We refer to it as "Fast-Buff" as is indicated by the star marker bar with the light green bar as a base. Overall, Buff outperforms all its competitors on most datasets. Gorilla relies on *XOR* between two adjacent values, then dynamically decides the number of leading and trailing zeros, and writes the middle residue bits. Gorilla's complicated encoding mechanism and variable encoded length limit its encoding throughput.

Buff is faster than FIXED and Sprintz because of its fast byte-oriented writing. According to profiling, Buff byte-oriented writing is 6% ∼ 12% faster than writing bits. Sprintz for float also relies on arithmetic multiplications to quantify input float values into integer values, which causes overflow issues such that Sprintz and SCALED-SLICE fail on the PMU dataset. Overall byte-oriented compression is slower than Buff and Sprintz. Snappy achieves the highest compression throughput on the CPU dataset, but Snappy performs poorly on the CPU dataset in the previous compression ratio experiment, where no compression is achieved at all. According to previous experiments, Gzip compression provides a higher compression ratio but uses more CPU resources than Snappy, thus lowering compression throughput. Compared with ByteSlice variations, Buff writes fewer bits but pays more overhead on writing the last trailing bits. Therefore, we can see very similar compression throughput on most datasets for both approaches, but Buff is faster on GPS and Stock datasets where more space is saved by sparse encoding.

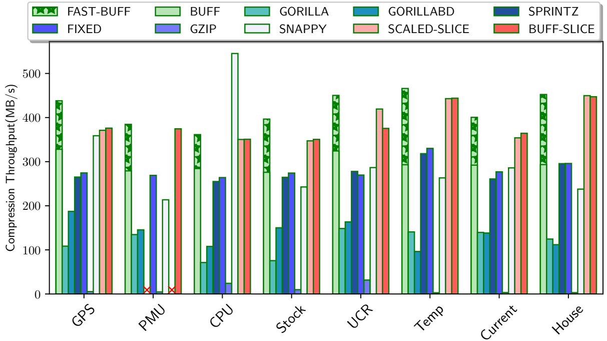


Figure 9: Compression throughput shows Buff performs best in most cases with user given range stats.

4.5 Query Performance

We also evaluate decompression and query performance. We measure execution time, including reading the data block from memory, parsing data, and evaluating the query. For all queries evaluated, we leverage each compression approach's characteristics to speed up its query as much as possible. Techniques, such as query rewriting, early pruning, and progressive filtering, are used for each approach whenever feasible, as discussed in each experiment section. We use the most frequent value (lower selectivity) in the corresponding dataset as a predicate operand to conduct a fair comparison for all equality filtering and range filtering. Progressive filtering is less efficient with low selectivity predicate as fewer values are skipped during query execution. We also run high selectivity queries to show Buff's potential. SCALED-SLICE and Sprintz bars for PMU dataset are missing as they fail on PMU dataset because of an overflow issue.

4.5.1 Filtering. We include filtering performance evaluation with query rewriting, early pruning, progressive filtering, data skipping and SIMD execution enabled when possible. For Buff, instead of materializing all records before applying the filtering predicate, we rewrite the query into query predicates combination on all sub-columns, then apply the translated query predicate on each sub-column sequentially. We also use adaptive filtering and data skipping with the aid of intermediate bit-vector results. For Sprintz and FIXED, we apply query rewriting techniques to avoid decompression overhead. For ByteSlice variations, we adopt their SIMD solution for filtering queries.

Our first experiment set shows low selective equality filtering performance. In Figure 10, Buff achieves outstanding performance

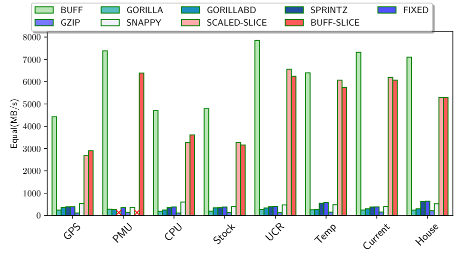


Figure 10: Low selective equality filter

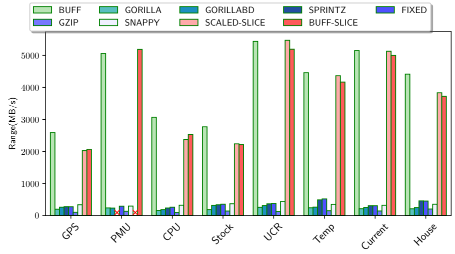


Figure 11: Low selective range filter

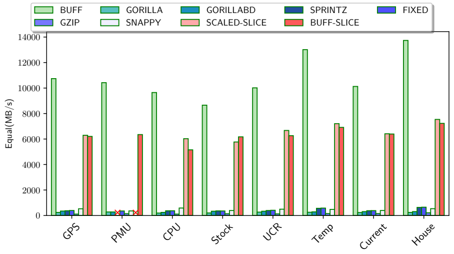


Figure 12: High selective equal filter

compared to all the other competitors. This speedup is mainly from adaptive filtering and data skipping. As we execute the predicate on the first sub-column, only a tiny part of the input records are qualified and need to be further checked in the subsequent sub-column predicates. For most datasets, less than 5% of entries are qualified and need to be further checked after filtering on the first sub-column with Buff. However, for GPS and Stock datasets, more than 90% entries are still qualified because of code space inflated by outliers. As we hit the frequent value in this case, we can skip decoding the first sub-column by directly using a bit-vector from sparse encoding, which speeds up the query. Sprintz and FIXED filtering use query rewriting to avoid record decompression as well. Compressed bits are extracted and compared with the translated to the target directly by using integer arithmetic. ByteSlice variations are always the second to the best approaches due to the SIMD speedup and early stopping. Buff has a better data and code locality, as it works on the same predicate for the current sub-column. While ByteSlice needs to load all sub-columns and jump to different sub-column with corresponding predicate evaluation in each iteration. Additionally, always using SIMD can slow down ByteSlice's throughput as a single match , resulting in SIMD loading and evaluating for all adjacent records within the same SIMD word length. In Buff, data skipping can efficiently jump to the target record and avoid unnecessary data loading. Except for these aforementioned approaches, decompression is required before predicate evaluation on float numbers. General-purpose byte-oriented compression approaches are sequential algorithms and need to decompress the last bits before the original data can be accessed. Gorilla encodes value with different bits and variable bits representation, which impedes query translation and data skipping. These bring a high latency for filtering the compressed data with those methods.

Figure 11 shows the low selective *greater than* range filtering experiments with the same operand. As we can see, the range filtering performs similarly to equality filtering but with smaller throughput overall. The selectivity of the range query is much lower than previous equality filtering. There are more qualified records in this case, which triggers the *IF* branch that edits the bitmap more frequently. Thus filtering throughput is deteriorated compared with previous equality filtering experiments. Similarly, ByteSlice variations is slower than equality filtering because of more comparison and less early stop. However, Buff still performs best. Similar to equality filtering, all other approaches require full decompression. As we can see from the figure, their filtering performance is proportional to their materialization performance, which we will discuss later in Section 4.5.3. Figure 12 shows high selective equality queries that filter out majority of records. Buff is more efficient on the high selective query as more bits are skipped by progressive filtering. Overall, Buff achieves 20× for average query speedup compared with the second-best float compression approach. Buff has similar trends on high selective range queries, and it achieves 16× on average query speedup compared to the second-best float compression method (we omit the figure).

4.5.2 Aggregation. An aggregation query is another frequent query for numeric data analysis. We evaluate the query performance of *min/max* and *sum* for all compression approaches.

Figure 13 shows *min/max* query throughput. The query time includes bit parsing and query execution. Even though we can quickly extract min/max value from data statistics in the metadata block for Buff, we force aggregation query execution on compressed data for a fair comparison, and this is necessary for aggregation with a filter. In addition to this setting, we enable all the techniques used in previous experiments. For *min/max* aggregation, we can use progressive filtering and data skipping for Buff. For FIXED and Sprintz, we find the *min/max* integer code directly then convert this value into float numbers. For other approaches, float numbers are decompressed for evaluation. The result shows that Buff achieves 22× on average query speedup than the second-best float compression approaches Sprintz or FIXED. The *min/max* value is usually with a low frequency such that only a tiny portion of entries are added to the bit-vector and need to be checked in the following sub-columns. This makes Buff extremely efficient for *min/max* aggregation queries since a huge amount of entries (bits) are skipped during the evaluation step.

The sum query performs similarly to materialization as all compressed data has to be decompressed, and all techniques mentioned previously are ineligible in this scenario. Figure 14 shows the *sum* query throughput on the same dataset. Buff outperforms all its competitors. For Sprintz, FIXED and Buff, we use lazy materialization to avoid the overhead of full decompression for each entry. The sum intermediate result is calculated with an integer representation for Sprintz and FIXED, then converted back to float numbers at the very end. This is similar to sum operator with Buff, where the sum is calculated on each sub-column with its integer representation respectively and concatenated into a final float-point result in the end. The performance difference between those approaches comes from the more efficient reading and parsing of Buff and no data dependency between adjacent numbers. All other approaches perform poorly since full decompression is inevitable.

4.5.3 Full Materialization. Materialization performance varies a lot for each compression approach. Overall, materialization overhead is comprised of bit parsing time and assemble time. Fixed-length

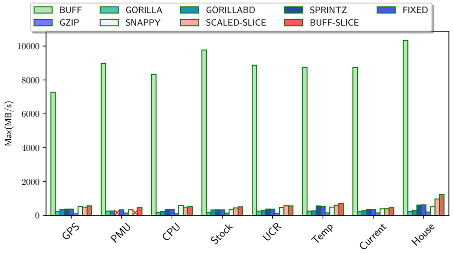


Figure 13: Max throughput

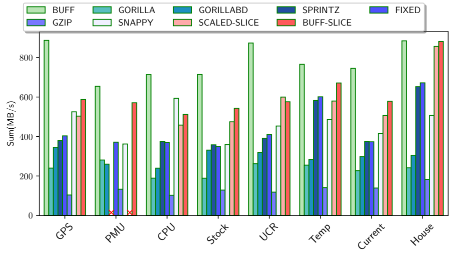


Figure 14: Sum throughput

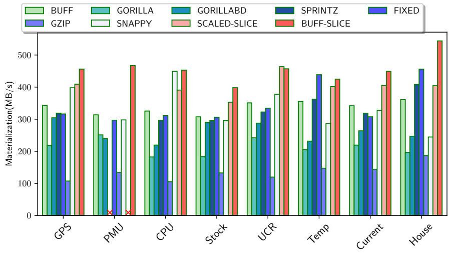


Figure 15: Materialization throughput

encoding helps for bit parsing. As is shown in Figure 15, Sprintz and FIXED perform better than Buff as they read single fixed-length data chunk during materialization, which is faster than Gorilla that reads variable-length bits for each record, and Buff where multiple data chunks need to be fetched as requested, then parsed and concatenated into a float number. However, Buff is still comparable with Sprintz and FIXED thanks to the efficient reading and parsing of Buff. ByteSlice variations are faster than Buff because of its fast byte reading and decoding benefits. Since Buff bit extraction is more efficient than the scaled version on decompression, BUFF-SLICE is always faster than SCALED-SLICE. Gzip is the slowest among all candidates since it has to decode Huffman encoding before restoring the recurring sub-sequences. Snappy achieves a better performance than Gzip because it skips Huffman encoding for higher throughput.

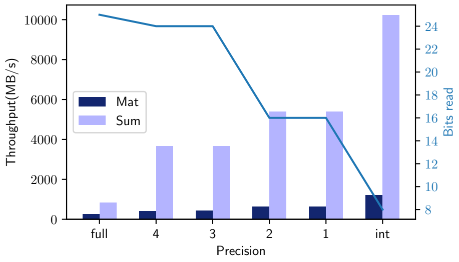


Figure 16: Materialization and aggregation with targeted precision boosts throughput by only reading the leading bytes.

4.5.4 Materialization and Aggregation with Requested Precision. We also show the versatility of Buff on variable precision materialization and aggregation support, which are especially useful when users only care about limited precision in formatted output and fast estimation of aggregation queries. Figure 16 shows the materialization and aggregation with requested precision on the UCR dataset. The red line-based right y-axis shows the bits needed to support the requested precision. The query performance linearly increases as bytes read decreases. A single expensive bit read can significantly slow down the query performance, as we can see from the first set of results on full data with 3 bytes and 1 bits read.

4.6 Benchmark Evaluation

In this set of experiments, we show complex query performance with TSBS. On TSBS, we generate "last-loc", "low-fuel" and "high-load" queries, which respectively include projection on longitude and latitude attributes (project), filtering on a fuel_state attribute containing single sub-column with Buff encoding (range-single), and filtering on current load attribute (range). In addition to the float-related operations, those end-to-end queries also include any filter, join, sorting on timestamp, string, or integer attributes. The runtime for these columns is independent of float compression approaches, so we take do not show them and only focus analysis on query runtime of float attributes. For Gorilla, the float-associated cost makes up 54% to 76% of the total runtime. However, Buff can diminish those costs to 1 or 2 orders of magnitude smaller on those queries. Figure 17 highlights the query runtime on float attributes, Buff always performs the best over all other baseline and is faster than ByteSlice variations on TSBS range filter queries since adaptive filtering chooses progressive filtering execution for the highly selective filter.

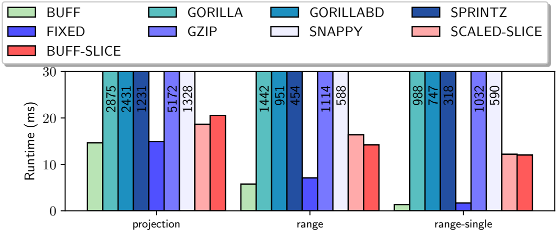


Figure 17: TSBS query runtime for float attributes.

5 CONCLUSIONS

This paper proposes Buff, a novel decomposed float compression for low precision floating-point data generated everywhere and every day. Buff uses "just-enough" precision for a given dataset, and leverages the data distribution feature (range, frequent value) to simplify its encoding mechanism and enhance the compression performance. The compression supports efficient in-situ adaptive query on encoded data directly. In addition, Buff provides fast aggregation and materialization for different precision levels. Buff's precision bounded technique is also applicable to the state-of-the-art Gorilla method and improves Gorilla's compression performance significantly. Our evaluation shows Buff achieves fast compression and query execution, while keeps comparable compression ratio to the best approaches. Future work includes studying the performance of Buff for common downstream complex analytical tasks, such as classification [39], clustering [36, 37], and anomaly detection [10].

ACKNOWLEDGMENTS

We thank the reviewers for their helpful feedback. This work was supported by the CERES center and gifts from NetApp, Google DAPA, Cisco Systems, and Exelon Utilities. Results presented were obtained using the Chameleon testbed supported by the NSF.

REFERENCES

[1] Huge stock market dataset. https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs.

[2] Individual household electric power consumption data set. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption.

[3] Influxdb - open source time series, metrics, and analytics database. https://www.influxdata.com/.

[4] Time series benchmark suite (tsbs). https://github.com/timescale/tsbs.

[5] Time-series data simplified. https://www.timescale.com.

[6] Tlc trip record data. www.nyc.gov/html/tlc/html/about/triprecorddata.shtml.

[7] Torque characteristics of a permanent magnet motor. https://www.kaggle.com/graxlmaxl/identifying-the-physics-behind-an-electric-motor.

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

[9] 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.

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

[11] R. S. Boyer and J. S. Moore. Mjrty—a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.

[12] J. Burgués, J. M. Jiménez-Soto, and S. Marco. Estimation of the limit of detection in semiconductor gas sensors through linearized calibration models. *Analytica chimica acta*, 1013:13–25, 2018.

[13] J. Burgués and S. Marco. Multivariate estimation of the limit of detection by orthogonal partial least squares in temperature-modulated mox sensors. *Analytica chimica acta*, 1019:49–64, 2018.

[14] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, 2008.

[15] S. Cherubin and G. Agosta. Tools for reduced precision computation: A survey. *ACM Computing Surveys (CSUR)*, 53(2):1–35, 2020.

[16] F. De Dinechin, J. Detrey, O. Cret, and R. Tudoran. When fpgas are better at floating-point than microprocessors. In *FPGA*, volume 8, page 260, 2008.

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

[18] H. Dietz, B. Dieter, R. Fisher, and K. Chang. Floating-point computation with just enough accuracy. In *International Conference on Computational Science*, pages 226–233. Springer, 2006.

[19] 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.

[20] 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.

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

[22] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.

[23] B. Hentschel, M. S. Kester, and S. Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 857–872, 2018.

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

[25] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless compression of predicted floating-point geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[26] H. Jiang and A. J. Elmore. Boosting data filtering on columnar encoding with simd. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, pages 1–10, 2018.

[27] 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.

[28] K. Kissock. University of dayton kettering labs window energy use analysis. 2007.

[29] 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.

[30] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[31] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[32] C. Liu, M. Umbenhower, H. Jiang, P. Subramaniam, J. Ma, and A. J. Elmore. Mostly order preserving dictionaries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1214–1225. IEEE, 2019.

[33] I. Müller, A. Arteaga, T. Hoefler, and G. Alonso. Reproducible floating-point aggregation in rdbmss. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1049–1060. IEEE, 2018.

[34] J. Paparrizos. 2018 ucr time-series archive: Backward compatibility, missing values, and varying lengths, January 2019. https://github.com/johnpaparrizos/UCRArchiveFixes.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.

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

[43] 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.

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

[45] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423*, 2020.

[46] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742, 2012.

[47] E. Stewart, A. Liao, and C. Roberts. Open μpmu: A real world reference distribution micro-phasor measurement unit data set for research and application development. 2016.

[48] J. Stokes. *Inside the machine: an illustrated introduction to microprocessors and computer architecture.* No Starch Press, 2007.

[49] 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.

[50] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang. Accelerating generalized linear models with mlweaving: A one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment*, 12(7):807–821, 2019.

[51] N. Whitehead and A. Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21(1):18749–19424, 2011.

[52] Wikipedia contributors. Snappy (compression) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Snappy_(compression)&oldid=977673115, 2020. [Online; accessed 14-September-2020].

[53] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[54] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.