Two-Level Data Compression using Machine Learning in Time Series Database

Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, Yue Xie
Alibaba Group
{xinyang.yu, y.peng, lifeifei, sh.wang, leyu.sxw, huijun.mhj, jimmy.xy}@alibaba-inc.com

***Abstract*—The explosion of time series advances the development of time series databases. To reduce storage overhead in these systems, data compression is widely adopted. Most existing compression algorithms utilize the overall characteristics of the entire time series to achieve high compression ratio, but ignore local contexts around individual points. In this way, they are effective for certain data patterns, and may suffer inherent pattern changes in real-world time series. It is therefore strongly desired to have a compression method that can always achieve high compression ratio in the existence of pattern diversity.**

In this paper, we propose a two-level compression model that selects a proper compression scheme for each individual point, so that diverse patterns can be captured at a fine granularity. Based on this model, we design and implement AMMMO framework, where a set of control parameters is defined to distill and categorize data patterns. At the top level, we evaluate each sub-sequence to fill in these parameters, generating a set of compression scheme candidates (i.e., *major mode selection*). At the bottom level, we choose the best scheme from these candidates for each data point respectively (i.e., *sub-mode selection*). To effectively handle diverse data patterns, we introduce a reinforcement learning based approach to learn parameter values automatically. Our experimental evaluation shows that our approach improves compression ratio by up to 120% (with an average of 50%), compared to other time-series compression methods.

I. Introduction

Nowadays, time series data is being continuously generated from a wide range of applications, such as finance [5], Internet services [23] and Internet of things [12]. Such increasing demand for storing and processing time series data facilitates the emergence and development of time series databases (TSDBs), such as OpenTSDB [21] and InfluxDB [9]. One of the key component in these databases is an effective compression scheme that reduces storage footprint, leading to two performance boosts. First, more records can reside in the memory cache for fast access. Second, lower data transfer cost is required between devices (e.g., RAM to disk, and to FPGA or GPU if they are used to speedup processing).

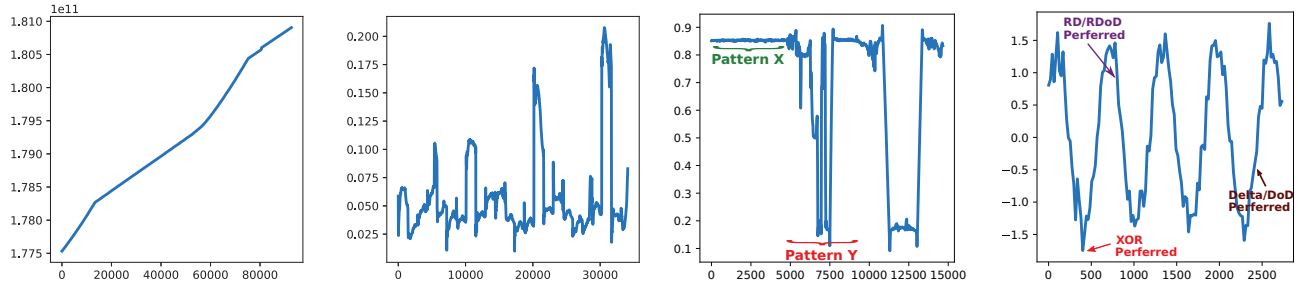
There are plenty of compression schemes to use off the shelf [1], [3], [15], [22]–[25]. However, the effectiveness of a scheme largely depends on its input, where each is only effective on a certain types of inputs (i.e., patterns). Hence in a TSDB, especially a cloud TSDB service that manages data from various sources, it is difficult to rely on a pre-selected compression scheme to sustain high compression ratio for any data. Figure 1 illustrates four examples of real-world time-series data. As shown, these data are distinct from each other, making a single compression scheme difficult to achieve satisfactory compression ratio on all of them. Ideally, we hope that a TSDB could automatically search for the scheme (as well as its parameters) that achieves high compression ratio for a given piece of data. We refer to this problem as *adaptive compression scheme selection*.

This problem gets more challenging in real worlds, since time series data in many applications are awfully complex: even within one time series, the pattern can change over time (Figure 1c); and different parts from a single piece of data may prefer different schemes (Figure 1d). Therefore, in order to achieve good compression ratio, we cannot simply apply one compression scheme for the entire data piece. Instead, we have to examine the data in a fine granularity, and decide the appropriate scheme for sub-pieces of the original data. In the extreme case, we could break down the data into individual points and find the optimal compression scheme for each data point. However, this is almost impractical since both the search space for the optimal scheme and the number of points contained in a TSDB are huge, incurring prohibitive computation cost if we perform the search for each point.

In this paper, we work towards enabling automatic per-point compression scheme selection to achieve satisfactory compression ratio in real-world applications. Note that the per-point approach (needed as illustrated in Figure 1c and Figure 1d) introduces extra space overhead on meta data compared to a global compression scheme. To balance the trade-off between scheme selection efficiency and compression efficiency, we propose a two-level compression model, which is able to adaptively select proper compression schemes for each point. It is built on a concept called *parameterized scheme space* that abstracts the entire search space as a set of scheme templates and control parameters. At the top level, for each *timeline* (a sequence of continuous points), we construct a *scheme space* (a small search space) by filling in values for control parameters. At the bottom level, the most effective scheme from the space is selected for each point.

However, there remain two challenges in the proposed model. First, how to construct a parameterized scheme space that has the potential to always perform well against unpredictable patterns. It is desired that the entire search space is small but covers most cases. Second, how to fill in values of control parameters for a chosen timeline. It is hard to distill hidden patterns from timelines and translate them into parameter values, both effectively and efficiently. For the first challenge, we design and develop AMMMO

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00119



(a) Finance (b) Internet monitoring (c) Resource utilization (d) Sensing device

Fig. 1: Four use cases from real scenario; different use cases have different patterns. Fig 1c shows an example that different periods from data could have different patterns; Fig 1d illustrates the preferred compression scheme at different parts of data.

(Adaptive Multi-Model Middle-Out) framework. In particular, it divides the entire search space into four sub-spaces, called *major modes*, designed with insightful preferences (i.e., transform/bitmask/offset-preferred or mixed in Section IV-B). Each major mode further contains four *sub-modes*, which are constructed compression schemes. For the second challenge, we propose both rule-based and learning-based approaches to address it. In particular, for learning-based approach, since labels (i.e., ground truths) for optimal parameter settings are unavailable, we apply reinforcement learning on a neural network structure to learn parameters interactively. When the compression ratio drops due to pattern changes, this network can be easily retrained.

In summary, we make following major contributions:

- We introduce a two-level model to select compression schemes for each individual point, which addresses inherent data diversity in time series. A parameterized scheme space is proposed to facilitate the representation of the entire search space.
- We design a compression framework AMMMO, following the two-level model. This framework encompasses four major modes to categorize typical patterns, and defines sophisticated control parameters to help construct sub-modes, i.e., compression schemes.
- We design a neural network structure with reinforcement learning to tune parameter values automatically. It solves the issue that no labeled training samples available in the context of data compression.
- We conduct extensive experimental evaluations using large-scale real datasets. The results show that our approach improves compression ratio up to 120% (with an average improvement 50%), compared to other time-series compression methods.

The rest of the paper is organized as follows. We first discuss background and motivation in Section II. We then introduce the two-level scheme selection model in Section III, present the AMMMO framework in Section IV, followed by reinforcement-learning-based model selection in Section V. We evaluate our approach in Section VI and conclude in Section VII.

II. Preliminaries

A. Data Compression

The data compression can be regarded as a process to transform a byte sequence in some representation (e.g., floating numbers for metric value) into a new byte sequence that contains the same information but with less bytes. As surveyed in [16], there are many universal compression techniques, such as (static or adaptive) Huffman and arithmetic coding, which are ubiquitous in real-world applications. Besides, there are also compression algorithms tailored for certain data types. For example, HEVC [30] and H264 [10] are designed for videos, while gzip, 7zip and snappy are designed for texts.

Based on different byte alignment strategies for compressed bit streams, compression algorithms can be categorized into *byte-level* and *bit-level* [29]. In general, the bit-level compression achieves higher compression ratio, while the byte-level compression avoids costly bit shift operations, and is more suitable for partition and parallel processing. In this work, we will focus on byte-level compression since it has higher throughput and is more hardware-friendly.

B. Time Series Data

As defined in [21], [24], a time series consists of many data points, each of which is uniquely identified with a timestamp, a metric name, a measured value, and a set of tags. A simple example is shown in Table I. The combination of tags allows users to record different data properties and issue queries intuitively. Among all fields, timestamps and measured values dominate the storage consumption. Therefore, major task is to compression values in these two fields effectively.

Although general-purposed compression algorithms can be directly applied to time series data, these data have their unique characteristics (detailed in Section III-A) that could potentially help us to tailor more effective compression scheme for them.

C. Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning concerned with how to take appropriate actions to maximize reward in a particular situation. It could be useful for training without ground truth [32].

Reinforcement learning has been widely applied in many interactive scenarios, such as game playing [7], [28] and

TABLE I: An example of raw time-series data on CPU load requests.

timestamp	metric name	value	tagA	tagB
1386806400	cpu.load	0.07	node=alpha	type=system
1386806402	cpu.load	0.13	node=alpha	type=user
1386806403	cpu.load	0.05	node=beta	type=user
1386806405	cpu.load	0.48	node=beta	type=user

resource arrangement [18], [19]. In this work, we explore machine learning based compression methods for TSDBs. Since it is hard to obtain ground truth (i.e. optimal compression scheme and configuration), reinforcement learning becomes a promising solution to facilitate our learning process.

D. Related Work

There are several compression algorithms designed for time-series data in the literature. [3], [15] present lossy compression methods running on sensors over a time series stream. Gorilla [23] adopts a lossless algorithm that first conducts delta-of-delta operations on timestamps and xor operations on measured metric values in server side. However, its bit-level compression model is not friendly to heterogeneous computing, such as GPU and FPGA, which is prevalent for computation acceleration. In the parallel middle out algorithm [25], a byte-level compression algorithm is proposed achieving higher throughput. It adopts only one mode for contiguous 8 data points hence achieves lower compression ratio. Similar, [26] develops a SIMD based algorithm for integer data, achieves higher performance while its compression ratio improvement is minor. Sprintz [1] presents a time series compression for the Internet of Things. Its focus is on 8/16 bit integers and is not optimized for floats. PBE [22] utilizes dynamic programming to achieve optimal compression scheme for monotonically increasing time series data, but it cannot handle general time series data with ups and downs. Compression planner [24] presents an interesting idea that constructs and selects compression plan dynamically on GPU from various compression tools like Scale, Delta, Dictionary, Huffman, Run Length and Patched Constant. However, since it tries to compress all the time-series data several times with various tools and then pick the best one, this is not suitable for high-throughput scenarios. ModelarDB [11] develops a model-based algorithm which segments data into 2 major static models, Gorilla and constant function model which performs well in regular time series and lossy mode. In data mining area, lossy compression for time-series data is widely adopted [13], [17]. However, these methods are unacceptable for a commercial-level TSDB production system.

To support commercial systems for various applications, one of our objectives is to design a lossless byte-level compression framework for time series data that can achieve both high compression ratio and high throughput.

III. Two-level compression scheme selection

A. Time-Series Data Characteristics

Based on our observations from production environments, we summarize several inherent characteristics in time-series data, which play a crucial role in guiding the design of an effective and practical data compression algorithm:

- **Time Correlation**. A time series contains data points continuously collected over a time period. These points are usually sampled with a pre-defined interval (e.g. per second). This leads to two consequences: consecutive timestamp values advance at a relatively fixed rate; and consecutive metric values are always close to each other. These observations should be utilized by compression schemes for good compression ratio.
- **Pattern Diversity**. Time series data can be generated by diverse applications and domains, e.g., finance, Internet, and IoT, from which data patterns may vary dramatically. Furthermore, patterns of a time series may vary over time due to circumstance changes. A compression strategy designed for certain patterns often performs poorly on others, which must be carefully handled in a practical compression solution.
- **Data Massiveness**. As data points are often generated at high rate in time series applications, the underlying storage engine requires both high write throughput (e.g., tens of millions operations per second) and fast processing capacity (e.g., tens of petabytes per day). It is therefore essential to have a compression component that not only helps reduce data volume, but also completes (de-)compression tasks fast.

B. Model Formalization

According to data compression fundamentals [16], a data compression process consists of two stages: a *transform* stage that transform data from one space to another which is more regular; and a *differential coding* that use various coding methods to represent the differential value after transform. There are many primitives available in both stages, e.g., Delta, Scaling, Prediction, Dictionary for transform (IDCT, intra-prediction etc in H264 [10]), and Huffman coding, Arithmetic coding, Run-length differential coding (e.g. VLC, CABAC in H264). By selecting different primitives in each stage, along with corresponding parameters, we can derive a large number of compression schemes (Section IV-A) for time-series data.

Due to the pattern diversity issue discussed above, a single manually-selected compression scheme cannot sustain satisfactory compression ratio for all scenarios. At the other extreme, it is also prohibitive to enumerate all possible schemes for each individual point to obtain the local-best compression ratio. Hence, to ensure that pattern diversity can be gracefully handled with acceptable overheads, we propose two-level time-series compression scheme selection model. At the top level, shared contextual patterns within a consecutive point sequence (i.e., timeline) are detected in order to limit the search space of potentially effective compression scheme candidates

Algorithm 1: Basic Two-Level Compression

```
Input: a time series ts, scheme spaces sps
while ts.empty() == false do
    /* read next timeline from ts                    */
    timeline = ts.readNext();
    /* select scheme space for timeline              */
    space = sps.select(timeline);
    for each point p in timeline do
        /* select scheme for the point               */
        s = space.select(p);
        compress p using scheme s;
```

(i.e., space selection). Here, we say points within a sequence share a 'contextual pattern' if they potentially have good compression ratio under the same set of compression scheme candidates, e.g., the majority of delta-of-delta difference can be represented by 2-byte-value. At the bottom level, by searching over these scheme candidates, we are able to find an effective scheme for each individual point (i.e., scheme selection). Since the top-level phase is conducted once for a timeline, a heavy but effective algorithm is affordable. However, for the bottom-level phase, the efficiency and efficacy are both critical.

In the context of this paper, a *timestamp* is represented as a 64-bit integer, and a *measured metric value* is a 64-bit floating number. In our data compression framework, all data points in a time series with the same metric and tags are modeled as a sequence of ⟨timestamp, value⟩ pairs. We denote the i-th point in the sequence as $\langle t_i, v_i \rangle$. Note that a time series is compressed separately as two sequences $T = \{t_1, t_2, ..., t_n\}$ and $V = \{v_1, v_2, ..., v_n\}$.

1) Scheme Selection Modeling: In this section, we explain how the scheme selection problem is modeled as a two-level procedure. Suppose that we have a collection of transform primitives P_{trans} (e.g., xor of two consecutive values $\{v_i, v_{i-1}\}$, delta-of-delta of $\{v_i, v_{i-1}, v_{i-2}\}$) and a collection of differential coding primitives P_{code} (e.g., 6-bit bitmask coding, 1-byte offset coding). We define several important notions as follows:

Definition 3.1 (Compression Scheme): A compression scheme is the basic unit to compress a single value. It is represented as a tuple $s = \langle a, b, \lambda_a, \lambda_b \rangle$, where $a \in P_{trans}$, $b \in P_{code}$ and λ_a, λ_b are parameters associated to a, b respectively[1]. The scheme contains all information required to compress a value.

Definition 3.2 (Scheme Space): A scheme space S contains a collection of compression schemes $S = \{s_1, s_2, ..., s_n\}$. Each data point belongs to a single scheme space S_i, and is compressed by a specific scheme $s_j \in S_i$.

Given a time series and a set of scheme spaces, we propose the *basic* scheme selection model for each point, as shown in Algorithm 1. For each *timeline* (a consecutive point sequence), a scheme space is selected. Then it is easy to find the most effective scheme for each point, by simply compressing the point using different schemes and keeping the best one.

[1]We use one parameter per scheme for simplicity. Each scheme can have multiple parameters instead.

However, from this simplified model, some critical challenges are not addressed yet:

- How to have a concise scheme space representation, instead of explicit member list? A concise representation facilitates the automatic space selection without human interaction. (Section III-B2)
- How to construct scheme spaces that has the potential to perform well against diverse patterns? It is desired that the entire search space is small in size but covers most frequent patterns. (Section IV)
- How to choose the most suitable scheme space among all candidates? It is challenging to distill hidden patterns from time series and associate them to scheme spaces effectively and efficiently. (Section V)

2) Parameterized Scheme Space: In order to support fast space selection, the number of space candidates must be kept small, which inherently declines the capability of handling different patterns. Instead of having pre-defined static spaces, it is therefore demanded that these candidates can keep adapting themselves to pattern changes, e.g., by leveraging machine learning techniques. However, the member list representation used in Definition 3.2 treats each compression scheme as an independent element, and hence is difficult to be learned by machine learning tasks. Alternatively, we propose a concise representation called *parameterized scheme space*, which divides a scheme space into two parts: a static *scheme template* set and a configurable *variable* set.

Definition 3.3 (Scheme Template and Variables): A scheme template is an extension to a specific compression scheme. It replaces one or more elements in $\langle a, b, \lambda_a, \lambda_b \rangle$ with variables, which are determined at runtime. A variable can represent either a scheme $(i.e., a, b)$ or the parameter associated to a scheme $(i.e., \lambda_a, \lambda_b)$

For example, $\langle X, \texttt{offset}, \lambda_X, \lambda_{\texttt{offset}} \rangle$ with variable X means that the coding scheme is offset coding and the transform scheme is determined by the value of X. Similarly, $\langle \texttt{xor}, \texttt{bitmask}, \lambda_{\texttt{xor}}, Y \rangle$ with variable Y means that xor transform and bitmask coding are used, and the parameter (e.g., byte shift size) of bitmask coding is determined by the value of Y.

Definition 3.4 (Parameterized Scheme Space): A parameterized scheme space contains a collection of scheme templates and a collection of associated variables. A variable can present in multiple scheme templates, which introduces correlations between different schemes within the space.

With the concept of parameterized scheme space, we can transform the task of finding scheme spaces into the task of determining variable values. These values can be adjusted dynamically to catch up pattern changes.

IV. AMMMO Compression Framework

The two-level scheme selection model provides a general-purpose abstraction for data compression. Based on this model, we design AMMMO (Adaptive Multi-Model Middle-Out) compression framework, which is a specific implementation tailored for time-series data. In this section, we discuss key design choices made in AMMMO, including scheme primitives,

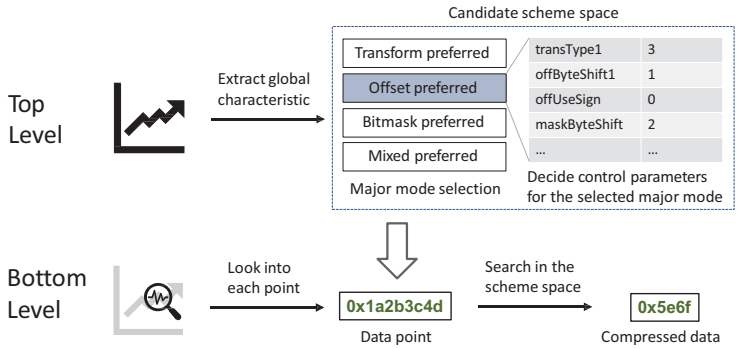


Fig. 2: The two-level compression framework AMMMO for compressing time-series data.

TABLE II: List of differential coding primitives. 0x and 0b prefixes stand for hexadecimal and binary formats.

Primitive Name	Parameter	Format	Example		
			Input	Param	Output
`Offset`	`offByteShift`	1 byte: 1-bit control bits || 7-bit value 2 bytes: 2-bit control bits || 14-bit value 3 bytes: 3-bit control bits || 21-bit value	0x 00 17 00 00 00	3	control bits || 0b10111
`Bitmask`	`maskByteShift`	1 value: (up to 6-bit) bitmask || value 2 values: 6-bit bitmask || value1 || value2	0x 10 00 26 84 00	1	0b1011 || 0x 10 26 84
`Trailing-zero`	`N/A`	2-bit (or 3-bit) trailing-zero control bits || 3-bit non-zero control bits || value	0x 00 14 04 09 00	`N/A`	1 (0b01) || 3 (0b010) || 0x 14 04 09

TABLE III: List of transform primitives.

Id	Primitive Name	Operation Description
0	`delta`	$v_i - v_{i-1}$
1	`reversed delta (RD)`	$v_{i-1} - v_i$
2	`xor`	v_i xor v_{i-1}
3	`delta-of-delta (DoD)`	$(v_i - v_{i-1}) - (v_{i-1} - v_{i-2})$
4	`reversed delta-of-delta (RDoD)`	$(v_{i-1} - v_{i-2}) - (v_i - v_{i-1})$
5	`delta xor (DX)`	$(v_i - v_{i-1})$ xor $(v_{i-1} - v_{i-2})$

scheme spaces (i.e., control parameters, major modes and sub-modes), and an intuitive rule-based method for scheme space selection. These choices of primitives and parameter settings are empirical and have been proven to be effective in practice. However, they are orthogonal to our model and can be replaced or extended arbitrarily. An overview architecture of AMMMO framework is shown in Figure 2.

A. Compression Scheme Primitives

Recall that a compression scheme consists of a transform stage and a differential coding stage (or called encoding stage). In AMMMO, we use a set of primitives to build these stages.

Transform primitives. At the transform stage, for the i-th data point, we apply transform operations to its value v_i, which additionally takes previous two points v_{i-1}, v_{i-2} as input. Table III lists six transform primitives used in our framework, which are proven effective from our empirical evaluations in production environment. Note that those transform primitives suffering low throughput (e.g., dictionary) are excluded from our implementation.

Encoding primitives. The transform stage converts each 8-byte raw value into a 8-byte differential value. At the encoding stage, this value is to be encoded using less bytes. Table II lists three encoding primitives designed in our framework, and we detail them as follows:

- *Offset coding* uses a global parameter `offByteShift` to indicate from which byte the non-zero value starts. In our implementation, three coding lengths (1/2/3 bytes) are supported, each with own `offByteShift` value.
- *Bitmask coding* has a bitmask (with up to 6 bits) to indicate whether a byte has non-zero values. To handle the case that only high-order bytes have non-zero value, a global parameter `maskByteShift` is defined to remove low-order bytes from the mask. To further reduce the number of control bits, two consecutive values can share one bitmask, halving the amortized bits per value.
- *Trailing-zero coding* separates the 8-byte value into three parts: trailing-zero bytes, non-zero bytes and leading-zero bytes. In this format, 2 or 3 bits (trailing-zero control bits) are able to indicate up to 3 or 7 bytes of trailing zeros; and 3 bits (non-zero control bits) representing range $[1, 8]$ instead of $[0, 7]$, indicate the number of non-zero bytes (up to 8) to be coded after trailing zeros are removed.

B. Compression Scheme Space Definition

AMMMO applies different scheme selection strategies to metric values and timestamp values. For metric values, the two-level selection model is used, which determines a major mode (i.e., scheme space) followed by a sub-mode (i.e., scheme), as shown in Figure 2. Timestamp values only use bottom level (sub-mode) selection model as their patterns are more regularized.

Based on compression scheme Definition 3.1 and primitives design IV-A, AMMMO constructs nine (first four specify a and b, and last five specify λ_a and λ_b) control parameters as listed in Table IV (defining possible compression scheme spaces), which major mode (each with four sub-modes) is further defined in Table V.

TABLE IV: The control parameters in AMMMO which defines a compression scheme space in top level.

Parameter	Bits	Range	Description
majorMode	2	[0, 3]	Indicate the selected major mode
transType1	3	[0, 5]	Refer to a transform in Table III
transType2	3	[0, 5]	Similar to transType1
transType3	3	[0, 5]	Similar to transType1
offByteShift1	3	[0, 7]	Byte offset for 1-byte offset coding: offByteShift = offByteShift1
offByteShift2	1	[0, 1]	Byte offset for 2-byte offset coding: offByteShift = offByteShift1 - offByteShift2
offByteShift3	1	[0, 1]	Byte offset for 3-byte offset coding: offByteShift = offByteShift1 - offByteShift2 - offByteShift3
offUseSign	1	[0, 1]	Indicate if offset coding use sign
maskByteShift	3	[0, 5]	Byte offset for bitmask coding

TABLE V: The major modes and sub-modes in AMMMO with control parameters.

Id	Major Mode	Control Bits	Sub-Mode Compression Scheme
0	Transform Preferred	0b00	transType1 with 6-bit bitmask
		0b01	transType2 with 6-bit bitmask
		0b10	transType3 with 6-bit bitmask
		0b11	transType1 with 6-bit trailing-zero
1	Bitmask Preferred	0b00	transType1 with 6-bit bitmask
		0b01	transType2 with 2-value bitmask
		0b10	transType3 with 6-bit bitmask
		0b11	transType1 with 6-bit trailing-zero
2	Offset Preferred	0b1	transType2 with 1-byte offset
		0b01	transType2 with 2-byte offset
		0b000	transType2 with 3-byte offset
		0b001	transType1 with 5-bit trailing-zero
3	Mixed	0b1	transType1 with 1-byte offset
		0b01	transType2 with 2-value bitmask
		0b000	transType3 with 5-bit bitmask
		0b001	transType1 with 5-bit trailing-zero

A space (i.e., major mode) contains 4 sub-modes to adaptively compress each point at bottom level. This choice is a balance between the coverage of search spaces and the overhead of meta data and computation. Note that the definitions of the major modes and sub-modes are based on experimental results and are further extended or replaced when dealing with different workloads. In our current design, a timeline requires 20 bits (Table IV) of meta data for top-level mode selection, and an average of 2 bits (Table V) for bottom-level mode selection.

C. Compression Procedure

In AMMMO, 32 data points are processed as a block with header and data segment, and timestamps and metrics value are compressed separately:

Timestamp compression. Timestamps in a timeline usually have fixed intervals, such as 1 second. This pattern can be easily captured by the delta-of-delta transform operation. However, it is common that some points violate this pattern, e.g., due to sampling jitter or network delay. Inspired by Gorilla [23], we take advantage of the delta-of-delta operation, and try to align the irregular data in bytes to speed up. To simplify transform process, we consider the byte representation of each point as a 64-bit integer. Detailed compression procedure is shown in Algorithm 2. We maintain two data segments, i.e., *header* for delta-of-delta outputs and *data* for irregular data

Algorithm 2: Compression for Timestamp Values

```
   Input: timestamp sequence ts
   Output: compressed bitStream bs
1  bitStream header, data;
   /* write first two points (8 bytes each)                    */
2  data.append(ts[0], ts[1]);
3  for each point ts[i] where i ∈ [2, ts.length) do
4      d = (ts[i] − ts[i-1]) − (ts[i-1] − ts[i-2]);
5      if d == 0 then header.appendBit(0) ;
6      else
7          header.appendBit(1);
8          if d > 0 then d = d − 1;
9          if d ∈ [−4, 4) then
10             data.append(1b'1 —— 3b'd); // 4 bits
11         else if d ∈ [−32, 32) then
12             data.append(2b'01 —— 6b'd); // 8 bits
13         else
14             if bitmask coding is preferred then
15                 data.append(3b'001 —— 5b'bitmask);
16                 data.append(bitmask-encoded d);
17             else
18                 data.append(3b'000 —— 5b'control-bit);
19                 data.append(zero-trailing-encoded d);
   /* final output concatenates header and data                */
20 return bs.write(header, data)
```

Algorithm 3: Compression for Metric Values

```
   Input: metric sequence ms
   Output: compressed bitStream bs
1  bitStream header, data;
2  data.append(ms[0]);
3  for each point ms[i] where i ∈ [1, ms.length) do
4      d = ms[i] − ms[i-1];
5      if d == 0 then header.appendBit(0) ;
6      else
7          header.appendBit(1);
8          for each mode m from sub-modes do
9              calculate compression ratio r for d;
10             if r ¿ best_ratio then
11                 best_ratio = r;
12                 best_mode = m;
13         data.append(best_mode.compress(d));
14 return bs.write(header, data)
```

points. The first two points of a timeline are stored in *data* in their raw formats (line 2). For subsequent points, we calculate their delta-of-delta values and indicate in *header* (line 3-7). When irregular value occurs (i.e., delta-of-delta is not zero), they are encoded in different lengths (e.g., 4, 8, 16 bits) and appended to *data* segment (line 8-19) [2].

Metric value compression. The compression for metric values is similar to timestamps, as shown in Algorithm 3. Here, deltas of consecutive metric values are used as indicators (line 3-7), instead of delta-of-deltas. For each point different from its predecessor (i.e., $d \neq 0$), we try all 4 sub-modes to find the best compression scheme for d. For each sub-mode, we construct the corresponding scheme to encode d and calculate compression ratio (line 8-12). We then choose the sub-mode with the best ratio and append compressed d into the $data$ segment (line 13).

D. Rule-based Scheme Space Selection Algorithm

The remaining challenge is to select proper compression scheme space, i.e., fill in values for all parameters in Table IV. A rule-based algorithm (Algorithm 4) invokes metric *benefit*

[2]Since $d = 0$ is unoccupied, we utilize it to cover a larger range (line 8).

Algorithm 4: Rule-based Scheme Space Selection

Input: metric value sequence ms
Output: control parameter values $params$ in Table IV
/* PART I: calculate the $benefit\_score$ */
/* $benefit\_score$: the total number of bytes can be saved against the worst case 9-byte original representation per point (i.e. 1 byte of control bits and 8 bytes of differential value) for different compress schemes in a timeline */
1 a 6 × 6 array with zero initializations: $benefit\_score$;
2 **for** each point $ms[i]$ where $i \in [1, ms.\text{length})$ **do**
3 **for** each transform mode $tm[j]$ in Table III **do**
4 **for** each coding format $cf[k]$ in Table II **do**
5 $benefit$ = calculateBenefit($ms[i]$, $tm[j]$, $cf[k]$);
6 $benefit\_score[j][k]$ += $benefit$;
7
/* PART II: calculate the $params$ based on $benefit\_score$ */
/* $best\_majorMode\_score$ represents the best score among 4 major modes */
8 $best\_majorMode\_score$ = 0;
9 **for** each majorMode $mm[i]$ where $i \in [0, mm.\text{length})$ **do**
10 $majorMode\_score$ = 0;
/* $best\_subMode\_score$ represents the best score among 4 sub modes of the majorMode $mm[i]$ */
11 $best\_subMode\_score$ = 0;
12 **for** each subMode $sm[j]$ where $j \in [0, sm.\text{length})$ **do**
/* find the array indexes in $benefit\_score$ that match $mm[i]$ and $sm[j]$, say s and t */
13 s, t = findIndex($mm[i]$, $sm[j]$);
14 **if** $benefit\_score[s][t] > best\_subMode\_score$ **then**
15 $best\_subMode\_score = benefit\_score[s][t]$;
16 $majorMode\_score$ += $best\_submode\_score$;
17 **if** $majorMode\_score > best\_majorMode\_score$ **then**
18 $params$ = findParams($mm[i]$);
19
20 **return** $params$

score to represent a compression scheme's efficiency, and the whole process is separated into two parts generally. Firstly, *benefit score* for all the possible submodes are constructed (line 2-6). Secondly, the best major mode is selected based on the benefit scores, and then fill control parameter values accordingly (line 8-18).

However, there are issues that limit its efficiency and are difficult to resolve, for example: whether *benefit score* is a good enough metric? Moreover, when the definitions of Table II ˜ V need to be adjusted (e.g., to tune compression performance), such an analysis process has to be manually re-designed and implemented, which is complex and time-consuming. Therefore, it is appealing to have an efficient, automatic and adaptive method to tune these parameters, which motivates us to adopt machine learning approaches.

V. Machine Learning for Mode Selection

From Table IV, we have $2^3 \cdot 4 \cdot 6^4 \cdot 8 = 331,776$ possible combinations for control parameters in total. It is prohibitive to enumerate all cases to find the best *control setting*. Alternatively, getting the appropriate control setting can be formulated as a classification problem, which supervised machine learning approaches are good at solving. However, it is infeasible in our context to generate ground truths for training since:

- Suppose a training sample has 32 points (256B), there will be 256^{256} possible cases in theory. And for each sample, we need random 331,776 times to find the best compression ratio and take it as label. The sample organization and computation consumption to create dataset is prohibitive.
- Supervised learning requires the label to be only one or a few, while given a sample, it's likely that multiple control settings are able to achieve the same compression ratio, which leads to multiple labels associated with a sample.
- Once the primitive and parameter definition is refined, the original training set becomes useless.

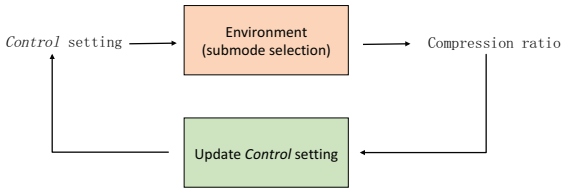


Fig. 3: An overview of policy gradient RL based control setting generation model.

Hence, we propose a framework that adopts reinforcement learning to tune control parameters automatically in two steps:

1) *Learn control settings from blocks.* Each timeline is further divided into fixed-size blocks (e.g., 32 points per block in our implementation). A neural network (Section V-B) is trained via reinforcement learning (Section V-A) to interactively learn proper settings of control parameters for each block .
2) *Determine the control setting of a timeline.* The trained network is used to generate control settings for all blocks. After that, the best setting for the entire timeline is determined following a statistical strategy (Section V-C).

A. Training with Reinforcement Learning

Figure 3 illustrates the overview of our learning flow. In particular, a neural network takes a block as a state, generates a control setting as the action. Under this setting, the submode selection module computes the compression ratio as the feedback to adjust the neural network interactively.

For a given block, its control setting (or called control option) is defined as $cop = \{ op_0, op_1, ..., op_8\}$, where op_i belongs to the value range of i-th control parameter listed in Table IV. For example, op_0 is `majorMode` and its value belongs to range [0, 3]. Let $r_i(cop)$ represent the compression ratio of the i-th block with control setting *cop*. The objective of the control setting generation network is to find the *cop* that maximizes the compression ratio:

$$J(\theta) = E_{x \sim p(x|\theta)}[f(x)] \quad (1)$$

where: x stands for a control setting; θ is the policy defined by the network (Section V-B) that generates the *cop*; $p(x/\theta)$ is the probability to generate control setting x; and $f(x)$ represents the reward to the RL network, i.e., compression ratio $r_i(x)$. The network aims to maximize the expected reward of control setting decision x under probability distribution $p(x/\theta)$.

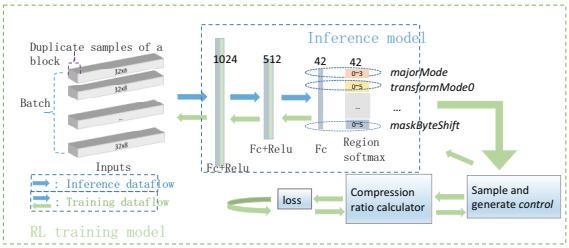


Fig. 4: Neural network architecture.

Decisions are the control option cop_i drawn from the softmax distribution in θ.

The weights in the network are learned using Adam [14] optimizer based on the policy gradients computed via the REINFORCE equation [32]:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta E_x[f(x)] \\ &= E_x[f(x)\nabla_\theta logp(x)] \\ &\approx \frac{1}{N}\sum_{i=1}^{N} f(cop_i)\nabla_\theta logp(cop_i)\end{aligned} \tag{2}$$

We sample N control settings to estimate ∇_θ as specified in Formula 2. It implies that the network is updated towards the direction of better compression reward $f(cop_i)$.

For N different sampled settings of a block, we should activate the network to generate higher possibility for those *cop*s with high rewards. To achieve this goal, block-level normalized compression reward $f_n(x)$ and summarized cross entropy $Hcs(x)$ are applied:

$$f_n(x) = (f(x) - mean(f(x)))/std(f(x)) \tag{3}$$

$$Hc(x) = -(p(x)logp(x) + (1-p(x))log(1-p(x))) \tag{4}$$

$$Hcs(x) = \sum_{i=x:op0}^{x:op8} Hc(i) \tag{5}$$

Consider that the SGD scheme uses M blocks as a batch, each with N samples, the final loss function to be minimized is:

$$\frac{1}{M*N}\sum_{i=1}^{M*N}(f_n(cop_i) * Hcs(cop_i)) - \lambda * H(cop) \tag{6}$$

where $H(cop)$ is the average entropy value of all *cop* used as entropy regularization that avoids network converge to local optimum [20]. We use regularization parameter $\lambda = 0.01$ to leverage it.

B. Neural Network Architecture

Figure 4 shows the neural network structure and overall data flow of training and inference. In the RL training stage (module), a batch of total M blocks, each of which has N duplicate samples, are fed into the network. First, the blocks pass through the network and generate $M \cdot N$ control setting candidates. Second, for each field in a control setting, we sample with associated probabilities to determine its value. Third, compression ratios under each control setting are calcuated and then are used to derive the loss according to Formula 6. Last, backward propagation is applied and weights are updated accordingly. In the inference stage (module), either a single block, a batch, or all blocks of a timeline can be fed into the network. We pass through the network and generate control setting for each block. Then for each field in a control setting, we choose the highest probability value among all the blocks as the final setting.

TABLE VI: Neural network layers.

Layer	Type	Size	Activation
1	Fully-connected	1024	ReLU
2	Fully-connected	512	ReLU
3	Fully-connected	42	None
4	Region Softmax	42	None

Table VI lists layer configurations used in the network. The basic input unit is a block, which has the size $32 \cdot 8$ bytes. The fully connection is applied to layer 1, 2 and 3, in which the first two layers have the ReLU activation function. The size of the hidden node in layer 3 is 42, which is the number of all available values in nine control parameters. Each parameter field could make its own decision independent from others. Hence, rather than using the global softmax as in image classification tasks, we introduce a *region softmax* that conducts softmax on each field, instead of using all 42 logits.

C. Determine Timeline Control Setting

A timeline has only one global control setting that takes effect on all blocks. We use statistic strategy to assemble blocks of control setting to one. The procedure is: consider all these parameters as independent from each other, select the most frequent values for each field in control parameters. Note, some other strategies (i.e., select the most frequent combination of control settings, or choose the most frequent value for `majorMode` first, and then select most frequent values for other fields from those blocks belong to this mode) have slight difference in the final result view.

VI. Experiments

To evaluate the proposed AMMMO framework, we test it on datasets collected from our business operations[3], ranging from IoT applications to performance monitoring of large clusters. In addition, a public timeseries repository UCR [4] is also used. We compare our method against other state-of-the-art compression methods in terms of both the compression ratio and efficiency. Also, we evaluate AMMMO framework using various mode selection algorithms, showing the advantages of machine learning based methods.

A. Experiment Setup

Datasets. We collect a representative set of time series datasets that exhibit different patterns, characteristics and sizes. In particular, 28 datasets are selected, where 8 of them are from IoT applications and 20 of them are from

[3]https://github.com/JonyYu/ATimeSeriesDataset.

cluster monitoring. They are of different sizes, from 10k to 500k points[4]. For the purpose of applying machine learning methods, these datasets are further divided into a test set A (for training) and a test set B (for validation), as listed in Table VII[5].

TABLE VII: Datasets with 28 selected time series.

Test Set A					
Name	Points	Name	Points	Name	Points
IoT0	430,737	IoT6	430,413	Server35	147,395
IoT1	429,745	IoT7	313,539	Server41	136,594
IoT2	428,390	Server30	158,188	Server43	29,233
IoT3	344,581	Server31	147,385	Server46	154,585
IoT4	306,736	Server32	165,395	Server47	140,199
IoT5	372,868	Server34	140,194	Server48	157,051

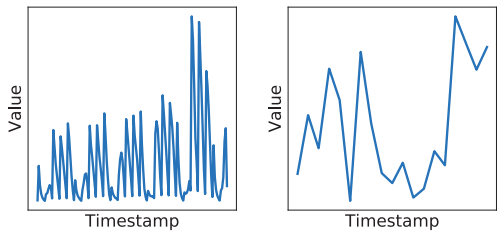
Test Set B			
Name	Points	Name	Points
Server57	26,779	Server94	140,198
Server62	32,569	Server97	158,194
Server66	135,409	Server106	136,478
Server77	136,598	Server109	153,438
Server82	143,798	Server115	165,384

Also, there is a well-known time series datasets named UCR [4], which is a repository of 85 univariate time series datasets from various domains, commonly used for benchmarking time series algorithms. Since most of them are small, and each has several different patterns (pattern id 0, 1, 2, ...). We choose the longest 8 datasets, and concatenate all the pattern 0 to form a single longer time series. We also separate this dataset into test sets A and B in Table VIII.

Figure 5 illustrates two of the datasets. As shown in figure, the data from different datasets exhibit different characteristics. Even in one dataset, the internal data patterns are complex.

TABLE VIII: 8 longest time series datasets in UCR

Test Set A		Test Set B	
Name	Points	Name	Points
HandOutlines	641,796	CinC_ECG_torso	8,190
Haptics	19,638	InlineSkate	16,929
StarLightCurves	155,496	MALLAT	6,138
UWaveGestureLibraryAll	115,168	Phoneme	4,092



(a) Server48 (b) Haptics

Fig. 5: Illustration of two of the datasets.

Methods and Variants. Three open-sourced baseline compression methods and six AMMMO variants are used to evaluate the performance of AMMMO framework and machine learning mode selection.

- **Gorilla** [2]: a state-of-the-art commercial bit-level compression algorithm applied in server side. It applies delta-of-delta on timestamp values followed by a variable length coding, while applies xor on metric values followed by Huffman coding.
- **MO (Middle-Out)** [25]: a byte-level compression algorithm good for parallel processing, which takes 8 points as a process block. It applies xor on timestamp values followed by byte-aligned differential coding, while applies xor on metric values followed by trailing zero coding on the differences.
- **Snappy** [6]: a general-purpose compression algorithm developed by Google. It is byte-level and is used by InfluxDB [9], KairosDB [8], OpenTSDB [21], RocksDB [31], the Hadoop Distributed File System [27] and numerous other projects.
- **AMMMO_Lazy**: AMMMO framework with a fixed control setting, i.e., set `majorMode` to 0, `transTypes` to 2/5/0, and all other parameters to 0.
- **AMMMO_Rnd5000Best**: Control setting is selected from random generations. Generate 5000 times and select the one with the best compression ratio. Although we do not cover all possible settings, it still gives a good approximation for the optimal ratio that AMMMO could achieve. Note that this method is unaffordable in production deployment, and we only use it as a reference point.
- **AMMMO_Rnd1000Best**: Generate random control settings 1000 times, report the best compression ratio.
- **AMMMO_Rnd1000Avg**: Generate random control settings 1000 times, report the average compression ratio.
- **AMMMO_Analyze**: Scheme space selection is done by rule-based analysis algorithm (Section IV-D).
- **AMMMO_ML**: Scheme space selection is done by machine learning (Section V).

B. Compression Ratio

In this section, we evaluate the compression ratio achieved by AMMMO framework on both timestamp and metric values. We also demonstrate the advantages of using machine learning methods.

1) Timestamp Compression: For timestamp values, we use one-level mode selection in AMMMO, and hence all variants have the same behavior. Therefore, we omit their internal comparisons and only provide results for Gorilla, MO and AMMMO.

Figure 6 shows the compression ratios for all 18 time series in test set A. There is no result for Gorilla on server monitoring applications, as it cannot handle the case that the difference between two consecutive timestamps is larger than 2 hours (which exists in all these time series). As can be seen, Gorilla and AMMMO perform much better than MO. It is because that delta-of-delta operations are more suitable for timestamps compared to xor operations. Moreover, AMMMO achieves the best results in all cases, since it uses 3 bits to handle

[4]Each dataset is treated as a single timeline for simplicity. However, in real world, a long time series (with diverse patterns) is always split into multiple timelines to mitigate the pattern diversity within a timeline.

[5]In test set A, 8 out of 18 datasets contain constant metric values over long periods (e.g., with most values being zero), which can be easily compressed. Hence, we only include them in timestamp evaluation.

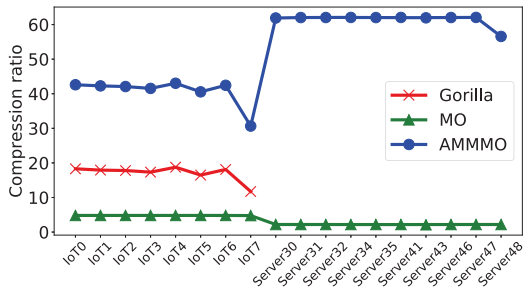
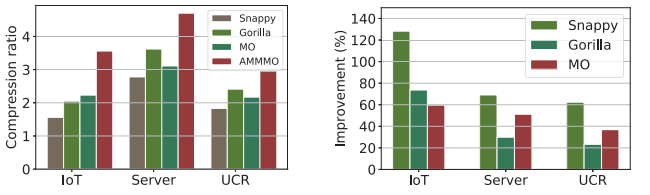


Fig. 6: Compression ratios on timestamp values.



(a) Compression ratio

(b) Improvement of AMMMO over other schemes

Fig. 7: Comparison of compression ratios.

sample jitter pattern and uses adaptive multiple modes (e.g., offset, bitmask, trailing-zero) to handle other patterns. We can observe that the overall compression ratio improvement in AMMMO are significant: an 8-byte raw value only costs 1 to 2 bits in average after compression. Note that Gorilla also requires only 4 bits in average, indicating that timestamp compression is quite effective in most cases.

2) Metric Value Compression with AMMMO: Recall that AMMMO framework enables variants with different mode selection strategies as listed in Section VI-A. To clearly show its overall capacity, we define AMMMO as the best compression ratio achieved by all variants.

We define X *improved compression ratio over* Y as $\frac{(X-Y)}{Y}$. Figure 7a shows the average compression ratio on all tests in IoT/Server/UCR datasets. From the result, Snappy achieves worst performance since it does not consider characteristic of timeseries much; Gorilla is slightly better than MO in general (MO is better in IoT cases since IoT data is usually simpler), because it uses bit level packing operation; AMMMO is significantly better in all these cases. Figure 7b shows the improved compression ratio comparison. The average improved ratio of AMMMO over Snappy, Gorilla, and MO are around 87%, 42% and 49%.

Figure 8 shows the detailed compression ratios on each time series in test set A. For better readability, the x-axis is sorted by Gorilla's compression ratio. As can be seen, Snappy is worst, Gorilla and MO are close, and AMMMO variants significantly outperform others in most cases; an exception is `UWaveGesture`, where snappy achieves very good compression ratio and AMMMO_Rnd5000Best achieves slightly better result. Note that AMMMO_Rnd5000Best is better than AMMMO_Rnd1000Best in most cases except for `IoT5` and `Server43` (due to randomness). This indicates that AMMMO_Rnd5000Best is a proper approximation of AMMMO's

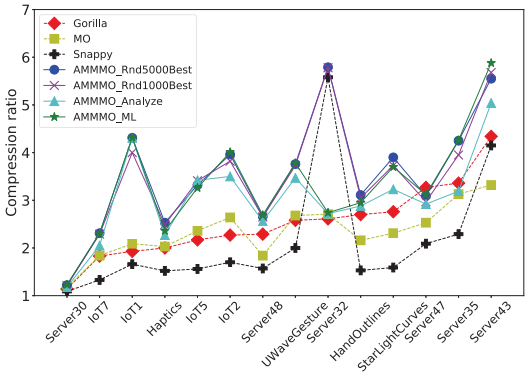


Fig. 8: Comparison of metric value compression ratios.

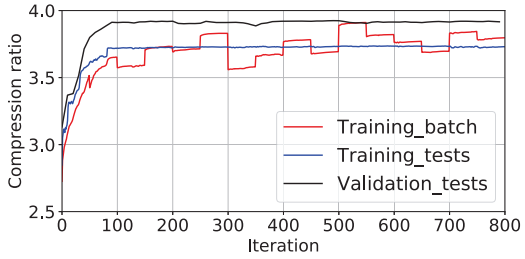


Fig. 9: Training curve of compression ratio on training batch, training tests, and validation tests.

best performance. For practical variants AMMMO_Analyze and AMMMO_ML, machine learning approaches outperform manually-determined rules in most cases. And AMMMO_ML is very close to or even better (`Server43`) than AMMMO_Rnd5000Best, except `UWaveGesture`.

3) AMMMO with Machine Learning: In this part, we focus on the evaluation of whether AMMMO_ML is able to learn proper parameter values from training data, i.e., test set A. Therefore, we compare it with other variants on validation test set B. In the training stage, we set batch size M to 896, block duplicate sample N to 64, and start learning ratio to 0.0005. Figure 9 illustrates the average compression ratio achieved in training batch, training tests and validation tests. Thanks to the effective loss function design, the RL scheme learns how to compress quickly. These ratios increase rapidly in first 100 iterations, and after that the curve becomes stable. Suppose we have a time series with 1024 blocks. According to our findings, we only need to train 100 iterations to obtain acceptable compression ratio, taking around 100 seconds.

AMMMO has a large number of control setting options, and it is critical to have a mode selection strategy that can find a proper one. Figure 10 shows the compression ratios achieved by different mode selection strategies. As can be seen, AMMMO_Rnd1000Avg is the worst, and even AMMMO_Lazy outperforms it. Both AMMMO_ML and AMMMO_Analyze sustain good performance in most cases. Moreover, AMMMO_ML achieves astonishing compression ratio on `Sever115` and `Sever97`. We observe that, AMMMO_ML performs worse than AMMMO_Rnd1000Best on `Sever109`. With a closer look, we observe that block-level

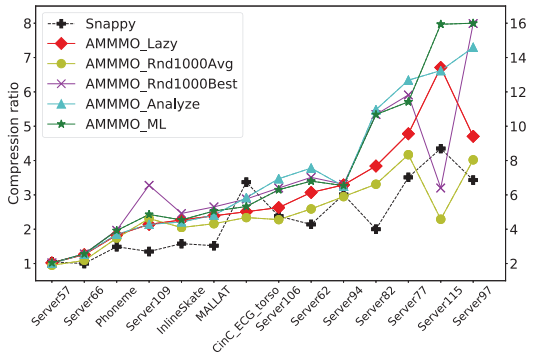
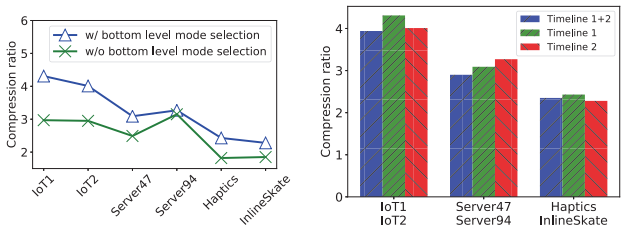


Fig. 10: Metric value compression ratios from different methods. Right vertical axis is for `Server97` and left axis for other datasets.



(a) Bottom level mode selection (b) Compression on split timelines

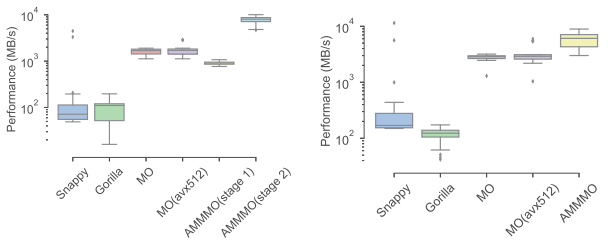
Fig. 11: Comparison of different mode selections.

compression ratios are still good in AMMMO_ML. However, these parameter values are quite diverse, which makes the statistic polices hard to derive a proper global control setting.

In summary, AMMMO_ML is able to learn patterns from training set A and achieve excellent compression ratio on validation set B. This confirms that machine-learning based mode selection is effective, and can help to automate the parameter tuning process. In fact, our initial AMMMO only contains six control parameters and two major modes. This automatic mode selection facilitates the discovery of nine effective parameters and four representative modes as the backbone of our AMMMO framework.

4) Parameter Verification: It is interesting to check whether AMMMO_ML figures out meaningful control setting. Table IX lists the timeline control setting decisions made by different AMMMO variants, on `IoT1`, `IoT2`, `Server35` and `Server48` in set A. The results show that AMMMO_ML is able to find almost the same setting for `maskByteShift` and `offsetByteShift`, which indicates that it understands the effect of data resolution after transform. It also performs well on `majorMode` selection and transform types selection.

Figure 10 shows the importance of top-level mode selection, which indicates the advantages of AMMMO_ML against other variants that do not have mode selection. It is also interesting to look at bottom-level mode selection effect. Figure 11a shows the result when only submode 4 (i.e., similar mode as Gorilla in Table V) is used, the performance is significantly worse as can be expected. For the timeline length, if a timeline contains different patterns in each sub-parts, splitting it into



(a) Compression (b) Decompression

Fig. 12: Compression and decompression performance comparison among different schemes.

shorter timelines is able to take advantage of different scheme spaces. Figure 11b shows that compression on split timelines (i.e., shorter and with less diverse patterns) achieves better compression ratio.

C. Compression Efficiency

Due to the massiveness of time series, the compression and decompression efficiency also plays a crucial role in compression algorithms. Here we provide throughput for both operations achieved by AMMMO, Gorilla, MO and Snappy.

On a CPU+GPU platform[6], the compression and decompression performance of Gorilla, MO and Snappy are illustrated in Figure 12. From the result, we can see that Snappy's performance varies a lot. In most cases, the performance is around 60MB/s (150MB/s). In some rare cases, it can reach up to 4.5GB/s (11.5GB/s), because the compression ratio is less than 1.1 and hence there is no or few compression. Gorilla and MO are much stable, and the average performances are around 100MB/s (100MB/s) and 1.3GB/s (2.9GB/s). In several cases, Gorilla's decompression performances is worse than compression performances. It is because that Gorilla is a bit-level coding, where the whole process has to be serialized and involves frequent bit operations and condition checks during decompression. With the same CPU resource, MO's performance is much better than Gorilla, which is attributed to the simpler algorithm and byte-level coding. AMMMO is also byte-level, and we evaluate its performance on GPU. Both its compression and decompression performances are stable. Compression procedure in AMMMO involves 2 stages[7] , mode selection stage and compression stage (Section IV-C). The performance is around 900MB/s and 7.5GB/s[8] respectively. For decompression, the performance is around 5.6GB/s, which is slightly worse compared to compression performance. It is because compression procedure can leverage higher parallelism during sub-mode selection for many points.

[6]CPU: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz; GPU: Nvidia gp100 with 16 GB video memory; Memory: 512 GB.

[7]The mode selection performance is for learning-based inference. In real deployments, not all timelines need mode selection, since we usually can reuse the last mode for subsequent timelines (due to time correlation). If the inference performs badly, we can retrain the network, which costs about 100s.

[8]This performance includes data copy through PCIe. The compression kernel itself achieves around 30GB/s.

TABLE IX: Control settings selected by different AMMMO variants.

	Algorithm	major Mode	trans Type1	trans Type2	trans Type3	offByte Shift1	offByte Shift2	offByte Shift3	offUse Sign	maskByte Shift
IoT1	Analyze	3	0	5	0	3	0	0	0	1
	RandomBest	3	0	5	3	3	1	1	0	1
	ML	3	0	5	4	3	1	0	0	1
IoT2	Analyze	2	0	2	0	3	1	0	0	0
	RandomBest	2	5	0	3	3	1	0	0	0
	ML	2	0	0	5	3	1	0	0	0
Server35	Analyze	2	0	5	0	5	0	0	0	0
	RandomBest	3	4	2	4	5	0	0	1	5
	ML	3	3	2	5	5	0	0	0	5
Server48	Analyze	2	5	5	0	4	0	0	0	0
	RandomBest	3	0	5	4	6	1	1	0	4
	ML	3	4	5	0	4	0	0	0	4

VII. Conclusion and Future Work

In order to improve compression ratio for time-series data, we propose a two-level compression scheme selection model to fully utilize diverse data characteristics. We design and implement AMMMO framework, a byte-level parallel-oriented compression method outperforming state-of-the-art approaches in terms of compression ratio and efficiency. It achieves 50% better compression ratio compared with Gorilla and MO. In this framework, we introduce a machine learning based method to learn diverse data patterns, and it helps perform mode selection and tune parameters automatically. We have used this automatic method to discover nine control parameters and four major modes, which in turn advises the design of AMMMO internal structures.

As future work, we will explore how to improve AMMMO by adjusting major mode definition, working with major modes, integrating with other compression tools and building blocks (for example support scaling). Another valuable direction is to design another neural network to replace statistic polices in generating timeline control settings. Lastly, it is an interesting open problem to extend the proposed framework to compress other data types.

References

[1] D. W. Blalock, S. Madden, and J. V. Guttag. Sprintz: Time series compression for the internet of things. IMWUT, 2(3):93:1–93:23, 2018.

[2] M. Burman. Time series compression library, based on the facebook's gorilla paper, 2016. https://github.com/burmanm/gorilla-tsc.

[3] H. Chen, J. Li, and P. Mohapatra. Race: time series compression with rate adaptivity and error bound for sensor networks. In 2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems, pages 124–133. IEEE, 2004.

[4] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista. The ucr time series classification archive, 2015.

[5] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In Proceedings of the 1994 ACM SIGMOD international conference on Management of data, SIGMOD '94, pages 419–429. ACM, 1994.

[6] S. Gunderson. Snappy: A fast compressor/decompressor, 2015.

[7] X. Guo, S. P. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In NIPS, pages 3338–3346, 2014.

[8] B. Hawkins. Kairosdb: Fast time series database on cassandra.

[9] InuxDB. Open source time series, metrics, and analytics database, 2015.

[10] ITU-T and I. J. 1. Advanced video coding for generic audiovisual services, 2012. https://www.itu.int/rec/T-REC-H.264.

[11] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Modelardb: Modular modelbased time series management with spark and cassandra. In Proceedings of the VLDB Endowment, volume 11. VLDB, 2018.

[12] S. D. T. Kelly, N. K. Suryadevara, and S. C. Mukhopadhyay. Towards the implementation of iot for environmental condition monitoring in homes. IEEE sensors journal, 13(10):3846–3853, 2013.

[13] E. J. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In KDD, pages 206–215. ACM, 2004.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In ICLR, 2015.

[15] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In Proceedings of the 19th International Conference on Data Engineering, pages 429–440. IEEE, 2003.

[16] D. A. LELEWER and D. S. HIRSCHBERG. Data compression. In ACM Computing Surveys 19, volume 1 of ACM 1987, 1987.

[17] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, pages 2–11. ACM, 2003.

[18] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean. A hierarchical model for device placement. In Proceedings of the International Conference on Learning Representations, ICLR '18, 2018.

[19] A. Mirhoseini, H. Pham, Q. Le, M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean. Device placement optimization with reinforcement learning. In International Conference on Machine Learning, ICML '17, 2017.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Proceedings of the 33rd International Conference on International Conference on Machine Learning, volume 48 of ICML '16, pages 1928–1937. ACM, 2016.

[21] OpenTSDB. A distributed, scalable monitoring system, 2013.

[22] D. Paul, Y. Peng, and F. Li. Bursty event detection throughout histories. In ICDE, pages 1370–1381. IEEE, 2019.

[23] T. Pelkonen, S. F. J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. In Proceedings of the VLDB Endowment, volume 8 of VLDB '15, pages 1816–1827, Dec. 2015.

[24] P. Przymus and K. Kaczmarski. Dynamic compression strategy for time series database using gpu. In New Trends in Databases and Information Systems, pages 235–244. Springer, 2013.

[25] Schizofreny. Middle-out compression for time-series data, 2018.

[26] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using simd instructions. In Proceedings of the Sixth International Workshop on Data Management on New Hardware, volume 48 of DaMoN'10, pages 34–40. ACM, 2010.

[27] J. Shieh and E. Keogh. isax: disk-aware mining and indexing of massive time series datasets. Data Mining and Knowledge Discovery, 2009.

[28] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. G. Aja Huang, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. Nature, 550:354–359, Oct. 2017.

[29] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross. Massively-parallel lossless data decompression. In 2016 45th International Conference on Parallel Processing, ICPP '16, pages 242–247. IEEE, 2016.

[30] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. IEEE Trans. Circuits Syst. Video Technol., 22(12):1648–1667, Dec. 2012.

[31] F. D. E. Team. Rocksdb: A persistent key-value store for fast storage environments.

[32] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 8(3):229–256, May 1992.