Machete: An Efficient Lossy Floating-Point Compressor Designed for Time Series Databases

Yang Shi[†], Xiangyu Zou[†], Xinyu Chen[‡], Sian Jin[*], Dingwen Tao[*], Cai Deng[†], Yufan Chen[†], and Wen Xia[†,§,✉]

[†] Harbin Institute of Technology, Shenzhen [‡] Washington State University
[*] Temple University [*] Indiana University [§] Peng Cheng Laboratory
✉ xiawen@hit.edu.cn

Abstract

As time series data become popular, their volume increases rapidly. Time series databases are designed for such data, and they process data in short slices, meaning that the compression units for compressors are small. How to compress the short slices of floating-points while reserving a high compression ratio and a high decompression speed remains a problem.

To solve the problem, we propose a lossy compressor Machete. It uses an efficient hybrid encoder of Huffman encoding and variable length quantity (VLQ). Adaptive encoding selection makes it excel on short-slice data compression ratio, while the simple framework ensures fast decompression. We also find a limitation in VLQ and propose the optimal VLQ to further improve the compression ratio.

Our evaluation on four real-world datasets shows that Machete outperforms state-of-the-art compressors by 32%−80% on compression ratio, and achieves the fastest decompression speed on two datasets. When applied to a well-known time series database InfluxDB, Machete saves disk usage up to 79% and improves the query performance of the InfluxDB database by saving I/O.

1 Introduction

Time series data are widely used in various fields [1–4]. This kind of data is made up of a series of timestamp-value pairs with string tags, where values are usually floating-points, to record how a variable (e.g., voltage or power [5]) changes over time. Generally, these data are used to identify trends and patterns in analysis works, such as prediction [1, 6] and anomaly detection [7, 8].

It is common to collect several terabytes of time series data every day and to store them for years [9–11]. To process the data efficiently, time series databases are designed with convenient analysis functions and efficient data layouts for time series data [9, 11, 12]. As TSDBs become popular, TSDB compressors are receiving more attention because of the huge data scale. With studies on current TSDBs, we summarize **three remarkable points** on compression in TSDBs: (1) **Strict error bounds make lossy compressors practical for most cases**. This is because time series data are mainly used for trend analysis and prediction [1, 2, 6–8], and small errors caused by lossy compressors have little impact on the trend. In addition, small errors could be fixed by rounding for fixed-point data. Lossy compressed values of data with l decimal places can be losslessly restored by rounding if compression errors are small than $\pm 0.5 \times 10^{-l}$. Therefore, lossy compressors can be used for general cases with lossless ones for extreme cases where no errors are allowed and values have uncertain

decimal places. (2) **Data are compressed in short slices**. TSDBs compress data in short slices separately, which helps to reduce read amplification (in decompression) and process data parallelly. (3) **Decompression speed is critical.** This is because decompression speed directly affects the query throughput and latency. Compression speed is less important because data compression is asynchronous with the write response and thus has little influence on write latency and throughput.

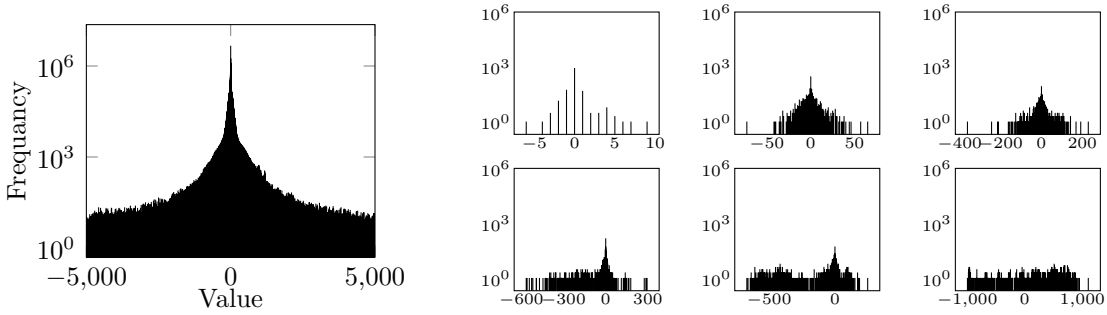
There are many previous works on time series data compression. Gorilla [9], Chimp [13], and Elf [14] are lossless TSDB compressors. They follow a similar idea of XORing data and striping heading/trailing zeros to save space. LFZip [15] is a lossy compressor for time series data. It uses a prediction-correction framework that first makes a prediction for each datum and encodes the differences between the predictions and the actual values. SZ3 [16–18] is another state-of-the-art lossy compressor but is designed for high-performance computing. Its framework is similar to LFZip, but it uses different predictors and encoders. However, **lossless TSDB compressors hardly achieve high compression ratios, while existing lossy compressors encounter significant performance drops in short-slice compression.**

To this end, we propose a novel lossy compressor, Machete, that combines the high compression ratio of lossy compressors with the short-slice-friendly and high decompression speed of TSDB compressors. Machete follows the prediction-correction framework that LFZip and SZ3 use. But unlike LFZip and SZ3 focusing on high-accuracy predictors, it uses a simple predictor and achieves the above remarkable points with an efficient encoder. We summarize our main contribution as follows: **(1) We propose a hybrid encoder.** The differences between predictions and actual values show complex characteristics in short-slice compression, so we proposed an encoder that adaptively and swiftly assigns each datum to the best-fitting encoding between Huffman encoding and VLQ. **(2) We propose the optimal VLQ.** We lift a limitation in VLQ for array compression, making it configurable, and propose a method to find an optimal configuration according to input to improve the compression ratio. **(3) We apply Machete to a popular time series database InfluxDB**, and evaluate its performance with other state-of-the-art approaches.

Evaluations on real-world datasets suggest that Machete outperforms the second-best by 32%−80% on compression ratio and decompresses as fast as the TSDB compressors. Furthermore, when running in InfluxDB, it saves 47%−79% disk space and improves query throughput and latency because of I/O saving without slowing down write throughput.

2 Motivations

Both LFZip [15] and SZ3 [16–18] employ a **prediction-correction framework**. The framework has two phases in compression: prediction and encoding. In the prediction phase, a predictor generates a prediction for each datum and then outputs the quantized differences between the predicted value and the actual one. In the encoding phase, an entropy encoder encodes the quantized differences. The prediction phase transforms floating-points into integers gathered around zero, making them easier to encode. In decompression, the quantized differences are decoded, after which the same predictions are made and corrected with the quantized differences.



(a) Overall distribution (b) Distributions on different slices of 1000 data

Figure 1: Distributions of quantized difference values on the GeoLife dataset [19–21]. Note that the y-axis is in the logarithm scale, and the presented range is cropped to have a better view of the majorities.

Compressors with such a framework encounter several difficulties in a time series database (TSDB). On the one hand, **a complex predictor usually gates the compression/decompression speed.** We evaluate the Normalized Least Mean Square Predictor (NLMS) of LFZip and find its throughput to be about 60MB/s, which is unacceptable for a TSDB. On the other hand, **a pure entropy encoder is not suitable for short-slice compression.** Fig. 1 lists the frequency distribution of all quantized differences from the GeoLife dataset [19–21] and those of some short slices. It shows the difficulties for entropy encoders:

(1) **Redundancy reduces**. The frequencies in Fig. 1b are much lower than those in Fig. 1a, which is a certain result of data slicing. Redundancy reduction may not influence data entropy, but it increases the time and space cost of entropy recording (e.g., the Huffman tree of Huffman encoding or the probability table of arithmetic encoding). Meanwhile, leveraging redundancy across slices is not allowed as it brings sequence processing and read amplification, which contradicts the intention of data slicing performed by TSDBs.

(2) **Distributions vary**. As shown in Fig. 1b, the distributions can be very different from slice to slice. Such variance affects data entropy, and in some extreme cases (e.g. the last figure in Fig. 1b), the frequencies are evened.

To handle these difficulties, we propose the Machete compressor. Unlike previous lossy compressors that focus on predictor design, we pay more attention to the encoder phase. While simplifying the predictor for speed, we propose a hybrid encoder that adaptively applies two high-speed encodings: Huffman encoding and variable length quantity (VLQ). The encoder separates the low-redundancy part from the high-redundancy part in an efficient way and then encodes them with VLQ and Huffman encoding, respectively. Furthermore, we propose the optimal VLQ to maximize the compression ratio of VLQ, which lifts the limitation due to single value encoding (will be detailed in Section 3.3), and other minor improvements including Huffman tree structure compression and Huffman decode table.

3 Design and Implementation

Fig. 2 illustrates the design overview. A time series database passes an array of floating-points as the input of the Machete compressor. As the first step, Machete

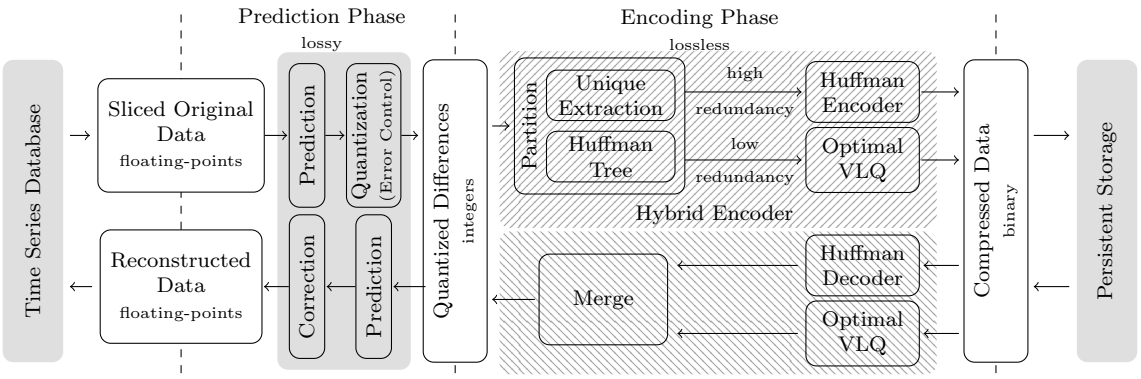


Figure 2: Overview of our approach. The solid-shaded area is the prediction phase, and the line-shaded area is the encoding phase.

makes a prediction of each value, calculates the difference between the predicted value and the actual one, and quantizes the difference into an integer. Some information is lost during the quantization, but the loss is guaranteed to be within a user-specified bound. After getting all the quantized differences, Machete compresses them with a lossless encoder. During decompression, the decompressor decodes the quantized differences, makes the same predictions, and then corrects the predictions with quantized differences to reconstruct the floating-point data.

3.1 Prediction, Quantization, and Error Control

Let x_i be the i^{th} datum of the input array, p_i be the prediction of x_i, and y_i be the corresponding reconstructed value. Then, the quantized differences δ_i between x_i and p_i is calculated with Formula 1:

$$\delta_i = [\frac{x_i - p_i}{2\Delta}] \tag{1}$$

In Formula 1, Δ is the user-defined error upper bound, and $[f]$ means rounding f to an integer. y_i is reconstructed (or corrected) using Formula 2:

$$y_i = p_i + 2\Delta\delta_i \tag{2}$$

And we can prove the error bound is strictly followed:

$$\delta_i = [\frac{x_i - p_i}{2\Delta}] \Rightarrow \frac{x_i - p_i}{2\Delta} - \frac{1}{2} < \delta_i \leq \frac{x_i - p_i}{2\Delta} + \frac{1}{2}$$
$$\Leftrightarrow -\Delta < -x_i + (p_i + 2\Delta\delta_i) \leq \Delta \Leftrightarrow -\Delta < y_i - x_i \leq \Delta \Rightarrow |x_i - y_i| \leq \Delta$$

Machete predicts the current value x_i equals the last one x_{i-1}. However, the decompressor does not know x_{i-1} but knows y_{i-1}, so we have to let $p_i = y_{i-1}$ for both the compressor and the decompressor. In addition, x_1 is stored in the compression result without compressing and is assigned to y_1 directly.

3.2 Hybrid Encoder

After floating-points are transformed into quantized differences, the hybrid encoder is used to compress them. It combines Huffman encoding and variable length quantity (VLQ) to handle various data distributions. Huffman encoding assigns fewer bits to high-frequency values, while VLQ assigns fewer bytes to close-to-zero values. In

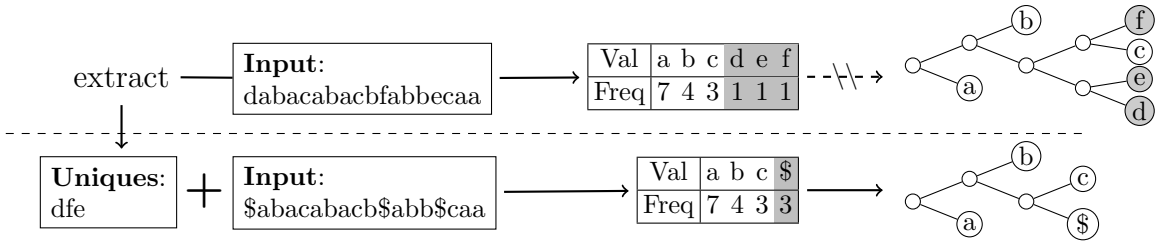


Figure 3: Removing unique values from Huffman encoder

Fig. 1, we know that the quantized differences are likely to be close to zero in general, which is why VLQ is chosen to compress the low-redundancy data that Huffman encoding is not good at.

With the underlying encoders chosen, the remaining problem is to properly partition the inputs and assign them to the encoders. In fact, the Huffman encoding already does part of the job. If we use a Huffman encoder to encode all the data, the low-redundancy data is within the Huffman tree: all presented values are recorded in the tree leaves exactly once. However, instead of directly using VLQ to compress the recorded values in the Huffman tree, we add an extra step to extract unique values (values that appear only once) from the Huffman encoder. Fig. 3 illustrates how to extract the unique values. After we count the frequencies of each value as the Huffman encoding required, we learn the unique symbols, record them in another array, and replace them with the same special value ('$' in Fig. 3). After we extract the unique values and finish the Huffman encoding part, we compress the Huffman tree values together with the unique values using VLQ. When decoding, we load a unique value from the array of uniques every time we meet the special value.

Further analysis of the GeoLife dataset [19–21] suggests the high efficiency of our hybrid encoder. First, **the Huffman tree spends much space on recording present values.** There are 181 values on average in a slice of 1000 data. Assuming the average code length of a value to be 7, then the Huffman code bitstream of the values is about 875 bytes. Meanwhile, $181 \times 4 = 724$ bytes are used to record the presented value in the Huffman tree (values are 32-bit integers). Second, **among the values,** 39% **of them are unique on average.** Unique values are bad cases for Huffman encoding since they will lead to a long code length, a large Huffman tree, and a long tree construction time. As a result, it is critical to extract and compress them in other ways when compressing short-slice data.

3.3 Optimal Variable Length Quantity

Variable length quantity (VLQ, also known as variable length integer) is an encoding that encodes an integer with a variable length of bytes. It strips heading zeros (or ones if negative) of the integer (but at least one heading zero/one bit is kept as the sign bit), saving space when the integer is close to zero. VLQ uses the most significant bit of each byte to be a flag indicating whether there are more bytes to read and stores data in the rest bits.

If we put the flag bits scattered in each byte together, we find that VLQ is de facto using a prefix code to record the byte number, as shown in Tab. 1. Because VLQ is designed to encode individual integers, each encoded integer needs to be byte-aligned, that is, having a multiple of 8 bits. However, Machete uses VLQ to encode arrays, so

Table 1: The fixed flag-length mapping due to the byte alignment.

VLQ Format	Flag bits	Data Length
0 xxxxxxx	0	1 byte (7 bits data)
1 xxxxxxx 0 xxxxxxx	10	2 bytes (14 bits data)
1 xxxxxxx 1 xxxxxxx 0 xxxxxxx	110	3 bytes (21 bits data)
...	...	...

the byte-aligned limitation is removed, and the flag-length mapping can be redefined to improve the compression ratio.

Obviously, for different arrays, there are different optimal mappings that maximize the compression ratio. We will first formalize the problem of finding an optimal mapping and then introduce how to solve it. Because flags are either a single '0' or in the form of '11...10', given the bit length of a flag, there is one and only one flag whose length matches. Let $L(flag)$ be the length of the given flag, and $f(L(flag)) =$ *data bit length* be a mapping. Let $L(\delta)$ be the length of the quantized differences δ, so that $L(\delta) = l(l > 0)$ iff $\delta \in [-2^{l-1}, 2^{l-1})$ and $\delta \notin -[-2^{l-2}, 2^{l-2})$. Specially, let $L(\delta) = 1$ when $\delta = 0$. Given a mapping f, we can construct a function $g(L(\delta)) = L(flag)$, so that $f(g(L(\delta))) \geq L(\delta) > f(g(L(\delta))-1)$. It means that using mapping f, δ is encoded into $g(L(\delta))$ bits of flag and $f(g(L(\delta)))$ bits of data. Then, the encoded bit length of an array of δ (noted as A_δ) using f is calculated by formula 3:

$$SIZE(f, A_\delta) = \sum_{\delta \text{ in } A_\delta} \left(f\left(g\left(L\left(\delta\right)\right)\right) + g\left(L\left(\delta\right)\right)\right) \tag{3}$$

An optimal mapping for A_δ is a mapping that minimizes $SIZE(f, A_\delta)$. For convenience, we will also represent f as an array $[f(1), f(2), \cdots, f(n)]$ where n is the maximum flag length in f's domain.

The optimal mapping can be found using dynamic programming. We define subproblems as follows: Subproblem $SP(l)$ is to find the optimal f_l while assuming all δ is no longer than l, which means using $L_l(\delta) = \min(L(\delta), l)$ to replace $L(\delta)$. Then, proving the optimal substructure is to prove that if $f = [f(1), f(2), \cdots, f(n)]$ is an optimal mapping of $SP(f(n))$, then $f' = [f(1), f(2), \cdots, f(n-1)]$ is an optimal mapping of $SP(f(n-1))$. This can be proved by contradiction:

Proof. Assume $f'' = [f''(1), f''(2), \cdots, f''(m) = f(n-1)]$ is a better mapping than f' of $SP(f(n-1))$, that is, $SIZE(f'', A_\delta) < SIZE(f', A_\delta)$. Then, we construct a new mapping $f^* = [f''(1), f''(2), \cdots, f''(m), f(n)]$. Because $SIZE(f, A_\delta) - SIZE(f', A_\delta) = SIZE(f^*, A_\delta) - SIZE(f'', A_\delta)$, we can infer that $SIZE(f, A_\delta) > SIZE(f^*, A_\delta)$, which means f^* is better than f and contradicts with that f is optimal. Therefore, f' is an optimal mapping of $SP(f(n-1))$ if f is an optimal mapping of $SP(f(n))$. □

As a result, we can solve the problem by solving $SP(1), SP(2), \cdots, SP(l_{max})$ (where $l_{max} = \max(L(\delta))$) step by step. For $SP(1)$, there is only one possible f, which is $[f(1) = 1]$. Note f_l as the optimal mapping of $SP(l)$, then f_l is one of the following: $[l]$, $f_1 + [l]$, $f_2 + [l]$, $\cdots$, $f_{l-1} + [l]$, where $f_x + [l]$ means appending l to the f_x. And we calculate the $SIZE$ of each candidate to find out the optimal one.

In addition, optimal VLQ also needs to have the optimal mapping recorded. Since the quantized differences are 32-bit integers, the range of f is $[1, 32]$, so we can use 32 bits to record f by setting the $f(1)^{th}$, $f(2)^{th}$, $\cdots$, $f(n)^{th}$ bits.

Table 2: List of time series datasets used for evaluation.

Name	Size	Decimal Places	Error Bound	Description
GeoLife	380MB	not fixed	1×10^{-6}	GPS trajectory data dataset collected in GeoLife project [19–21]
System	1.4GB	not fixed	1×10^{-3}	CPU and memory monitor data of our server collected with Telegraf (https://www.influxdata.com/time-series-platform/telegraf/)
REDD	431MB	2	5×10^{-3}	Low-frequency part of Reference Energy Disaggregation Data Set [5]
Stock	11GB	3	5×10^{-4}	Financial data from INFORE project (https://zenodo.org/record/3886895#.Y3H3QnZBybj)

3.4 Other Improvements

Besides the hybrid encoder and the optimal VLQ, we also have some minor improvement techniques. The first one is the Huffman tree structure compression. Huffman tree is a complete binary tree, so its structure can be stored by recording its leaf height from left to right. Transforming the Huffman tree to a canonical tree [22] makes each leaf no higher than any leaf at its right, thus sorting the leaf height array. Then, the leaf heigh array the records the tree structure can be easily compressed with run-length encoding. The second one is the Huffman decode table used by the ZStandard compressor [1]. Unlike a Huffman tree that decodes bit by bit, the decode table decodes one value within $O(1)$ time, improving the decode speed.

4 Evaluation

The evaluation has two parts, both of which are run in Ubuntu 18.04 on a server with an Xeon Gold 6154 CPU, 112.5 GiB DRAM, and 7200 rpm disks. In the first part, we compare Machete with other compressors, and in the second part, we apply Machete and some other compressors into InfluxDB [12], one of the popular open-source time series databases (TSDB) to explore the impact.

4.1 Direct Evaluation

We compare Machete with other compressors, including: (1) general purpose lossless compressors: Zlib[2] (also known as GZip) and ZStandard (ZSTD) [1], (2) TSDB lossless compressors: Gorilla [9], Chimp_{128} [13], and Elf [14], and (3) lossy compressors: LFZip [15], SZ3 [16–18]. In this part, we measure compression ratio (the ratio of original data size to compress size), compression speed, and decompression speed. Relatively, the compression ratio and decompression speed matter, just as the discussion in Sec. 2.

And the datasets used are listed in Tab. 2. All data are presented as 64-bit floating-points, and REDD and Stock have fixed decimal places. Tab. 2 also lists the default error bound used in our evaluation for all lossy compressors. For GeoLife and System, the error bound is small enough for most applications. For REDD and Stock, the error bound is small enough to retrieve the original data by rounding the reconstructed data to corresponding decimal places. Instead of compressing a dataset as a whole, we first slice the data into slices no longer than 1000 (which is the slice length used in InfluxDB) and then compress the slices separately.

Tab. 3 shows the evaluation result with the best record highlighted. Machete has the absolute advantage in compression ratio, outperforming the second-best by

[1]https://github.com/facebook/zstd

[2]https://www.gnu.org/ software/gzip/

Table 3: Evaluations on different datasets

Metric	Dataset	Zlib	ZSTD	Gorilla	$Chimp_{128}$	Elf	LFZip	SZ3	Machete
Compression Ratio	GeoLife	1.65	1.55	1.58	1.71	3.18	5.99	3.41	7.93
	System	20.44	19.50	6.71	4.85	5.18	17.11	15.94	37.98
	REDD	11.53	11.17	4.36	4.82	8.38	8.00	5.74	16.48
	Stock	5.16	4.68	1.73	3.93	6.50	9.65	11.82	21.22
Compression Speed (MB/s)	GeoLife	27.2	86.38	994.8	797.7	149.6	6.9	4.9	75.2
	System	137.8	439.44	2356.1	1082.2	497.8	10.6	24.4	195.4
	REDD	71.5	346.82	1731.5	844.0	176.9	7.8	11.1	120.5
	Stock	48.6	194.13	1007.7	843.4	137.4	8.8	8.1	153.2
Decompression Speed (MB/s)	GeoLife	208.2	358.19	908.8	854.4	538.5	10.0	21.7	954.0
	System	469.6	1217.2	2145.8	1198.3	1298.3	15.4	85.2	1940.9
	REDD	448.0	845.86	1612.2	1036.3	816.4	12.1	48.9	1348.3
	Stock	331.1	526.38	918.6	937.0	457.5	13.1	39.4	1382.7

32%−80%. Meanwhile, the decompression speed of Machete is also in the top tier, close to Gorilla. Regarding compression speed, Machete has no advantage, but the evaluation in Sec. 4.2 shows that it is sufficient for a time series database.

In the evaluation, LFZip and SZ3 perform much worse than expected, which is a result of the short-slice compression. We conduct another similar evaluation with slice length setting to 2^{16} and found the compression ratio of LFZip and SZ3 getting close to Machete, and they ran a few times faster. Meanwhile, the performance of Machete improves only a small amount. In other words, compared to LFZip and SZ3, Machete suffers less performance drop as the slices get shorter.

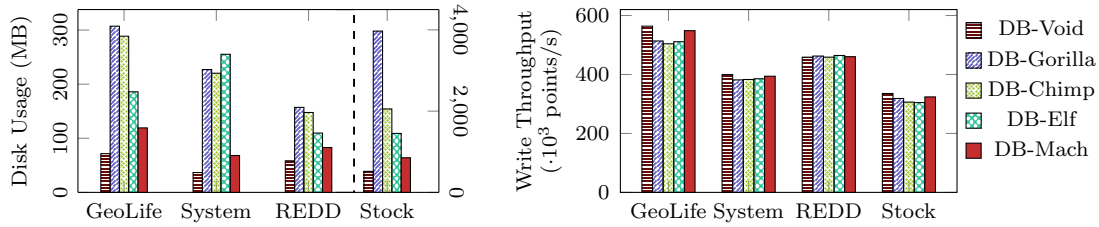
4.2 Database Evaluation

In this section, we apply $Chimp_{128}$, Elf, and Machete into InfluxDB to see the impact on disk usage, write throughput, and query latency/throughput while using different compressors. The databases are named DB-Chimp, DB-Elf, and DB-Mach respectively. In addition to the three we just mentioned, there are also DB-Void, which drops all floating-point data and returns 0 when queried, and DB-Gorilla (Gorilla is the original compressor of InfluxDB). DB-Void is used to show the upper bound of the improvement that a compressor can bring to the database. We use the same dataset lists in Tab. 2 in this part as well.

Disk Usage: We inject the datasets into the databases, and their disk usage is shown in Fig. 4a. DB-Void shows the space taken by non-floating-point data, i.e., timestamps, tags, indices, etc. The difference between DB-Gorilla and DB-Void shows that the floating-point data take the majority of the space and are hard to compress. On the other hand, the disk usage of DB-Mach is close to DB-Void. Compared to DB-Gorilla, DB-Mach saves disk space by 47%−79%. Compared to the second-best (excluding DB-Void), DB-Mach saves disk space by 24%−69% (or 41%−83% with space used by DB-Void stripped).

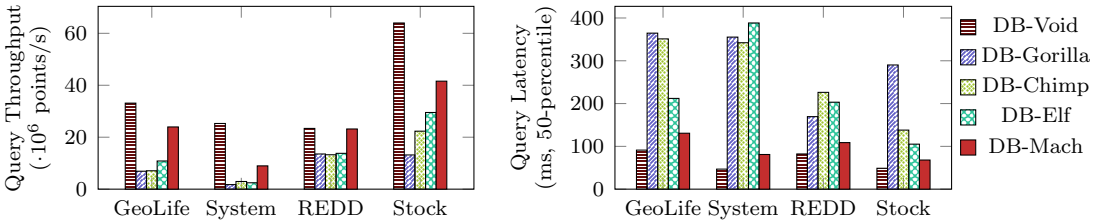
Write Throughput: We record the injection time and calculate the write throughput of each database. The result is shown in Fig. 4b. Since data write is asynchronous with data compaction, all database has similar write throughputs even though their compressors are different in compression speed.

Query Throughput and Latency: We perform massive queries and record their latencies. The queries ask for simple aggregations of data within a time range, which are common queries for time series databases. The overall query time is also recorded for database query throughput. The query throughput and 50-percentile



(a) Disk Usage. The Stock dataset uses the y-axis on the right.

(b) Write Thoughput



(c) Query Throughput

(d) 50-Percentile Query Latency

Figure 4: Database Evaluations

query latency are shown in Fig. 4c and 4d. The query throughput of databases is negatively correlated with disk usage. It suggests all the compressors are fast enough for InfluxDB, and the disk I/O becomes the bottleneck. Among the databases besides DB-Void, Machete is the best and outperforms the second-best by $1.4\times-3.0\times$.

5 Conclusion

We propose a lossy floating-point compressor Machete. It is aimed at high compression ratios while meeting the requirements of time series databases (TSDB). Instead of improving the predictor accuracy which likely causes high calculation overhead, Machete uses a simple predictor and focuses on efficient encoder design. The hybrid encoder it uses combines Huffman encoding and variable length quantity (VLQ), targeting fast processing on short-slice data. In addition, the concept of optimal VLQ is also proposed to further improve the compression ratio. Our evaluation shows that Machete outperforms the state-of-the-art compressors in time series database scenarios on compression ratio by 32%−80% with decompression speed similar to TSDB compressors. When applied to InfluxDB, it saves up to 79% disk space and improves query performance significantly due to the saved I/O. The source codes are available at https://github.com/Gyhanis/Machete.git.

Acknowledgements

This work is supported by the Major Key Project of PCL (Grant No. PCL2022A03), the Shenzhen Science and Technology Program (Grant No. RCYX20210609104510007, KJZD20230923114610021, and JCYJ20200109113427092), and the National Science Foundation (Grant No. OAC-2303064, OAC-2247080, OAC-2311876, and OAC-2312673).

References

[1] V. Dhar, C. Sun, and P. Batra, "Transforming finance into vision: concurrent financial time series as convolutional nets," *Big Data*, vol. 7, no. 4, pp. 276–285, 2019.

[2] Y. Zheng and X. Zhou, *Computing with spatial trajectories*. Springer Science & Business Media, 2011.

[3] J. Cheng and M. Mitzenmacher, "The markov expert for finding episodes in time series." in *DCC*, 2005, p. 454.

[4] N. Cruces, D. Seco, and G. Guitérrez, "A compact representation of raster time series," in *2019 Data Compression Conference (DCC)*. IEEE, 2019, pp. 103–111.

[5] J. Z. Kolter and M. J. Johnson, "Redd: A public data set for energy disaggregation research," in *Workshop on data mining applications in sustainability (SIGKDD), San Diego, CA*, vol. 25, no. Citeseer. Citeseer, 2011, pp. 59–62.

[6] M. Bahari, I. Nejjar, and A. Alahi, "Injecting knowledge in data-driven vehicle trajectory predictors," *Transportation research part C: emerging technologies*, vol. 128, p. 103010, 2021.

[7] H. Ren, B. Xu *et al.*, "Time-series anomaly detection service at microsoft," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 3009–3017.

[8] K. Shaukat, T. M. Alam *et al.*, "A review of time-series anomaly detection techniques: A step to future perspectives," in *Future of Information and Communication Conference*. Springer, 2021, pp. 865–877.

[9] T. Pelkonen, S. Franklin, and other, "Gorilla: A fast, scalable, in-memory time series database," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.

[10] X. Shi, Z. Feng *et al.*, "Byteseries: an in-memory time series database for large-scale monitoring systems," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 60–73.

[11] C. Adams, L. Alonso *et al.*, "Monarch: Google's planet-scale in-memory time series database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3181–3194, 2020.

[12] InfluxData, "Influxdb," Nov. 2022. [Online]. Available: https://www.influxdata.com/products/

[13] P. Liakos, K. Papakonstantinopoulou, and Y. Kotidis, "Chimp: efficient lossless floating point compression for time series databases," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 3058–3070, 2022.

[14] R. Li, Z. Li *et al.*, "Elf: Erasing-based lossless floating-point compression," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1763–1776, 2023.

[15] S. Chandak, K. Tatwawadi *et al.*, "Lfzip: Lossy compression of multivariate floating-point time series data via improved prediction," in *2020 Data Compression Conference (DCC)*. IEEE, 2020, pp. 342–351.

[16] X. Liang, S. Di *et al.*, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 438–447.

[17] K. Zhao, S. Di *et al.*, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1643–1654.

[18] X. Liang, K. Zhao *et al.*, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, 2022.

[19] Y. Zheng, Q. Li *et al.*, "Understanding mobility based on gps data," in *Proceedings of the 10th international conference on Ubiquitous computing*, 2008, pp. 312–321.

[20] Y. Zheng, L. Zhang *et al.*, "Mining interesting locations and travel sequences from gps trajectories," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 791–800.

[21] Y. Zheng, X. Xie *et al.*, "Geolife: A collaborative social networking service among user, location and trajectory." *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 32–39, 2010.

[22] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, 1964.