Summary of MAKER

This paper presents a sophisticated data compression and transmission system, aimed at enhancing data transmission from remote locations, such as ships or extraterrestrial vehicles. Building upon the foundation of an earlier system developed by the group called MOBY, this research introduces a new system that addresses the challenge of balancing between complying with a user-defined error bound and a user-defined budget while maximizing the accuracy of the decompressed values.

The newly developed system applies an innovative method to adjust the bitrate of the compressed data by dynamically adjusting the error bounds of the time series for model creation. This intelligent adjustment is based on an online outlier analysis of the data stream, ensuring that crucial data points retain high accuracy during decompression, thereby facilitating subsequent accurate data analysis.

One of the major components of the system is an efficient algorithm for compressing timestamps in irregular time series. Recognizing that irregularity – for example, missing or delayed sensor readings – can indicate importance in a data set, the algorithm utilises a two-pronged approach. First, it constructs run-length compressed lists of irregular timestamps, creating a compressed representation of periods of irregular readings. Second, it further compresses these lists using Huffman coding, which is an established method for lossless data compression.

The second major component is the importance determiner, a method for deciding when to adjust the error bounds. This algorithm is constructed around Welford's Online algorithm combined with Z-scores, enabling the system to dynamically assess the statistical significance of incoming data points. This approach ensures that important data points are identified and preserved with higher accuracy during compression and decompression. Moreover, the system is flexible enough to allow users to replace the current importance-determining algorithm with others as needed, maintaining full compatibility with the rest of the system.

The third major component is the chunk-based scheduler, responsible for dividing data streams into chunks based on a user-defined time period and adjusting the error bounds within each chunk according to a user-defined data budget. This strategy allows the system to manage large volumes of data effectively while maintaining the necessary level of precision within the bounds of a specified data budget.

Together, these components form a versatile and effective system that delivers high-quality results and manages data efficiently. Evaluations show that the system complies well with data budgets, offering lossless compression when the budget permits and managing transmission of excess models efficiently when the data budget is too tight. The system also demonstrates strong performance in terms of processing and memory use, making it a particularly good solution for devices with limited hardware resources. The system's versatility is evident from its ability to perform well on both regular and irregular datasets. The paper concludes with a discussion of the system's major aspects and presents some suggestions for future improvements.

MAKER: Model- And chunK-based approach for Error-bound Regulation

Frederik Agneborn
Computer Science
Aalborg University
fagneb18@student.aau.dk

Emil Laurits Bech
Computer Science
Aalborg University
ebech18@student.aau.dk

Teis Vognstoft Harrington
Computer Science
Aalborg University
tharri15@student.aau.dk

June 16, 2023

***Abstract* – The need for compression and transmission of data from remote locations occurs in many different domains, but transmitting it can be a costly challenge. This paper introduces a novel system, MAKER, that builds upon the model-based compression approach of MOBY[12], adapting it to intelligently manage data compression and transmission under user-defined budgetary constraints. Unlike MOBY, which imposes strict error bounds across the entire data stream, MAKER adjusts error bounds continuously, prioritising important data points while increasing error bounds in time series where substantial data savings are likely and necessary. MAKER combines three main components: a measure of importance, which identifies outliers using Welford's online algorithm and Z-scores; a method of chunking, which segments the data stream into smaller time periods to monitor and adjust error bounds; and a method for compressing irregular timestamps in the data, which compresses the timestamps using a combination of a custom run-length encoding and Huffman coding to account for the potential significance of the irregular data points. Our evaluation demonstrates that MAKER is capable of adhering to the data budget in most cases by intelligently adjusting error bounds. Moreover, MAKER showcases substantial improvements in both performance and memory usage, with up to 26 times faster processing for less irregular data and up to 20 times less memory usage compared to MOBY. These advancements translate into a system that is faster, more efficient, and adaptable, effectively handling the dynamic demands and limitations of real-world scenarios.**

***Index Terms* – Data Compression, Optimisation Methods, Data Processing**

I. INTRODUCTION

Data compression and transmission from remote locations is a necessity across various industrial sectors. One such example is the shipping industry, where time series data from, e.g., sensors need to be transmitted from individual ships to the mainland. This transmission can happen through an expensive satellite communication network if cellular communication is not possible. Another example is the transmission of data from extraterrestrial vehicles, for instance, the *REMS* weather system on Mars.

To overcome this problem, we investigated in our previous work [12] how a lightweight system that is able to run on small processing units can be constructed for such scenarios. Here, we presented *MOBY*; a system able to compress time series using models in an online manner. A model is a mathematical representation of a passage of a time series. MOBY compresses each time series with all model types and selects the model that provides the best compression ratio when none of the models can accommodate the current data point and repeats the process. The model types utilised by MOBY are the constant model type *PMC-Mean*, the linear model type *Swing*, the polynomial model type *Polyswing* and the lossless model type *Gorilla*. Additionally, MOBY is designed so that all created models comply with a user-defined error bound. However, one drawback of this approach is that no guarantees are made regarding the bitrate; a small error bound can potentially result in an exploding bitrate, as the system has no way to limit the bitrate. This paper proposes a new system, *MAKER*, which deals with this problem through a novel approach to attempting to limit the bitrate while still complying with the user-defined error bound.

To better understand the problem domain, we review the landscape of state-of-the-art compression approaches. Different compression techniques are explored, namely *sequential*, *dictionary-based*, and *model-based* techniques. We also consider *Arithmetic coding* and *Delta-delta encoding*. We then direct our focus toward systems that are able to adapt to the input data dynamically. Our technical review lays the foundation for how we construct our intelligent adjustment system, MAKER.

MAKER builds upon and is supposed to run alongside the model-based compression of MOBY[12], however, the bounded bitrate aspect serves as the main contribution of this paper. We adjust the bitrate of the compressed data by adjusting the error bounds of the created models up to a maximum tolerated error bound, which is defined by the user. This error bound parameter differs from the user-defined error bound of MOBY in that it in MAKER denotes the tolerable error bound *when the data is not deemed important*, where MOBY's error bound is a hard limit for all data. This is because MAKER adjusts the error bounds of the time series intelligently based on online outlier analysis of the data stream. When incoming data points are deemed important, we decrease the error bound to achieve as accurate data as possible for these points when they are decompressed. This will benefit the process of analysing the data at a later stage. For the same reason, MAKER is able to compress timestamps of irregular time series losslessly. In this paper, we assume that outlier data points and irregularity indicate important passages in the time series, however, MAKER is implemented so that it can easily be extended with new algorithms for deciding importance of the data. Furthermore, MAKER constantly monitors the size of the compressed models and if our compression is not efficient enough with the current error bound, we increase the error bounds on the time series where we assume it will have the greatest effect on the compression ratio.

The main contributions of this paper are thus summarised in the following:

- We develop an efficient *Timestamp Compressor* algorithm for compressing timestamps for irregular time series, as irregularity (missing or delayed sensor readings) can be a sign of importance. This is done by constructing run-length compressed lists and further compressing those with Huffman coding. This is detailed in Section VII.

- We develop an *Importance Determiner* for determining the importance of the data, and we use this to decide when to adjust the error bounds. In this paper, we employ Welford's online algorithm combined with Z-scores, however, a system user can easily replace this with other approaches for determining whether data points are important, and the rest of the system will be fully compatible. This is detailed in Section VIII.

- We develop a *Chunk-based Scheduler* which is responsible for dividing the data points into chunks based on a user-defined time period and adjusting the error bounds within each chunk according to a user-defined data budget. This is detailed in Section IX.

Collectively, these components form the holistic MAKER, which synergistically leverages the strengths of each component to offer optimal performance and data quality. The system as a whole is versatile and suitable for a wide range of applications, even with the inherent limitations of small processing units.

Our evaluation shows that MAKER in most cases is capable of complying with the data budget to a satisfactory extent. MAKER only exceeds the budget in the critical cases where the data budget is too low compared to the compression ratio that the model-based compression is capable of achieving with the given maximum allowed error bound. When this happens, the excess models are delayed until the next transmission, where they are prioritised for transmission over the new models. On the other hand, if the budget allows it, MAKER offers lossless compression of all time series. Furthermore, we show that our *Timestamp Compressor* works especially well on data sets containing data with a high extent of regularity, while it does not explode on highly irregular data sets. Lastly, we show that MAKER is very efficient processing- and memory-wise, which is desirable as MAKER is supposed to run on devices with limited hardware where other services are running simultaneously.

The rest of the paper is structured as follows: Section II introduces definitions of core concepts used throughout the paper; Section III presents the relevant compression techniques and projects that lay the foundation for this paper; Section IV formalises the problem of this paper; Section V presents an overview of the components of MAKER; Section VI presents a small exemplary data set, which is used in Section VII, Section VIII, and Section IX, which details our three main components; Section X presents our evaluation of MAKER; Section XI discusses major aspects of the system; and lastly, Section XII concludes the paper.

II. DEFINITIONS

This section provides an overview of the technical terms used in this paper along with definitions for these. Definitions 1 to 4 and 6 are adapted from our previous work [12].

DEFINITION 1. *A* ***time series*** ts *is a sequence of pairs* (v, t)*, where* v *is a numerical value, e.g. a sensor reading, and* t *is a timestamp. For a* ts *consisting of pairs* $\langle(v_1, t_1), (v_2, t_2)..., (v_n, t_n)\rangle$*, where* n *is the number of pairs,* $t_{i+1} > t_i$ *holds for all* $1 \leq i \leq n$*.*

DEFINITION 2. *A* ***regular time series*** ts_r *is a* ts *where the condition* $t_{i+1} - t_i = t_{j+1} - t_j$ *holds for all* $1 \leq$

$i < n$, $1 \leq j < n$, *i.e. the interval between consecutive timestamps in* ts_r *is the same for all timestamps.*

DEFINITION 3. *An* ***irregular time series*** ts_i *is a* ts *where the condition* $t_{i+1} - t_i \neq t_{j+1} - t_j$ *holds for some* $1 \leq i < n$, $1 \leq j < n$, *i.e. there is at least one interval between two consecutive timestamps that is not equal to the interval between two other consecutive timestamps. Note that* ts_r *becomes irregular if one of its data points is issued too early, too late or if it is missing. We use the notation* ts *when the regularity has no importance.*

DEFINITION 4. *An* ***error bound*** ϵ *describes how much the decompressed data is allowed to deviate from the original data, i.e. the condition* $|v_i - \hat{v}_i| \leq v_i \cdot \epsilon$ *holds for all* $1 \leq i \leq n$ *(see Definition 6 for an explanation of* $\hat{v}$*). As* ϵ *is relative to the size of* v*, we tolerate a larger absolute difference between* v *and* $\hat{v}$ *for large values of* v *than for small values of* v*.*

DEFINITION 5. *A* ***model type*** *describes how a subsequence of a* ts *is compressed. Different model types use different approaches for compression, and the decompression process depends entirely on the model type used for compression. A* ***lossless*** *model type is a model type that guarantees that decompression results in the exact values that were compressed, i.e.,* $v_i = \hat{v}_i$ *for all* $1 \leq i \leq n$*. In this paper, we use the term* ***lossy*** *model type to denote a model type that allows an error, however, a lossy model type guarantees that the decompressed values do not exceed a set error bound.*

DEFINITION 6. *A* ***model*** m *is a representation of a subsequence of* ts*. A model has a model type along with the specific parameters relevant to the model type, which are used for decompression. For a data point* (v_k, t_k)*, we let* $\hat{v}_k$ *denote* $m(t_k)$*, i.e. the decompressed value using the model parameters.*

DEFINITION 7. ***Model-based compression*** *denotes a compression method where data is represented as models. The approach is to fit* (v_c, t_c)*, where* c *denotes the current data point, using a number of model types simultaneously. When none of the models are able to fit* (v_c, t_c)*, the model* m_{best} *that results in the smallest bitrate (see Definition 9). We let* (v_l, t_l) *denote the last* (v, t) *in* m_{best}*. The process then repeats from* (v_{l+1}, t_{l+1})*.*

DEFINITION 8. *A* ***chunk*** γ *is a sequence of* (v, t) *pairs, where for* $\forall(v, t) \in \gamma_i,\ \rho \cdot (i-1) \leq t < \rho \cdot i$*, where* $i > 0$ *and* ρ *is a number of seconds. A chunk thus denotes a time period with a fixed length of* ρ *time units. The sequence of chunks,* Γ*, consists of chunks that can be linearly ordered, such that* $\gamma_1, \gamma_2, \gamma_3 \ldots \gamma_m \in \Gamma$*, where* $m = \lceil \frac{t_n}{\rho} \rceil$ *and* $\forall(v_i, t_i) \in \gamma_i, \forall(v_j, t_j) \in \gamma_j,\ t_i < t_j$ *where* $i < j$*.*

DEFINITION 9. *The* ***bitrate*** *of a model* m *denotes how many bits on average that* m *uses to represent a value. It is defined as* $\frac{s(m)}{|m|}$*, where* $s(m)$ *denotes a function returning the total size of model parameters of* m *expressed in number of bits, and* $|m|$ *denotes the number of data points* m *represents.*

III. RELATED WORK

A. Compression techniques

In recent years, the need for effective time series compression techniques has grown significantly due to the massive amount of data generated by various sources, including IoT devices, industrial applications, and critical infrastructure systems. These devices generate an enormous amount of time series data, necessitating a diverse range of compression techniques to increase the efficiency of data collection, storage, and analysis. The main contribution of this related work section is to provide a comprehensive summary of current time series compression algorithms, which have been fragmented across sub-domains ranging from databases to IoT sensor management, as explained in the survey[7] that this subsection is inspired by. By presenting a taxonomy of these techniques based on their approach and properties, the author of [7] aims to guide the reader in making informed decisions when selecting the most suitable compression method. Reading into the different properties of the chosen techniques, it is possible to broadly categorise them into **sequential** and **model-based** compression approaches.

a. Sequential compression

Sequential compression algorithms combine simple compression techniques into a single compression chain to maximise the effect of each type of compression. Here we present two exemplary sequential algorithms:

- *Delta Encoding*, *Run-length*, and *Huffman (DRH)*[20] combines delta encoding, which represents the difference between consecutive data points; run-length encoding, which replaces consecutive occurrences of the same value with a single occurrence and a count; and Huffman coding, which assigns variable-length codes to data values based on their frequencies.
- *Run-Length Binary Encoding (RLBE)*[27] is a lossless method specifically designed for low-resource devices like the ones found in IoT infrastructures. It combines delta encoding, run-length, and Fibonacci coding, the latter of which represents integers using a unique combination of Fibonacci numbers.

b. Model-based compression

Model-based compression techniques aim to represent time series as functions of time by dividing them into subsequences and finding approximating mathematical functions for each subsequence. These techniques work much better when lossy compression is allowed but can function surprisingly well with a lossless approach. Additional positives of them are that they do not depend on the specific data domain. The following examples are model types that can be used as functions to compress the data and they also include a fully implemented technique that selects the correct model based on the best approximation:

- *PMC-Mean (Poor Man's Compression-Mean)*[19] is a basic data compression technique that replaces a number of data points with a single mean value. When a new data point causes any of the modelled data points to deviate too much from the mean, a new model is created to represent the next data points.
- The *Swing*[11] filter utilises a set of linear functions called the upper and lower boundaries. Any line situated between these boundaries can represent the data points collected so far while adhering to the current error bound. To preserve this characteristic, the lower boundary needs to be swung upward, while the upper boundary must be swung downward as new data points are recorded. The new boundaries are defined by the initial and the most recent data points collected. If a new data point falls too far below the lower boundary or too far above the upper boundary, it will be disregarded since no line can represent the latest data point and a new segment must be constructed.
- The *Gorilla*[23] technique is a lossless compression algorithm developed at Facebook for floating-point time series data. It consists of two main steps: timestamp compression and value compression. Timestamp compression uses *Delta-delta encoding* to store differences between consecutive timestamps, while value compression uses an XOR encoding between the binary representation of consecutive values.
- *Piecewise Polynomial Approximation* (PPA)[10] divides a time series into several segments of fixed or variable lengths and finds the best polynomial approximations for each segment. A maximum deviation from the original data can be enforced to maintain a certain reconstruction accuracy. PPA employs a greedy approach and three online regression algorithms to approximate time series with constant functions[19], straight lines[8], and polynomials[26].

By categorising these techniques into sequential and model-based techniques, we gain a comprehensive understanding of the current landscape of time series compression algorithms that fit within our scope. Both categories have their own advantages, limitations, and unique characteristics, making them suitable for different scenarios and requirements. When selecting the most suitable compression technique for a specific application, it is crucial to consider factors such as the nature of the data, desired compression ratio, the acceptable level of loss, computational resources, and the specific needs of the application. While specialised tools and techniques have their own advantages in some applications of compression, the state of the art in time series compression research is comprised of various general-purpose data compression techniques developed to handle a wide variety of time series data. Additionally, the constant evolution of time series compression research suggests that new, more efficient algorithms are likely to be developed in the future, further expanding the options available for practitioners in this field.

c. Other compression techniques

In addition to the techniques described in [7], we present in this section two commonly used encoding techniques; *Arithmetic coding* and *Delta-delta encoding*.

- *Arithmetic coding*[30] is a compression technique that maps a sequence of elements to a subinterval inside a larger interval. It utilises information about element frequency to construct the subintervals so that the subintervals of frequently occurring elements are larger than subintervals of rarely occurring elements. Compression then happens by "zooming" into the subinterval corresponding to the element to be compressed. The process then repeats using the subinterval instead of the larger interval. As the subinterval becomes small, it becomes problematic to keep precision, so a series of re-scaling operations are employed by both the compressor and decompressor.
- *Delta-delta encoding*[23] is a technique that is similar to the aforementioned Delta encoding, however, Delta-delta encoding builds a bitstring based on how much the difference between the latest two timestamps differs from the previous difference. The smaller this difference becomes, the fewer bits are used to express the new timestamp.

B. Dynamic adaptable compression systems

To garner a better understanding of dynamically adapting time series compression systems, we review the literature on systems that can adapt the compression ratio and the selected algorithm based on both the importance of the data and the available bandwidth. A variety of approaches have been proposed to address these challenges and improve the performance of time series compression under such circumstances.

[5] proposes a novel approach to rate control in multimedia systems. The authors introduce an elastic scheduling framework that dynamically adjusts the rate at which multimedia data is transmitted based on network and system conditions. The elastic scheduling framework is based on three key components: a rate estimator; a rate allocator; and a congestion controller. The rate estimator measures the available network bandwidth and estimates the maximum transmission rate that can be sustained without causing congestion. The rate allocator then assigns transmission rates to multimedia data based on their priority, size, and available bandwidth. Finally, the congestion controller monitors network traffic and adjusts the rate to avoid congestion and maintain overall system performance. The results of their evaluation show that the approach is effective in maintaining high-quality multimedia streaming while avoiding congestion and other performance issues. Additionally, the approach is also shown to be robust and able to adapt to different network conditions, making it suitable for a wide range of multimedia applications.

Optimising time series compression ratios require a method tailored to data characteristics, but the time used for tuning is an overhead that slows down compression, making it unsuitable for real-time monitoring systems. Therefore, in the context of leveraging the computational power of GPUs for time series data compression, [24] introduces a dynamic compression strategy planner that effectively selects compression techniques based on the data's characteristics, such as data distribution and frequency of updates. The authors demonstrate that their method outperforms the non-dynamic approach in terms of compression ratio, query performance, and energy consumption, making it suitable for real-time data processing.

[13] introduces an intelligent and adaptive data compression algorithm selection strategy to handle big data processing tasks within local file systems. By incorporating a dynamic algorithm selection module consisting of three components: a profiler; a selector; and a compressor, the module chooses a high compression ratio algorithm for easily compressible data, a quick compression algorithm for data with low compressibility, and omits compression for incompressible data. The evaluation demonstrates that this dynamic selection module can boost response time and decrease storage requirements for large data sets.

Energy and transmission constraints in sensor networks necessitate minimising data volume while retaining high data quality. The *Rate Adaptive Compression with Error bound (RACE)* algorithm introduced in [6], which is a wavelet-based, error-aware compression technique, addresses these needs by adapting to limited bandwidth and adjusting errors based on network capacity. Their approach facilitates easy error estimation during data reconstruction and takes advantage of excess bitrate availability to refine error ranges using a feedback mechanism when possible. Consequently, the algorithm enhances performance and preserves the statistical interpretation. Evaluations confirm the RACE algorithm's effectiveness in rate adaptivity, error range reduction, and preservation of data quality.

As sensor data collection on boats, like cargo ships, becomes more frequent, managing vast amounts of data with limited storage space and costly, slow satellite networks becomes a challenge. In [12], we introduce MOBY, a compression system that efficiently compresses time series data within specified error bounds and sends it to a remote server. Developed with a focus on minimal memory usage and binary size, MOBY is tailored for compatibility with older, resource-limited systems prevalent in the maritime industry. MOBY offers effective compression rates and compact binary size, making it a feasible solution for the sensor collection systems currently in use in the industry.

[28] presents FRaZ, a novel fixed-ratio lossy compression framework tailored for scientific floating-point data, which adheres to user-defined error bounds. The study makes two key contributions:

1. Devising an effective iterative method for accurately identifying suitable error settings for various compressors based on desired compression ratios.
2. Assessing the performance and precision of the proposed framework using multiple cutting-edge compressors and real-world scientific data sets from a diverse range of domains.

Evaluation of the experiments shows that FRaZ can identify the optimal error settings of any given lossy compressor effectively. While fixed-ratio compression takes longer than fixed-error compression, FRaZ provides a significant new approach for handling extensive scientific floating-point data sets.

The literature on dynamically adapting time series compression systems highlights various innovative approaches and techniques for optimising compression under various restrictions. These approaches address challenges such as rate control, energy consumption, real-time processing, algorithm selection, and data quality preservation. Elastic scheduling frameworks, dynamic compression strategy planners, adaptive algorithm selection strategies, and wavelet-based error-aware techniques are among the solutions that have been proposed and evaluated. Furthermore, domain-specific compression systems like MOBY and FRaZ address the unique needs of the maritime industry and scientific data management, respectively. Collectively, these studies demonstrate the effectiveness and adaptability of such approaches, providing valuable insights for the development and improvement of time series compression systems tailored to more restrictive domains.

MOBY[12] and FRaZ[28] have their unique strengths and address important aspects of time series data compression, but they do not attempt to meet two crucial constraints simultaneously. These systems can either guarantee adherence to a predefined data budget or ensure data accuracy by not exceeding predefined error bounds. Systems that can manage data within a predefined budget can make optimal use of the available resources and ensure that critical data can be stored and transmitted in light of these limitations. Systems that can guarantee data accuracy within predefined error bounds ensure that compressed data remains suitable for its intended use. However, no solution exists that attempts to balance both of these targets simultaneously.

It is also apparent that the existing systems lack the capability to dynamically adapt to changes in the importance of the data. For instance, in time series data, the distribution and patterns in the data can evolve over time due to various factors. Taking data importance into account can further optimise compression strategies by prioritising the accuracy of crucial data. A truly adaptable compression system would need to be able to dynamically adjust its strategies not only based on changes in data characteristics and network conditions but also in response to the perceived data importance.

IV. PROBLEM

In the context of a compression system that would try to balance data accuracy and budget constraints, determining the importance of data plays a pivotal role in the overall effectiveness of the solution. By evaluating the significance of the data points, the system can be enabled to allocate resources and adjust error bounds intelligently, prioritising the preservation of critical information. By identifying and preserving these data points with higher accuracy, the system can provide valuable insights into the underlying data and help detect potential issues. We denote the importance of a data point (v_x, t_x) as $\mathcal{I}_x$ and define it as the following:

$$\mathcal{I}_x = \begin{cases} 1, & \text{if } (v_x, t_x) \text{ is deemed important} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$\mathcal{I}$ is defined as a binary measure rather than a numeric measure to be able to handle the case where the data of the processed tss fluctuate, which could yield a fluctuating numeric importance measure. If the error bounds of the models were adjusted using such an importance measure, the error bounds would potentially be lowered when the importance measure is raised even slightly due to fluctuations. This, in turn, would require that the constructed models are terminated, as the data points before the adjustment would no longer be guaranteed to be within the new error bound. A binary importance measure, on the other hand, ignores such fluctuations and instead processes all reported important data points equally. We also introduce the *error* of data points in a time series ts_k as $\mathcal{E}$ (Eq. (2)), which is defined through the *Mean Absolute Percentage Error (MAPE)* measure, i.e. as an average of errors between the original and the decompressed data points of ts_k. Further, $\mathcal{E}_\mathcal{P}$ denotes the error for a subset $\mathcal{P}$ of points in ts_k. We here use $\mathcal{P}_I$ to denote the subset of *important* points, i.e. $\mathcal{P}_I = \{(v_i, t_i) \in ts_k \mid \mathcal{I}_i = 1\}$ and $\mathcal{P}_O$ to denote the subset of *ordinary* points, i.e. $\mathcal{P}_O = \{(v_o, t_o) \in ts_k \mid \mathcal{I}_o = 0\}$.

$$\begin{aligned} \mathcal{E}_{\mathcal{P}_O}(\mathcal{M}_O) &= \frac{\sum_{(v_o,t_o)\in\mathcal{P}_O} \frac{|v_o - \hat{v}_o|}{v_o}}{|\mathcal{P}_O|} \\ \mathcal{E}_{\mathcal{P}_I}(\mathcal{M}_I) &= \frac{\sum_{(v_i,t_i)\in\mathcal{P}_I} \frac{|v_i - \hat{v}_i|}{v_i}}{|\mathcal{P}_I|} \end{aligned} \tag{2}$$

We use $\mathcal{M}$ to denote a sequence of pairs of models and corresponding error bounds to represent ts_k, i.e. $\mathcal{M} = \{(m_1, \epsilon_1), (m_2, \epsilon_2), ..., (m_n, \epsilon_n)\}$, where n is the number of models used to represent ts_k, and for each (v_k, t_k) represented by some m_j, where $1 \leq j \leq n$, we have that $|v_k - m_j(t_k)| \leq v_k \cdot \epsilon_j$ (i.e. $|v_k - \hat{v}_k| \leq v_k \cdot \epsilon_j$). $\hat{v}_i$ and $\hat{v}_o$ in Eq. (2) thus denote decompressed values from their corresponding model m_k in $\mathcal{M}$. The two parts of Eq. (2) each result in a low $\mathcal{E}$ when we intuitively have a high data quality, i.e. the decompressed values are similar to the original data points.

To handle multiple tss, we use $\mathcal{T}$ to denote a set of $\mathcal{M}$, i.e. $\mathcal{T} = \{\mathcal{M}^1, \mathcal{M}^2, ..., \mathcal{M}^n\}$, where n denotes the number of tss. We let $\mathcal{M}_I$ denote the subset of models in $\mathcal{M}$ involving important data points and $\mathcal{M}_O$ the subset of models in $\mathcal{M}$ involving ordinary data points. $\mathcal{T}_I$ is then defined as $\mathcal{T}_I = \{\mathcal{M}_I^1, \mathcal{M}_I^2, ..., \mathcal{M}_I^n\}$ and $\mathcal{T}_O$ as $\mathcal{T}_O = \{\mathcal{M}_U^1, \mathcal{M}_U^2, ..., \mathcal{M}_U^n\}$.

The objective of this paper is thus to find some $\mathcal{T}$, such that the sum of all $\mathcal{E}$ is minimised:

$$\min_{\mathcal{T}_I, \mathcal{T}_O} \sum_{\mathcal{M}_I \in \mathcal{T}_I} \frac{\mathcal{E}(\mathcal{M}_I) \cdot |\mathcal{M}_I|}{|\mathcal{T}_I|} + \sum_{\mathcal{M}_O \in \mathcal{T}_O} \frac{\mathcal{E}(\mathcal{M}_O) \cdot |\mathcal{M}_O|}{|\mathcal{T}_O|} \tag{3}$$

$$\text{s.t.} \begin{cases} \left(\sum_{\mathcal{M}\in\mathcal{T}} \sum_{(m,\epsilon)\in\mathcal{M}} S(m) + S(t(m))\right) \leq \Theta, & \\ \begin{aligned} 1 \leq i \leq n, \forall(m_I, \epsilon_I) \in \mathcal{M}_I^i, \forall(m_O, \epsilon_O) \in \mathcal{M}_O^i, \\ \epsilon_I < \epsilon_O, \end{aligned} & \text{if } S(\mathcal{T}_L) + S(t(\mathcal{T}_L)) > \Theta \\ \begin{aligned} 1 \leq i \leq n, \forall(m_I, \epsilon_I) \in \mathcal{M}_I^i, \forall(m_O, \epsilon_O) \in \mathcal{M}_O^i, \\ \epsilon_I = \epsilon_O = 0, \end{aligned} & \text{if } S(\mathcal{T}_L) + S(t(\mathcal{T}_L)) \leq \Theta \end{cases}$$

where S denotes a function returning the byte size of the input; $\mathcal{T}_L$ denotes a set of $\mathcal{M}_L$, where each $\mathcal{M} \in \mathcal{T}_L$ contains only lossless models; n denotes the number of tss; Θ denotes an upper bound for the bitrate; $|\mathcal{M}|$ denotes the number of data points represented by all $(m, \epsilon) \in \mathcal{M}$; $|\mathcal{T}|$ denotes the total number of data points represented in $\mathcal{T}$;

and t is a function that returns a lossless representation of the timestamps associated with the values represented by the models in the input.

We thus aim to find the optimal sets of models across all tss, $\mathcal{T}_I$ and $\mathcal{T}_O$ for important data points and ordinary data points respectively. The first condition in Eq. (3) ensures that the total byte size for all models in $\mathcal{T}$ is below the upper bound. The second condition ensures that we prioritise a lower error bound for important data points if it is not possible to compress all data points losslessly. The third condition allows all models to be losslessly compressed if the total byte size does not exceed the upper bound. Eq. (3) thus prioritises minimising the difference between $\{v_i \mid (v_i, t_i) \in \mathcal{P}_I\}$ and $\hat{v}_i$ over minimising the difference between $\{v_o \mid (v_o, t_o) \in \mathcal{P}_O\}$ and $\hat{v}_o$ across all tss. Eq. (3) is based on the assumption that the first condition can be met. There are, however, cases where this assumption does not hold, namely if Θ is set too low compared to the capabilities of the model-based compression. In this paper, we present *Model- And chunK-based approach for Error-bound Regulation* (*MAKER*); a system that aims to minimise Eq. (3) and is specifically designed to deal with this aforementioned case.

DEFINITION 10. *The* ***goal of MAKER*** *is to minimise Eq.* (3) *continuously as data points arrive while being able to deal with a critical case of Eq.* (3)*, namely the case where the first condition cannot be met, which happens when the data cannot be compressed to a sufficiently small size that complies with the upper bound.*

V. SYSTEM OVERVIEW

Building on the concepts discussed earlier, MAKER is designed to adapt to the determined importance of that data. The process is structured into two parts, as shown in Fig. 1: One for processing incoming data, and the other for acting when a new chunk of data is ready.

The first part of the process is designed to handle the incoming data stream and is implemented as generically as possible to handle any kind of time series data from various domains. Initially, the data is analysed using the *Importance Determiner* component which identifies data points that deviate significantly from expected patterns, hence potentially being more important or relevant. As MAKER is designed to be able to operate on small processing units akin to those found in, e.g., ship antennas or extraterrestrial vehicles, deploying deep learning approaches to identify outliers, i.e., potential failure patterns, are not viable. Instead, we propose an efficient and lightweight approach that leaves a small memory footprint. We outline our approach in Section VIII, but the system is designed to easily incorporate other user-defined algorithms for analysing the importance of data.

Following this, the values from the tss are compressed into models using the PMC-Mean, Swing filter, and Gorilla techniques as explained in the compression system presented in [12], while timestamps are processed using our novel *Timestamp Compressor*, further detailed in Section VII.

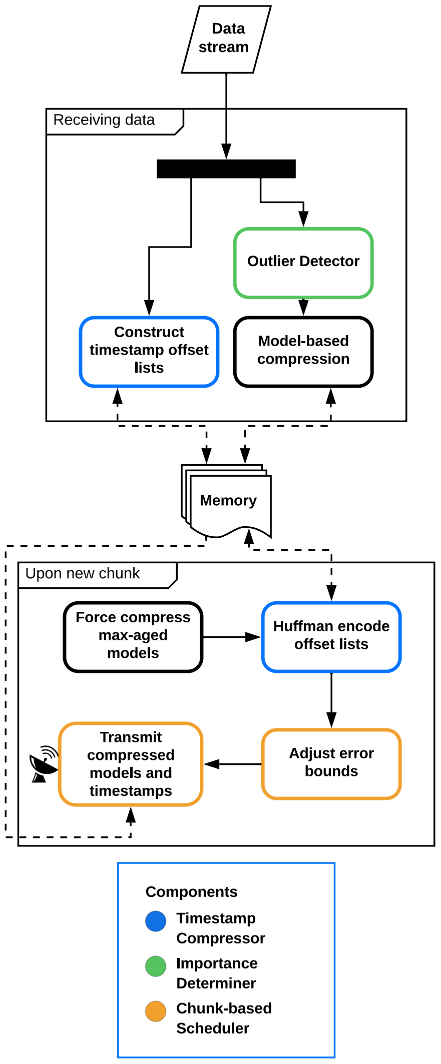


Figure 1: Flowchart of MAKER

One of the common existing methods for handling irregular time series includes re-sampling the time series by interpolating values, thereby making them regular if the time series is irregular due to missing values [2]. Another approach is to simply remove the data that contains irregularity, which is the approach of *ModelarDB* [17], as the system described in that paper can only handle regular time series. Handling irregular time series is crucial as it helps to maintain the integrity of the data, preserving critical information that could be lost or misrepresented if the data were to be re-sampled, interpolated or even discarded. Additionally, irregular time series may contain useful information about the underlying system that is not evident in regular time series. We know from both ModelarDB and our previous work *MOBY* [12] that irregular time series data do occur, and since missing or delayed values can be a sign of importance, as they can be caused by e.g. a malfunctioning sensor[1, 15], the focus should be on keeping this information rather than discarding it.

With our *Timestamp Compressor*, the system is equipped to handle such delayed or missing sensor readings instead of discarding or interpolating missing values through a process of constructing offset lists containing run-length compressed timestamps. This step involves reading from and writing to memory due to a caching mechanism, as shown on Fig. 1. Completed models are also stored in memory until transmission, preserving the information required for decompression.

Given the computational and storage constraints associated with some devices, it is often not desirable to write data to disk due to the computational costs of I/O. In addition, another reason to avoid writing to disk is the inherent longevity concern of the hardware. The lifespan of flash memory (e.g. SSD disks) can be greatly reduced due to the fact that they only allow a finite number of writes before their lifetime is over[9]. This highlights the potential risks and inefficiencies of disk-dependent operations. Therefore, recognising that we cannot also make the assumption of large memory availability in these devices, the emphasis on memory efficiency within our system becomes paramount. It is with these considerations in mind that our system has been designed to perform all its tasks in memory while optimising the usage of available memory resources. All operations, from the caching mechanism used in the compression process to storing the finished models, are memory-bound until the data is ready for transmission.

The second part of the process shown in Fig. 1 operates when a new data chunk is to be created. This component is responsible for transforming the residual data from the time series that have not yet been modelled into model representations. Subsequently, the offset lists containing the timestamps of the completed chunk are read from memory and compressed using the variable-length encoding method, Huffman coding [16]. The outcome of this step is then stored in memory and transmitted alongside the model representations of the data.

To ensure efficient utilisation of resources and maintain a user-defined data budget, the *Chunk-based Scheduler* adjuster is employed before transmission. This is further explained in Section IX. This component dynamically adjusts the error bounds of the compressed data models according to the restrictions of the data budget and the calculated importance of the data points, where crucial points are represented with lower error bounds. This introduces a balance between data quality and compression efficiency by favouring important data points over ordinary data points, as described by Eq. (3). The adjusted models are then transmitted, ensuring data integrity is maintained even under data budget constraints.

To summarise, the proposed architecture provides a cohesive solution for handling irregular time series data, optimising data quality and compression efficiency by minimising Eq. (3), and effectively adjusting error bounds according to the data budget and the determined data importance. By preserving critical information instead of discarding it, it is well-positioned to support data analysis tasks such as failure detection, offering enhanced capabilities for various applications in the industry.

VI. RUNNING EXAMPLE

In this section, we present a small data set which will be used in the following sections as the basis for the presented examples.

Example 1. Table 1 shows an example of five ts adapted from the *REMS* data set [21]. The *Timestamp* column contains all timestamps across all ts. The *Wind*, *Wind sensor temperature*, and *Humidity* tss are regular, as they have the same number of seconds between each of their values. The *ADC calibration* and *Pressure* tss, however, are irregular; the *ADC calibration* ts misses a value at timestamp 480, and the second reading of the *Pressure* ts is delayed five seconds.

Table 1: Sensor values and timestamps adapted from *REMS*

Time-stamp (Seconds)	Wind (Analog signal)	ADC calibration (Analog signal)	Wind sensor temperature (Analog signal)	Humidity (Analog signal)	Pressure (Analog signal)
0	377.0	31,325.0	3,656.0	492,779	-
60	348.0	32,543.0	-	-	3,235,335
120	269.0	32,440.0	-	492,761	-
180	281.0	31,286.0	3,779.0	-	-
240	306.0	32,394.0	-	492,791	-
300	319.0	2,180.0	-	-	-
305	-	-	-	-	3,235,330
360	276.0	20,196.0	2,463.0	492,795	-
420	243.0	64,658.0	-	-	-
480	237.0	-	-	492,804	-
540	248.0	13,775.0	1,666.0	-	3,235,339
600	271.0	32,719.0	-	-	-

VII. TIMESTAMP COMPRESSOR

To compress timestamps, we provide a more sophisticated approach compared to the naïve approach of MOBY [12]. Here, timestamps were duplicated across tss, resulting in a high degree of redundancy. The approach to handling timestamps in this paper aims to obtain four goals: 1) to minimise redundancy, 2) to make it possible to handle irregular time series, 3) to minimise the footprint in memory while the algorithm is running and 4) to minimise the size of the transmitted timestamps. Our approach consists of two stages. We provide an overview of these stages in Section VII.A.

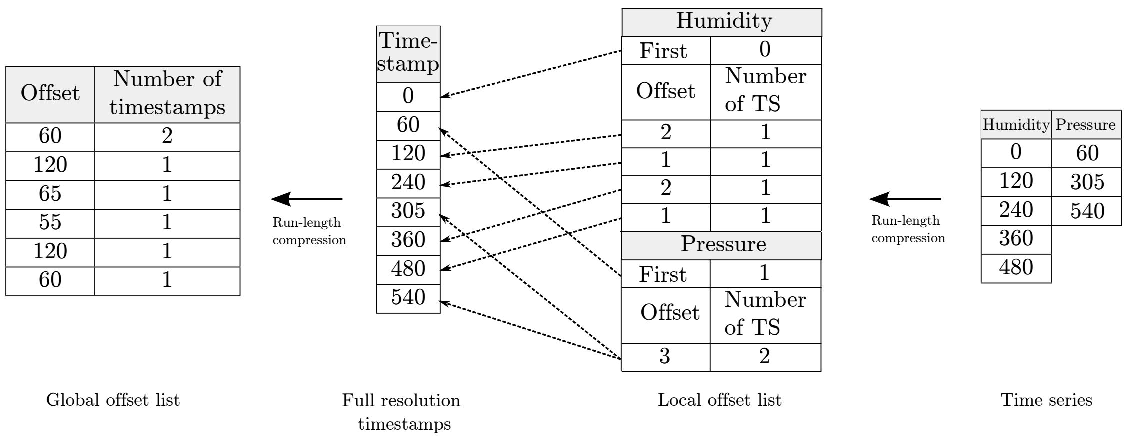


Figure 2: Overview of relationship between global offset list and local offset list

A. Overview of timestamp compression

The first stage of our *Timestamp Compressor* is to build two data structures called the *global offset list* and the *local offset list.*

Example 2. To illustrate the relationships between these lists and how they relate to the full resolution timestamps, an overview based on only *Humidity* and *Pressure* in Example 1 is shown in Fig. 2. Here, four tables are shown: a *Full resolution* table containing all timestamps across all tss, ordered from earliest to latest timestamp and without duplicates; a *Time series* table containing full resolution timestamps for each time series; and two compressed tables, the *global offset list* and the *local offset list*. The global offset list contains a compressed representation of the *Full resolution* table. The *local offset list* is a structure containing compressed representations for all timestamps across all tss. The elements in the *local offset lists* refer to the *Full resolution* table (indicated by dashed arrows). Furthermore, for each ts, we store the index of the first timestamp in the *global offset list.* We detail this stage in Section VII.B.

In the second stage of timestamp compression, the *global offset list* and the *local offset list* are both compressed using Huffman coding. We detail this stage in Section VII.C.

B. Offset lists

We now detail the process of constructing the offset lists while building a complete example containing all tss of Example 1. The general idea of this approach is to store all timestamps once and then for each ts refer to those timestamps instead. Both of these parts can be compressed as explained in the following. When a new timestamp arrives, it is compressed using a run-length approach on the offsets.

Table 2: Global offset list containing offsets from timestamp 0

Offset	Number of timestamps
60	5
5	1
55	1
60	4

Example 3. Table 2 shows how the timestamps of Example 1 are represented in a global offset list, corresponding to the leftmost table in Fig. 2. The first timestamp, *0*, is used as the starting point for calculating the offsets. From this, five timestamps follow, each with a 60-second interval. After this, a single interval of 5 seconds emerges followed by a single interval of 55 seconds. Finally, four timestamps follow, each with an interval of 60 seconds.

To store timestamps associated with each ts, timestamps are given an incremental ID, i.e. the first timestamp in the *Timestamp* column in Table 1 gets 0, next gets 1 etc. For each ts, the ID of the timestamp corresponding to the first value in that ts is stored. The local offset list, which is similar to the global offset list shown in Table 2, is then made, containing offsets for each ts. Here, however, the *Offset* column describes the number of timestamps in the timestamp list being skipped.

Example 4. The local offset list containing offset lists for each ts' timestamps are shown in Table 3. The first timestamp of *Wind* is reconstructed by looking up the timestamp on index 0 in the list of full resolution timestamps, reconstructed from the global offset list, i.e. timestamp 0. The next five timestamps (as denoted by the *Number of timestamps* column) for *Wind* are found by moving one index forward in the global offset list (as denoted by the *Offset* column) per timestamp. Timestamps for the other columns are reconstructed in the same way.

As Tables 2 and 3 illustrate, the output of this stage is two lists that can contain duplicate values. To compress these lists while taking advantage of the frequencies of the values, Huffman coding[16] is used as detailed in Section VII.C.

Table 3: Local offset list

Wind		ADC calibration		Wind sensor temp.	
Index for first timestamp					
0		**0**		**0**	
Offset	**Number of timestamps**	**Offset**	**Number of timestamps**	**Offset**	**Number of timestamps**
1	5	1	5	3	1
2	1	2	1	4	1
1	4	1	1	3	1
		2	1		
		1	1		

Humidity		Pressure	
Index for first timestamp			
0		**1**	
Offset	**Number of timestamps**	**Offset**	**Number of timestamps**
2	2	5	1
3	1	4	1
2	1		

To minimise the memory footprint of the timestamps, our model-based compression is designed to run on the local offset lists instead of keeping all full resolution timestamps in memory for each ts. We do, however, keep a list of all timestamps, which corresponds to a reconstructed version of the global offset list, in memory. This is a trade-off between memory efficiency and performance; we choose to keep this list of global timestamps in memory to avoid having to reconstruct the entire global offset list every time we need a corresponding timestamp for a cached value. The offset lists are created as the data points come in, and these representations are used for transmission. Just before transmission, however, the Huffman coding of the offset lists happens to further reduce the size of the transmitted data. The local offset list is kept uncompressed in memory instead of Huffman encoding them immediately to avoid the performance overhead of decompressing the Huffman encoded offset lists when the timestamps are needed during model compression. To reconstruct the full resolution timestamps for all tss, the *Full resolution* table (see Fig. 2) is first constructed based on the global offset list and finally, we look up the correct timestamps within this list based on the stored index of the first timestamp along with the offset values in the local offset list.

C. Huffman compression

This section describes the Huffman compression stage that is run on the offset lists just before transmission. For a thorough description of the core concepts of Huffman coding, we refer to [16], while we in this section describe how the method has been tailored to our needs.

Huffman coding is a *variable-length* method which assigns short bit-string codes to frequent values in a set, while infrequent values get longer codes.

Example 5. Figs. 15 and 16 in Section XIV (Appendices) show Huffman trees made from the offset lists shown in Tables 2 and 3. The corresponding Huffman codes are shown in Tables 8 and 9 (Appendices) respectively. This is included for completeness.

We use specific control codes to enable correct decompression. Specifically, we use a special value, *EOTS*, to indicate the end of a ts (in the case of the local offset list) and another special value, *EOL*, to indicate the end of the list. Like the elements of the offset lists, these values get their own Huffman code. Furthermore, the value that follows *EOTS* is always the index of the first timestamp in the global offset list (shown in Table 3).

To decode the Huffman codes, the decompressor needs the correct Huffman codes. These can be transmitted with an encoding of the Huffman tree constructed by traversing the tree recursively, starting from the root and accessing the children left to right. We then use a 1 bit to describe a leaf node followed by the value it represents, and a 0 bit to represent a non-leaf node. Algorithms 1 and 2 show the recursive process of encoding the tree. All leaf node values are represented with a fixed number of bits in the tree encoding. For instance, most systems allocate 32 bits for integers, so the same number of bits per value is used in the tree encoding. We include the algorithm for decoding the tree in Algorithm 5 (Appendices) for completeness. The process of encoding and decoding the Huffman tree is adapted from [18].

Algorithm 1 EncodeTree

```
Input: Root node Root
Output: Encoding of Huffman tree
1: Bit string Bits
2: Encoding ← EncodeTreeRecursive(Root, Bits)
3: return Encoding
```

Algorithm 2 EncodeTreeRecursive

```
Input: Tree node Node, bit string Encoding
Output: Encoding of Huffman tree
1: if (Node is a leaf-node) then
2:     Append 1 to Encoding
3:     Append Node.Value to Encoding
4: else
5:     Append 0 to Encoding
6:     EncodeTreeRecursive(Node.leftChild, Encoding)
7:     EncodeTreeRecursive(Node.rightChild, Encoding)
8: end if
9: return Encoding
```

Example 6. Tables 4 and 5 show the result of compressing the global offset list (Table 2) and the local offset list (Table 3) using the Huffman codes shown in Tables 8 and 9 (Appendices) respectively. The *Tree* rows show the results of Algorithms 1 and 2 on the offset lists. The values in square brackets in the tree encoding are encoded with a fixed length.

To reconstruct the offset lists from the Huffman coding, we first need to decode the Huffman tree. With this tree, the bit string can then be deciphered and, using the control codes, correctly formatted in the offset lists.

Table 4: Final compression of global offset list

Tree	0 0 0 1 [*EOL*] 1 [4] 1 [1] 0 1 [5] 0 1 [55] 1 [60]	
	Offset list	**Ctrl code**
TS	111 10 10 01 110 01 111 001	000

Table 5: Final compression

Tree	0 1 [1] 0 0 0 1 [*EOL*] 1 [3] 0 1 [4] 1 [5] 0 1 [2] 0 1 [0] 1 [*EOTS*]		
	First TS	**Offset list**	**Ctrl code**
S1	1110	[0 1011] [110 0] [0 1010]	1111
S2	1110	[0 1011] [110 0] [0 0] [110 0] [0 0]	1111
S3	1110	[1001 0] [1010 0] [1001 0]	1111
S4	1110	[110 110] [1001 0] [110 0]	1111
S5	0	[1011 0] [1010 0]	1000

VIII. IMPORTANCE DETERMINER

We now turn our focus on Eq. (1), and specifically how data is deemed important in MAKER, along with how the rest of the system is designed to handle important data. As detailed in Section IV, we utilise a binary importance measure to determine importance in order to separate the data points of a ts into two sets, $\mathcal{P}_I$ and $\mathcal{P}_U$. This section details a mechanism for determining importance based on whether a data point is an outlier.

The importance detection mechanism in the MAKER architecture is responsible for prioritising the preservation of essential information in the data stream as it is processed. This mechanism evaluates the importance of each data point using a combination of Welford's online algorithm and Z-score to identify outliers. The combination of Welford's online algorithm and Z-score was chosen for outlier detection in MAKER due to their combined capability of efficient real-time processing, numerical stability, and accurate relative positioning of data points, all while maintaining the flexibility to tailor the process according to specific needs using a simple threshold. By evaluating the importance of the data points, it enables the system to properly adjust the error bounds of the models to fit within the set data budget. The MAKER architecture also offers the flexibility to easily integrate custom importance algorithms, allowing users to tailor the importance detection mechanism to suit their specific needs. As discussed in Section IV, the only requirement for the algorithm is that it returns a binary value describing the importance of the data point. This extensibility enables more accurate prioritisation of information in various categories of data streams.

A. Welford's online algorithm

Welford's online algorithm[29] is a single-pass method for calculating the mean and variance of a stream of data. Single-pass algorithms for calculating variance can become very unstable if the variance is small relative to the square of the mean and can in some cases lead to a phenomenon known as catastrophic cancellation[14]. Welford's algorithm is designed to be numerically stable and still efficient, which makes it suitable for processing and calculating variance in data streams in real time. The algorithm is initialised with a mean (μ) value of 0, a variance accumulator (*S*) of 0, and a count (*n*) of 0. As each data point (v_k, t_k) is streamed, the algorithm updates the mean, variance accumulator, and count accordingly. Welford's algorithm is shown in Algorithm 3.

Algorithm 3 OnlineWelford

```
Input: Data point dataPoint,
       Number of data points n,
       Running mean mean,
       Running sum of squares of differences from the mean
       S
Output: The variance of the data stream
 1: $n \leftarrow n + 1$
 2: $\delta \leftarrow dataPoint - mean$
 3: $mean \leftarrow mean + \frac{\delta}{n}$
 4: $\delta_2 \leftarrow dataPoint - mean$
 5: $S \leftarrow S + \delta \cdot \delta_2$
 6: if $n > 1$ then
 7:     $variance \leftarrow \frac{S}{n-1}$
 8: else
 9:     $variance \leftarrow 0$
10: end if
11: return variance
```

B. Z-score

The Z-score is a statistical measure that describes a data point's relative position to the mean in terms of standard deviations. A Z-score of 0 indicates that the data point is equal to the mean, while a positive or negative Z-score represents the number of standard deviations above or below the mean, respectively. The Z-score for a data point (v_k, t_k) is calculated using the following formula:

$$Z_k = \frac{v_k - \mu}{\sigma} \tag{4}$$

where μ is the mean, and σ is the standard deviation (which can be obtained by calculating the square root of the variance). We combine the two concepts of Welford's online algorithm and the Z-score to be able to efficiently find outliers. The process is shown in Algorithm 4, where v_k is denoted *dataPoint*.

Algorithm 4 Z-score Based Outlier Detector

```
Input: Data stream of incoming data points $D$, threshold $\delta$
    Initialisation:
 1: $n \leftarrow 0$ {Number of data points}
 2: $mean \leftarrow 0$ {Running mean}
 3: $S \leftarrow 0$ {Running sum of squares of differences from the mean}
 4: $zScore \leftarrow 0$ {Z-score for the latest data point}
    Looping process:
 5: while $D$ is not empty do
 6:     $dataPoint \leftarrow D$.Current
 7:     $variance \leftarrow$ OnlineWelford($dataPoint$, $n$, $mean$, $S$)
        {Algorithm 3, parameters passed by reference}
 8:     $stdDev \leftarrow \sqrt{variance}$
 9:     if $stdDev \neq 0$ then
10:         $zScore \leftarrow \frac{dataPoint - mean}{stdDev}$
11:     else
12:         $zScore \leftarrow 0$
13:     end if
14:     if $zScore > \delta$ then
15:         Mark $dataPoint$ as an outlier
16:         $mean \leftarrow dataPoint$
17:         $n \leftarrow 1$
18:     end if
19: end while
```

Example 7. To demonstrate the outlier detection process, we consider the *ADC calibration* ts from Example 1. We will use Algorithm 4 to process the data stream and identify the outliers. The data points are streamed into the algorithm one by one, and the mean, variance, and Z-score are updated accordingly. In Fig. 3, the data points are plotted on a graph, with the Z-scores for each data point displayed next to them. We set a Z-score threshold of 2, and any data point with a Z-score greater than 2 or less than -2 is considered an outlier. In this example, the data point at timestamp 300 $(2, 180)$ is identified as an outlier. When this outlier is detected, the Z-score algorithm is reset, using the outlier as the first point, as shown in line 16-17 in Algorithm 4.

C. Outliers importance

Once the outliers are identified, the importance detection mechanism temporarily adjusts the error bound when a data point in the ts is deemed important, allocating more resources to the outliers while still adhering to the data budget constraints. We do this to find a satisfactory solution to the minimisation problem posed in Eq. (3). The updated error bounds are then used for compression, minimising information loss and maximising the utility of the compressed data. A cooldown mechanism is implemented in this process, which is activated after adjusting the error bounds. During the cooldown, no further modifications to the error bounds are allowed, thereby ensuring that the context, which includes the following data points, is compressed at a higher resolution while reducing the possibility of erratic adjustments.

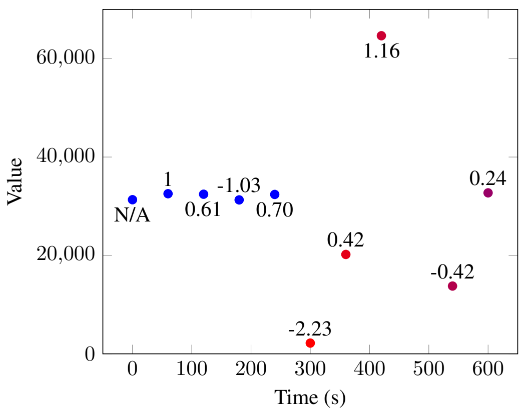
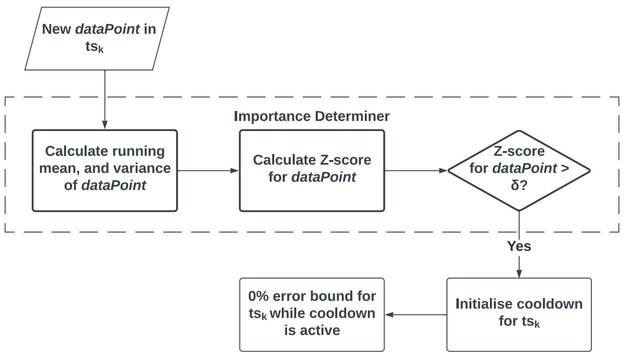


Figure 3: Outlier detection of *ADC calibration* ts

Example 8. Fig. 3 illustrates the cooldown mechanism with the red to blue fading data points, starting from the outlier detected at timestamp 300.

By introducing this cooldown period, MAKER can maintain a balance between adapting to the changing importance of data points while preserving some stability in the compression process. Fig. 4 outlines the complete process. The dashed lined box shows the *Importance Determiner* component, and the rest shows how the component is integrated into MAKER. It is also illustrated that the *Importance Determiner* component merely needs to return whether an incoming data point is an outlier or not. With this information, MAKER will trigger the cooldown mechanism if an outlier is detected.



New dataPoint in ts_k
Importance Determiner
Calculate running mean, and variance of dataPoint
Calculate Z-score for dataPoint
Z-score for dataPoint > δ?
Yes
Initialise cooldown for ts_k
0% error bound for ts_k while cooldown is active

Figure 4: Flowchart of the *Importance Determiner* component

IX. CHUNK-BASED SCHEDULER

In this section, we describe the *Chunk-based Scheduler* component *(CBS)* utilised by MAKER. The chunk-based mechanism divides the data stream into chunks and processes each chunk individually, allowing for efficient data management and error bound control. The main idea behind the chunk-based approach is to adapt the error bounds according to Θ (as defined in Eq. (3)). For this approach to work, we need to reformulate Θ (which is an upper bound for an entire ts) to an upper bound for each chunk. By defining the upper bound on a per-chunk basis instead, we achieve a new threshold value that can be used by the *Chunk-based Scheduler*. We call this new upper bound the $\mathbf{Budget}$, and the relationship between $\mathbf{Budget}$ and Θ is shown in Eq. (5).

$$\sum_{\gamma \in \Gamma} \boldsymbol{Budget} = \Theta \tag{5}$$

This ensures that the most important information is preserved within each chunk. We can use this approach to approximate a solution to Eq. (3), as we allow models to span over multiple chunks. The chunk-based approach thus serves as an approach for monitoring the bitrate continuously to adjust the error bounds as a means to find a solution for the entire ts rather than a way to strictly secure compliance of the budget within each chunk.

A. Deciding error change

The *CBS* relies on a set of parameters.

- $\mathbf{ChunkSize}$ ρ (see Definition 8): How often error bounds should be evaluated and models transmitted.
- $\mathbf{Budget}$ (see Eq. (5)): Number of bytes available for transmission of each chunk.
- $\mathbf{Buffer}$: Number of bytes of the accumulated $\mathbf{Budget}$ that has not been utilised.
- $\mathbf{BufferGoal}$: The size of the buffer the *CBS* strives on keeping at all times.
- $\mathbf{Rigorousness}$: In how many chunks the *CBS* needs to adapt to the $\mathbf{BufferGoal}$.
- $\mathbf{MaxAge}$: Maximum number of time units a model can span before it is terminated.

Example 9. Fig. 5 shows five *chunks*, with only PMC-Mean models. Swing and Gorilla are being left out for the sake of simplicity. Each square denotes a model which is finished, with a size of 32 bits. The $\mathbf{ChunkSize}$ is set to 120 seconds, which means that every 120 seconds, a $\mathbf{Budget}$ of 70 bits is allowed. Note that the $\mathbf{Budget}$ and $\mathbf{BufferGoal}$ are different parameters, however, both are 70 bits in this example.

This $\mathbf{Budget}$ can be utilised or saved for later if too few models are finished in the respective chunk. When a chunk ends, a decision is made on whether to change the error bounds or if the default error bounds are adhering to the set $\mathbf{Budget}$. In order to make this decision, linear regression is applied to the $\mathbf{Buffer}$ size of a number of trailing chunks, called $\mathbf{RegressionLength}$. This is done to ensure that chunks that contain a significant number of models will not increase the error bound accordingly. Instead, the decision is based on a trend.

Example 9. ***(continued)*** Fig. 5 shows this approach, where the $\mathbf{Buffer}$ of each of the five chunks is used. The prognosis shows the $\mathbf{Buffer}$ size trends away from the $\mathbf{BufferGoal}$, and, for this reason, the error bounds need to be increased in order to save space. This process is described in Section IX.B. If the prognosis trended toward the $\mathbf{BufferGoal}$, then the level of $\mathbf{Rigorousness}$ would decide whether the error bounds should be adjusted.

$\mathbf{Rigorousness}$ denotes the number of chunks in which the $\mathbf{Buffer}$ should reach the $\mathbf{BufferGoal}$, and if the trend does not favour the $\mathbf{Rigorousness}$, the error bounds are adjusted accordingly. The *CBS* can also force models to finish if they reach a certain age, which is denoted by $\mathbf{MaxAge}$. This is useful in cases where models become so long that they will never be transmitted. When a buffer becomes negative, it would be impossible to transmit models. Therefore they are kept in memory, while error bounds are increased in order to attempt to save enough buffer to send them. The models are kept for a period if the $\mathbf{Buffer}$ is too small to transmit them, and eventually, a decision as to whether discard the models or exceed the budget has to be made.

B. Adjusting error

To intelligently adjust error bounds, the bitrate of each ts is stored, denoting how much each ts impacts the cost. A moving weighted mean of the bitrate of the models on the trailing n data points represents the cost of the individual ts. The moving weighted mean is defined in Eq. (6), where x_t corresponds to the bitrate of a model at timestamp t and $|m_t|$ to the length of the sequence a model represents at timestamp t.

$$TB_t = \frac{\sum_{i=0}^{n-1} |m_{t-i}| x_{t-i}}{\sum_{i=0}^{n-1} |m_{t-i}|} \tag{6}$$

When error bounds need to be increased due to shortage of space, as described in Section IX.A, a number of tss (c) with the highest TB are marked as potential candidates for increased error bound. Two concurrent model compression processes are run on all of the potential candidates over the course of the next chunk

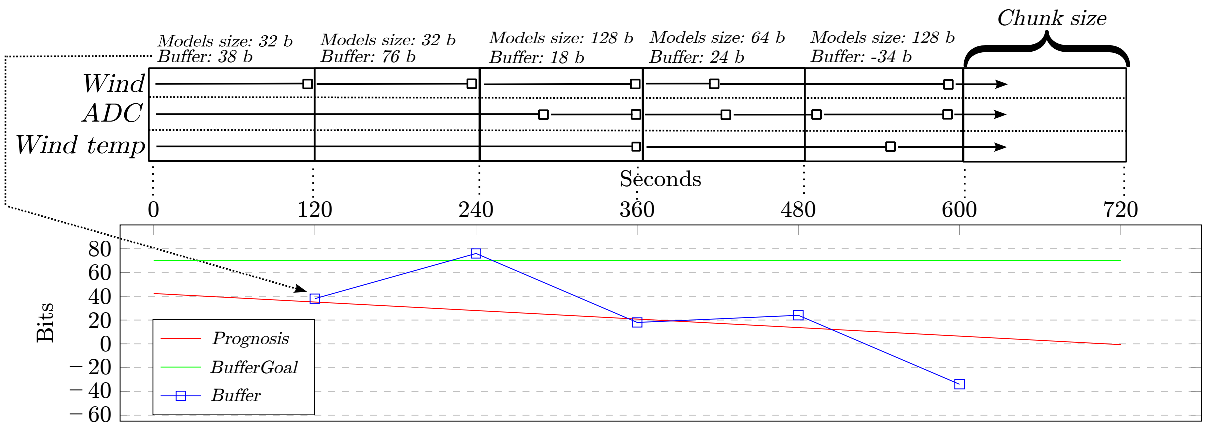


Figure 5: Chunk-based scheduler overview

using the default error bound and the maximum error bound. The concurrent compression enables evaluation of the increased error bound compared to the default error bound. The candidates with the largest savings are prioritised when determining which tss should be compressed using the max error bound. Based on the $\mathbf{Buffer}$ and $\mathbf{BufferGoal}$, the $\mathbf{Savings}$ needed are calculated as $\boldsymbol{Savings} = \boldsymbol{BufferGoal} - \boldsymbol{Buffer}$. Candidates are then chosen until the $\mathbf{Savings}$ requirements are met. The chosen candidate tss will have their error bound increased in the newly finished chunk, whereas the rejected candidate tss remain unchanged. This process also helps determine how many tss (i.e. c) should be marked as candidates next time the error bounds should be increased. If the savings obtained through these candidates was not enough to meet the $\mathbf{Savings}$ requirements, the number is increased and vice versa.

The mechanism for decreasing error bounds works much in the same way as increasing error bounds. However, the candidates are chosen based on the lowest TB to enable as many tss as possible to be decreased. The error bounds are always decreased to a 0 error bound, which means that they are lossless. The selection of the candidates is based on those which consume the most bytes, which is the opposite of what happens when the error bounds should be increased. It is important to note, that this process is different from the process described in Section VIII, as that component decreases the error bounds immediately and over the course of a cooldown period.

X. EVALUATION

A. Metrics

This section evaluates the performance of MAKER. The metrics that have been used for this evaluation are described in the following. See Section IV for notation details.

- $\mathit{ModelSize}\ (\mathit{bytes}) =$
$$\sum_{\mathcal{M}\in\mathcal{T}}\ \sum_{(m,\epsilon)\in\mathcal{M}} S(m) \tag{7}$$
This metric refers to the total size of the model representations for all models across all tss in the data set, not including the size of the timestamps. It reflects the data transmission cost, as larger models require more data to be transmitted. This metric is only used to define *CompressionRatio*.
- $\mathit{TimestampSize}\ (\mathit{bytes}) =$
$$\sum_{\mathcal{M}\in\mathcal{T}}\ \sum_{(m,\epsilon)\in\mathcal{M}} S(t(m)) \tag{8}$$
This metric refers to the total size of only the compressed timestamps across the entire data set.
- $\mathit{CompressionRatio}\ (\mathit{ratio}) =$
$$\frac{S(TS)}{\mathit{ModelSize} + \mathit{TimestampSize}} \tag{9}$$
This metric denotes how much larger the original data is compared to the compressed data.
- $\mathit{WeightedErrorBound}\ (\%) =$
$$\sum_{\mathcal{M}\in\mathcal{T}}\ \sum_{(m,\epsilon)\in\mathcal{M}} \frac{|m|\cdot\epsilon}{|\mathcal{T}|} \tag{10}$$
where $|m|$ denotes the number of data points represented by model m.

  This metric represents the weighted average error bound across all data points in all tss in the entire data set.

- $AverageErrorBound$ (%) =

$$\sum_{(m,\epsilon)\in\mathcal{M}} \frac{|m|\cdot\epsilon}{|\mathcal{M}|} \tag{11}$$

This metric denotes the average error bound across all the models representing a single ts.

- $WeightedActualErrorI$ (%) =

$$\sum_{\mathcal{M}_I\in\mathcal{T}_I} \frac{\mathcal{E}(\mathcal{M}_I)\cdot|\mathcal{M}_I|}{|\mathcal{T}_I|} \tag{12}$$

for models containing important data points and

$WeightedActualErrorO$ (%) =

$$\sum_{\mathcal{M}_O\in\mathcal{T}_O} \frac{\mathcal{E}(\mathcal{M}_O)\cdot|\mathcal{M}_O|}{|\mathcal{T}_O|} \tag{13}$$

for models containing ordinary data points. These metrics correspond to the terms of our minimisation formula expressed in Eq. (3). They represent the average actual error observed after decompression and measure the deviation of the decompressed points from the original uncompressed data points.

- $ImportantPointsPercentage$ (%) =

$$\frac{|\mathcal{T}_I|}{|\mathcal{T}|}\cdot 100\% \tag{14}$$

This metric denotes the percentage of important data points across all tss.

B. Environment

The tests concerning memory usage and runtime performance (described in Section X.H) are performed on a BeagleBone with an AM335x 1GHz ARM® processor with 512MB memory. The operating system used is *Debian "Bullseye" Minimal Image*[22].

C. Data sets

This section describes the data sets used for our evaluation of MAKER. We use two data sets:

- Rover Environmental Monitoring Station Telemetry (*REMS*)[21]
- Gas sensor array temperature modulation (*GAS*) [3, 4]

The *REMS* system is part of the *Mars Science Laboratory* mission launched by NASA in 2011 and it features data recorded from weather conditions on Mars, such as wind speed, temperatures and humidity. We use the data from SOL 1160-1293. The sampling rate is 1 Hz, however, it is irregular due to gaps with missing values. The *GAS* data set contains measurements from metal oxide semiconductor (MOX) gas sensors in various conditions. This data set is highly irregular as a result of having a sampling rate of 3.5 Hz, which is rounded to millisecond accuracy. Table 6 describes the characteristics of both data sets. *TSs* denotes the number of tss excluding the timestamp column. For *REMS*, we have discarded the tss containing text values, so only tss with floating point data remain.

Table 6: Data sets used for evaluation

Data set	**TSs**	**Sampling frequency (Hz)**	**Size (MB)**
REMS	86	1	3,274.37
GAS	19	3.5	584.16

D. Evaluation parameters

We now evaluate the effects of each of the parameters described in Section IX. Table 7 shows the different values we test for each parameter. When modifying one or more parameters during evaluation, the default values for the rest of the parameters are written in bold. Because the timestamps of *GAS* are expressed in milliseconds, the $\mathbf{MaxAge}$ is scaled by $1,000$. The $\mathbf{Budget}$ is chosen based on a balance between error and *CompressionRatio*. $\mathbf{ChunkSize}$ is chosen to have approximately $1,000$ data points in each chunk in order to perform error bound adjustments often. $\mathbf{BufferGoal}$ is $10,000$ bytes in order the keep a small capacity for peaks in model creation. $\mathbf{RegressionLength}$ is set to 10 to enable a fast reaction to changes. $\mathbf{Rigorousness}$ is set to 10 in order to not delay the adjustments for too long. The Z-score threshold is set to 3.0, which is a common threshold for detecting outliers as explained by the empirical rule, which states that 99.7% of all observations in a normal distribution are within 3 standard deviations of the mean[25].

E. Timestamp encoding techniques

This section examines the effects of using Huffman coding for compressing the offset lists compared to two other commonly used techniques; *Arithmetic coding* and *Delta-delta encoding*. These are described in Section III.A.c..

In this test, we compare *TimestampSize* with the results of running Arithmetic coding on the offset lists. The overhead occurring from the frequency information is included in these results. The Delta-delta encoding was run on the raw timestamps. Figs. 6 and 7 show the results of these

Table 7: Parameter values used for evaluation. Default parameters are written in **bold**

Parameter	***REMS***	***GAS***
MaxAge *(time units)*	**1M**	**1,000M**
Budget *(bytes)*	40k, 50k, 60k, 70k, 80k, 90k, **100k**, 110k, 120k	32k, 36k, 40k, 44k, 48k, **52k**, 56k, 60k, 64k, 68k
ChunkSize *(time units)*	**1k**	**286k**
BufferGoal *(bytes)*	**10k**; 100k	**10k**
RegressionLength	**10**; 100	**10**
Rigorousness	**10**	
Default error bound	**5**	
Max error bound	**10**; 15	**10**
Z-score threshold (δ)	1; 1.5; 2.0; 2.5; **3.0**; 3.5; 4.0; 4.5; 5.0; 5.5	

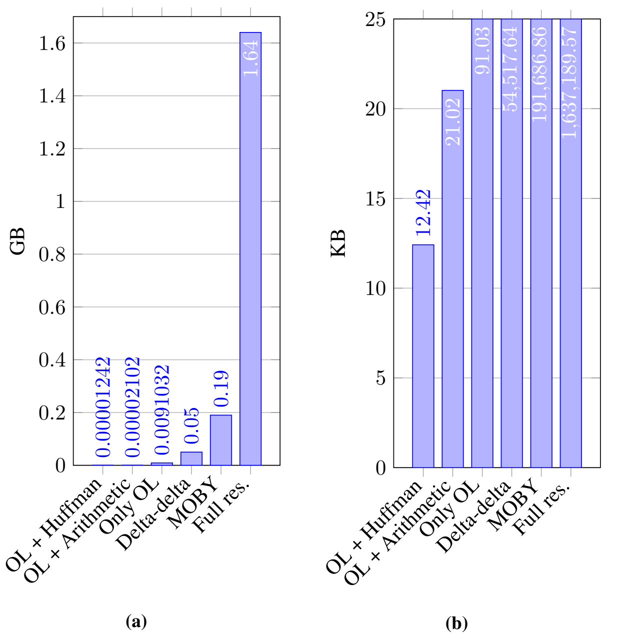


Figure 6: Timestamp compression for *REMS*. Fig. 6b is a scaled version of Fig. 6a. *OL* denotes *offset lists*.

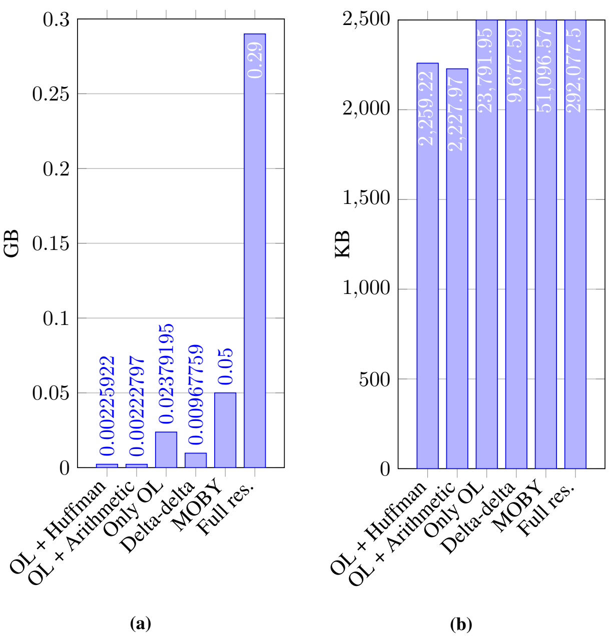
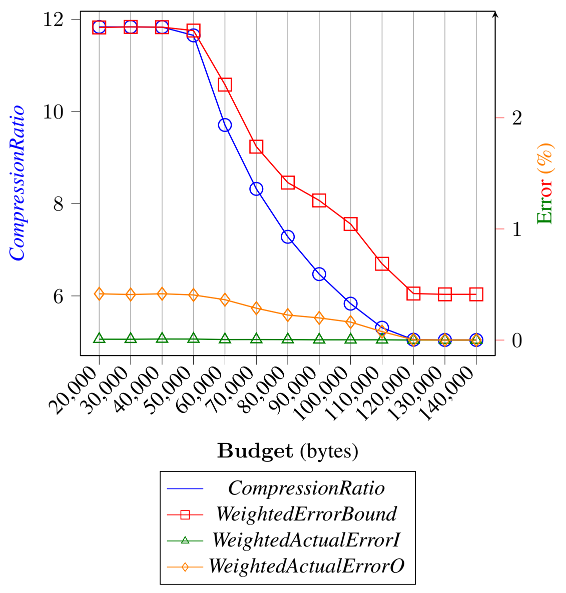


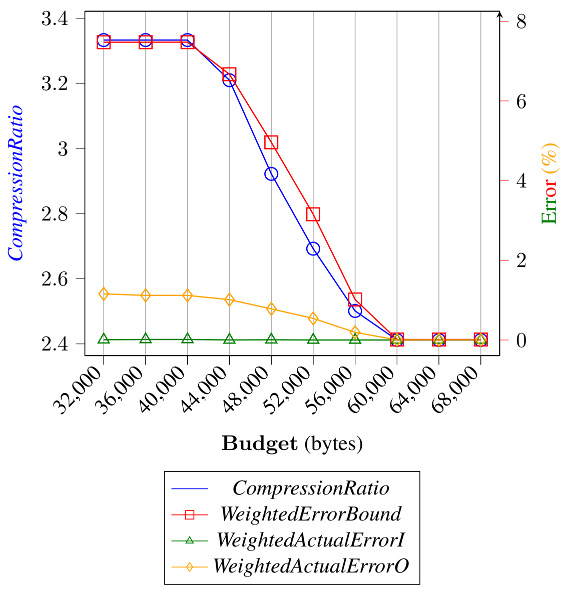
Figure 7: Timestamp compression for *GAS*. 7b is a scaled version of 7a. *OL* denotes *offset lists*.

compression techniques along with the results of compressing with MAKER's Huffman coding implementation (i.e. *TimestampSize*); the results of compressing timestamps with only the offset lists; the results of compressing using MOBY; and the full resolution sizes. Due to the difference in the sizes of the values, we provide two scalings for each data set. For *REMS*, the approach of MAKER offers the best compression, however, for *GAS*, a combination of offset lists and Arithmetic coding gives the best compression. On the *REMS* data set, MAKER provides a *CompressionRatio* of $131,818.8$ compared to storing the timestamps in full resolution and 8.5 compared to MOBY. For *GAS*, these ratios are 129.3 and 22.6 respectively. The reason for these improvements is the approach our *Timestamp Compressor* takes to handle timestamps. Where MOBY duplicates compressed timestamps for each ts, resulting in very high redundancy, the approach of MAKER is to reference the same timestamps across tss using the offset lists.

It is worth noting that *GAS* has a smaller full resolution size than *REMS*, however, the compressed timestamps are larger than those of *REMS*. This is due to the irregular nature of the *GAS* data set, which causes the offset lists to grow. This is seen in the *Only OL (offset lists)* bars in Fig. 7a when compared to the corresponding bar in Fig. 6a. However, the former also shows that the offset lists can be compressed efficiently using either Huffman coding or Arithmetic coding. This is because the offset lists contain a lot of very small values that occur frequently when timestamp intervals fluctuate. This is handled well by both the Huffman coding step, as short codes are assigned to frequently occurring values, and Arithmetic coding, as the sizes of the subintervals are based on the frequencies. This shows us that our approach is robust enough to not explode in size on irregular ts.



(a) *REMS*



(b) *GAS*

Figure 8: *CompressionRatio*, *WeightedErrorBound*, *WeightedActualErrorO*, and *WeightedActualErrorI*

F. Different **Budget** *Configurations*

The following results illustrate the trade-offs made by the compression system based on changing the **Budget** parameter.

a. The impact of **Budget**

The impact of the **Budget** on the *CompressionRatio*, *WeightedErrorBound*, *WeightedActualErrorI*, and *WeightedActualErrorO* was analysed for both *REMS* and *GAS*.

First, we analyse the *REMS* data set, shown in Fig. 8a. As the **Budget** increases from $20,000$ to $140,000$, the *CompressionRatio* decreases substantially, from 11.83 to 5.04. An examination of the *WeightedErrorBound* shows that it starts at 2.82 at a **Budget** of $20,000$ and significantly decreases to 0.41 when the **Budget** is raised to $140,000$. This demonstrates that a larger **Budget** permits the error bounds across the tss to be adjusted to more strict values, thereby reducing the *WeightedErrorBound* and forcing the system to compress with higher accuracy. The rate of decrease in *WeightedErrorBound* significantly slows down after a **Budget** of $120,000$, showing diminishing returns on the error bound reduction with further increases in the **Budget**.

The *WeightedActualErrorI* and *WeightedActualErrorO* results show a similar pattern to the error bounds. The *WeightedActualErrorO* decreases from 0.416 at a **Budget** of $20,000$ to 0.00052 at a **Budget** of $140,000$, while the *WeightedActualErrorI* is always zero. This is a significant feature of the system, demonstrating that it successfully prioritises the accuracy of important data points, keeping their error levels near zero, while ordinary data points are the ones that are mostly affected by the variations in the **Budget**. Similarly to the error bound, the rate of decrease in the *WeightedActualErrorO* slows down dramatically as the **Budget** exceeds $120,000$.

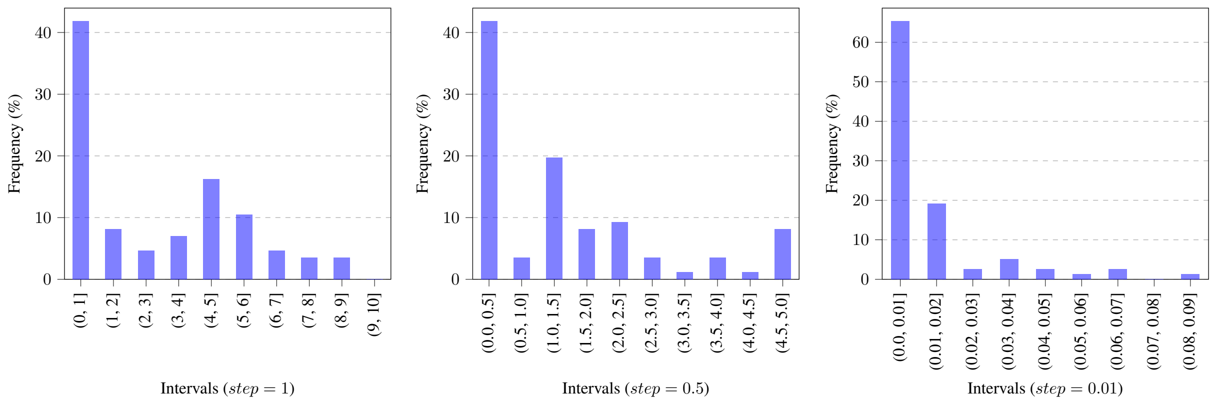
Next, we evaluate the impact of the **Budget** on the *GAS* data set, shown in Fig. 8b.

As the **Budget** increases the *CompressionRatio* decreases, starting at 3.33 at a **Budget** of $32,000$ and going down to 2.41 at a **Budget** of $68,000$. This behaviour is similar to what was observed in the *REMS* data set. *WeightedErrorBound* and *WeightedActualErrorO* decrease with an increasing **Budget**. Similarly to the *REMS* data, *WeightedActualErrorI* is at zero and remains stable, ensuring us that the system retains the critical information accurately. Additionally, the same diminishing returns can be seen on both the *CompressionRatio*, *WeightedErrorBound*, and *WeightedActualErrorO*, confirming that the **Budget** only affects the system in a certain range of values.

The observations from both data sets underscore the system's ability to maintain a lossless compression for important data points, thereby effectively managing the trade-off between *CompressionRatio* and data accuracy. However, determining the optimal **Budget** requires careful consideration of the specific application's needs and the limitations of the data transmission and storage capacity.

b. Distribution of *AverageErrorBound* across tss

Fig. 9 presents the distribution of the *AverageErrorBound* for different tss across the entire *REMS*, tested at three



(a) Budget 40,000 **(b) Budget** 80,000 **(c) Budget** 120,000

Figure 9: *AverageErrorBound* distributions of *REMS*

distinct **Budget** configurations: $40,000$, $80,000$, and $120,000$. The y-axis represents the frequency of occurrence of various intervals of the *AverageErrorBound* in the data set at the respective **Budget** configuration. The distributions provide an interesting insight into the overall performance of the compression system under varying **Budget** allocations. More specifically, they illuminate how the *AverageErrorBound* is distributed across the data set at each **Budget** level and how it evolves as the **Budget** increases.

- As demonstrated in Fig. 9a, at a **Budget** of $40,000$, the *AverageErrorBound* spans across a range of (0, 9]. Most frequently, it falls within the (0, 1] interval, though a considerable portion of tss experience larger error bounds. This variation shows that MAKER's performance can vary considerably under this **Budget**, with some tss achieving low error bounds while others experience higher ones.

- When the **Budget** is increased to $80,000$, as depicted in Fig. 9b, a significant shift in the distribution can be observed. Here, the majority (41.86%) of the tss have an *AverageErrorBound* in the $(0.0, 0.5]$ interval, signifying that the compression system can model a substantial portion of tss with minimal error at this **Budget** level. However, it is important to note that some tss still exhibit larger error bounds, indicating a balance must be struck between low error bounds and effective **Budget** management, especially for those tss that display more complex behaviours.

- Upon further increasing the **Budget** to $120,000$ (Fig. 9c), the *AverageErrorBound* significantly skews towards the lower end, with 65.38% of tss falling within the $(0.00, 0.01]$ interval. The shrinking range from $(0.0, 5.0]$ at a **Budget** of $80,000$ to $(0.00, 0.09]$ at a **Budget** of $120,000$ signifies a major improvement in the compression system's capability to reduce errors, even more so than the improvement observed from $40,000$ to $80,000$. This demonstrates that, with more resources at its disposal, the system is increasingly effective at representing the tss with high accuracy. One thing to note is that 2 of the ts had a higher *AverageErrorBound* of about 5, but these were excluded for clarity. The reason for their existence in this configuration is further explained in Section X.F.c..

The observed distribution alterations imply that the compression system improves its capability of assigning lower error bounds to all tss in the data set with a higher **Budget**. This provides a quantitative understanding of the **Budget** parameter's influence and showcases that a larger **Budget** contributes to a more consistent and precise compression throughout the entire data set.

However, it is important to recognise the concept of diminishing returns beyond a certain threshold. Despite the dramatic improvement from $80,000$ to $120,000$, it is unlikely that this rate of error reduction will continue indefinitely with further increases in **Budget**, but this is heavily correlated with the behaviour of future data. Further reductions are not only less likely but may also be less meaningful in practical terms due to the already high precision achieved.

c. The maximum *AverageErrorBound* and *AverageActualError* across increasing budgets

The *AverageErrorBound* and *AverageActualError* metrics provide an additional perspective on the compression system's performance. These metrics elucidate how the tss within the data set are treated by the system, specifically how the "worst-case" performance of the algorithm changes as the budget increases.

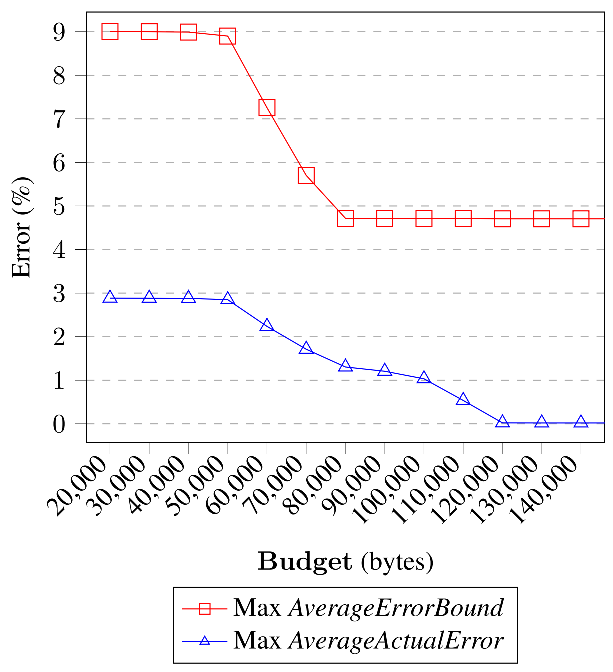


Figure 10: Maximum *AverageErrorBound* and Maximum *AverageActualError* of *REMS*

In Fig. 10, the following is observed:

- *Max AverageErrorBound*: This metric represents the highest *AverageErrorBound* across all tss for different $\mathbf{Budget}$ configurations. As expected, the maximum *AverageErrorBound* across all tss decreases with an increasing $\mathbf{Budget}$. This shows that a larger $\mathbf{Budget}$ allows the algorithm to allocate smaller error bounds, leading to more accurate model representations across all tss. An intriguing observation, however, is the minimal decrease in the maximum *AverageErrorBound* beyond a $\mathbf{Budget}$ of $100,000$. This is indicative of a point of diminishing returns, where further increases in the $\mathbf{Budget}$ contribute little towards "worst-case" model precision.
- *Max AverageActualError*: This metric represents the highest *AverageActualError* across all tss for different $\mathbf{Budget}$ configurations. As the $\mathbf{Budget}$ increases, the maximum *AverageActualError* decreases substantially. Notably, the interval between a $\mathbf{Budget}$ of $110,000$ and $120,000$ shows a disproportionate reduction in the maximum *AverageActualError*.

The maximum *AverageActualError* does not mirror the trend of the *AverageErrorBound*. This is because there are periods in the data stream where there is not enough $\mathbf{Budget}$ to cover the extreme peaks in bitrate, which causes a few of the ts to increase dramatically in error bound. Since these results are based on the *AverageErrorBound*, the maximum *AverageErrorBound* is heavily skewed by these few ts.

Although these maximum values represent the "worst-case" scenarios across all tss in the data set, we do observe in Fig. 9 that most tss experience significantly lower *AverageErrorBound*. The maximum values, however, provide insights into the potential extent of the error that the algorithm might exhibit for some tss in the data set.

The minimum *AverageErrorBound* was not included in this plot as it was always at or very near 0 due to the presence of at least one static ts in the data set.

G. Outliers

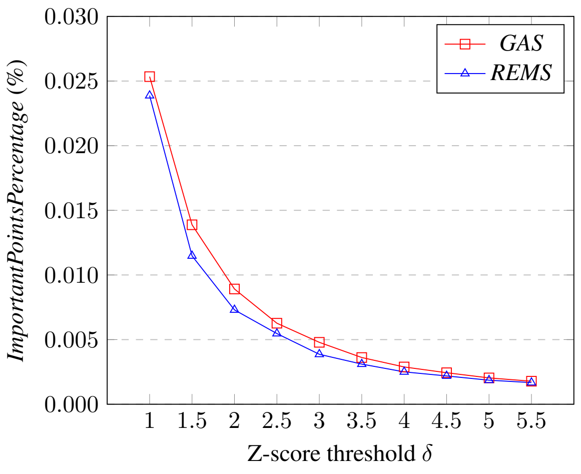


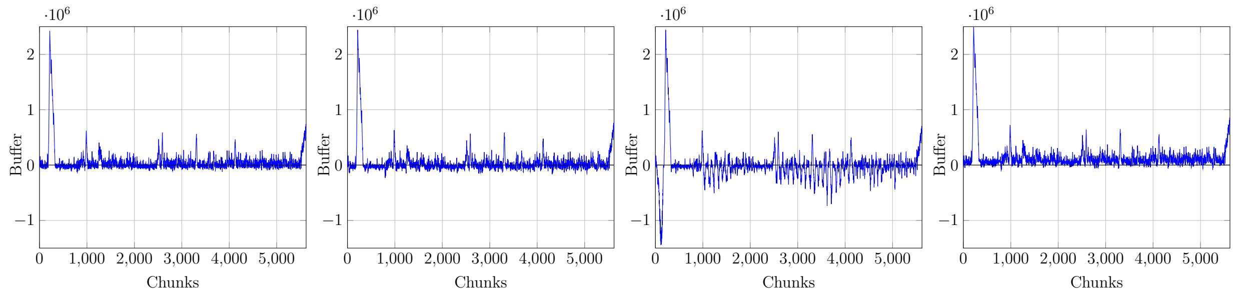
Figure 11: Percentage of important points of *GAS* and *REMS*

We now evaluate MAKER on the *ImportantPointsPercentage* metric. Figure 11 illustrates an inverse relationship between the Z-score threshold and the proportion of identified important points (outliers) within the data set. Specifically, we observe a trend where an increase in the Z-score threshold results in a decrease in the proportion of outliers. This decrease is exponential, suggesting that most data points in the distribution are within close proximity of the mean, and outliers become progressively rarer as we increase the threshold.

At each Z-score threshold, the proportion of outliers in the *GAS* data set is consistently higher than that in the *REMS* data set. This implies that the *GAS* data set possesses greater variability or more extreme values. Understanding these differences can provide insights for fine-tuning the Z-score threshold depending on the data set's characteristics.

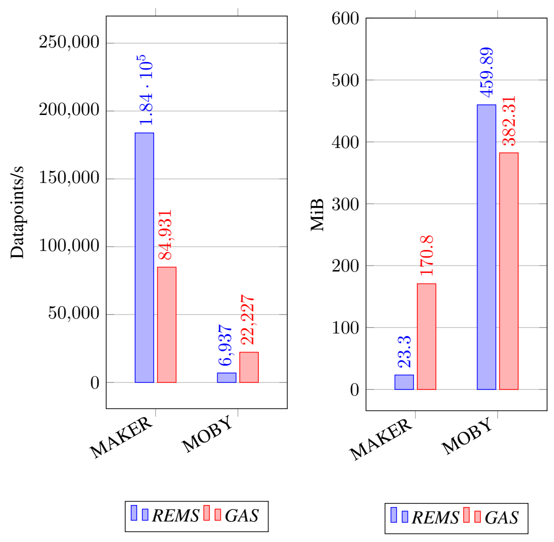
H. Performance

Fig. 13a shows significant improvements in processing rate as opposed to MOBY. On the *REMS* data set, MAKER compresses data points 26x faster with a processing rate of $184,000$ data points each second. However, the processing rate is decreased on the very irregular



(a) MAKER chooses candidates
(b) Max number of candidates
(c) MAKER chooses candidates + **RegressionLength** 100
(d) MAKER chooses candidates + **BufferGoal** 100,000

Figure 12: Development of buffer - *REMS*



(a) Processing rate
(b) Memory usage

Figure 13: Performance

data set *GAS* at a processing rate of 84,000 data points each second. This is due to the creation of and the occasional internal searches of the global offset list, which impacts the processing rate negatively.

Fig. 13b shows that memory impact on the *REMS* data set is vastly reduced compared to MOBY, where MAKER is using only 5% of the memory required by MOBY. The memory impact increases on the *GAS* data set because the full resolution timestamps and Gorilla models are exploding in size due to the irregularity of the data set.

I. **Buffer** *dynamics*

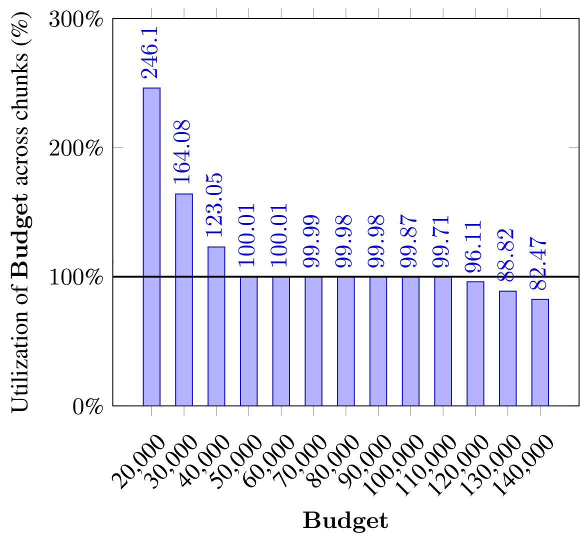
Fig. 12 shows the development of the **Buffer** throughout the runtime of the system. Fig. 12a shows how the **Buffer** behaves when MAKER chooses which tss should be candidates for error bound changes and Fig. 12b shows the case where the maximum number of candidates are chosen each time, e.g., the system is able to evaluate savings on all tss. The behaviour of the **Buffer** is almost identical, which demonstrates that MAKER chooses the right candidates in the majority of the cases. Fig. 12c reveals that when the **RegressionLength** is increased MAKER has a large negative **Buffer** several times, which means that the error bounds are lowered too much because the system reacts slower compared to a lower **RegressionLength**. It is essential for the system to keep the **Buffer** as close to the **BufferGoal** as possible at all times, and Fig. 12d shows that when the **BufferGoal** is increased to $100,000$, the **Buffer** is shifted upwards, leaving few cases with a negative **Buffer**. All of the figures show a positive spike, which is the result of an extended period of very limited bitrate across the majority of tss. The low bitrate results in a build-up of the **Buffer**, which can be utilised when possible to bring the **Buffer** down to the **BufferGoal**.

J. **Budget** *utilisation*

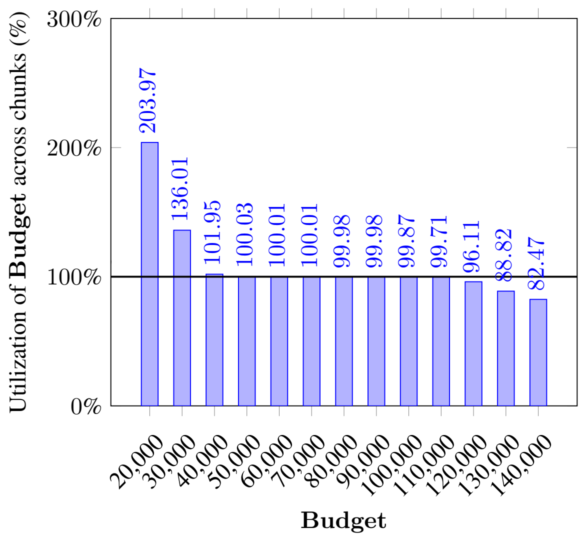
In the following evaluation, we analysed the system's **Budget** utilisation at different **Budget** levels to understand its effectiveness in prioritising important data and maintaining a balance between data accuracy and **Budget** constraints. The **Budget** utilisation results are computed as a percentage of the given **Budget** and are presented in Fig. 14.

From the results in Fig. 14a, we observe that for lower **Budget** configurations ($20,000$ to $40,000$ bytes), MAKER exceeds the available **Budget**, utilising up to 246.1% for a **Budget** of $20,000$ bytes. This situation represents the critical case mentioned in the Section IV, where the first condition of equation Eq. (3) cannot be met due to the data being unable to be compressed to a sufficiently small size that complies with the upper bound. However, MAKER was designed to handle such cases by prioritising data points deemed important through intelligent error bounds adjustment, thus preserving the most critical information even when the **Budget** is exceeded.

As we increase the **Budget** to $50,000$ bytes, we see that the **Budget** utilisation stabilises to around 100%, indicating that MAKER can effectively utilise the given



(a) Max error bound 10%



(b) Max error bound 15%

Figure 14: **Budget** usage by *ModelSize* of *REMS*

Budget. This observation is consistent up to a **Budget** of $90,000$ bytes, demonstrating that MAKER is capable of effectively managing resources, ensuring the data quality and maintaining **Budget** constraints according to Eq. (3).

As we increase the **Budget** further to $120,000$ bytes and beyond, the **Budget** utilisation starts to decrease, falling to 82.47% for a **Budget** of $140,000$ bytes. As we could see in Section X.G, the amount of data points deemed important is less than a 0.025%, which indicates that the system has a sufficiently large amount of resources to allocate more to not just the important data points, but many of the ordinary data points as well. In these cases, the system does not need to use the full **Budget** as it is already achieving a high data quality, even while complying with the constraints presented in Eq. (3).

When increasing the max error bound to 15%, shown in Fig. 14b, the utilisation remains virtually unchanged on the **Budget** configurations that stay within the **Budget** at 10% error bound. This is due to MAKER not needing the increased max error bound. However, when looking at smaller **Budget** configurations, their utilisation is decreased as they can make use of the extra error bound. The **Budget** configuration of $40,000$ almost stays within the **Budget**, whereas the $20,000$ and $30,000$ **Budget** configurations are decreased compared to 10% error bound.

To summarise, these results indicate that MAKER effectively utilises the **Budget**, adjusting its strategy dynamically based on the available resources. When resources are constrained, it prioritises the most important data points, ensuring the best possible data quality within the limited given **Budget**. When more resources are available, it can afford to decrease the error bounds, providing higher data quality across the board while still maximising the amount of **Budget** utilised. Therefore, the results demonstrate that MAKER can effectively balance the trade-off between data accuracy and **Budget** constraints, making it a viable solution for the problem presented in Section IV.

XI. DISCUSSION

In this section, we discuss some thoughts, ideas and considerations regarding the functionality and performance of MAKER.

A. *Exceeding the* **Budget**

The flexibility of MAKER could be improved by having options when confronted with exceeding the budgetary constraints. Users could tailor the response of the system based on their needs - either by discarding older data models or exceeding the established **Budget**. This adaptability would enhance MAKER's versatility and attempts to accommodate a myriad of scenarios.

Balancing between the freshness of data and ensuring its continuity is a dilemma. If the **Budget** is exceeded, the models not transmitted yet are stored in memory as explained in Section IX. When selecting which models to transmit first in the next chunk, MAKER could be equipped with two potential strategies: 'First In, First Out' (FIFO), where the oldest models are discarded, or focusing on the newest models, thereby always prioritising the most recent data. Each approach has its merits, contingent on the specific requirements of the user. The FIFO method ensures a continuous data stream, critical for understanding trends and changes over time. Conversely, retaining the newest models provides the most current snapshot of the data, essential for real-time response and decision-making. This approach might lead to gaps in the data, but the trade-off is access to the most recent and pertinent information. The most optimal strategy depends on individual needs: Timely information necessitates prioritising the latest models while understanding longer trends favours the FIFO approach.

A hybrid approach might serve best, where decisions are made considering both the model's age and its importance. An additional factor to consider could be the bitrate, where models with a smaller bitrate should be prioritised in order to preserve and transmit the maximum amount of data points.

In summary, MAKER strives for data accuracy, **Budget** adherence, and user-driven adaptability. Future efforts could delve deeper into exploring additional strategies to further fine-tune this decision-making process, thereby enhancing the system's robustness and flexibility.

B. Performance and memory improvements

Regarding compression and transmission systems, a balance must be struck between processing speed, memory footprint, and data integrity. MAKER has been designed to handle these constraints, presenting substantial improvements over its predecessor, MOBY, in both performance and memory usage.

As shown on Fig. 13a, the improved performance offers a large processing speed increase. Not only does this mean faster processing times, but it also translates into less energy consumed by the system. In an environment where other tasks might compete for CPU usage, or in the Mars rover's case, where energy must be consumed as sparsely as possible, the speed and efficiency of MAKER become crucial.

Simultaneously, we have also made substantial strides in reducing the system's memory footprint. This improvement translates into a system that is not only more lightweight and efficient but also more compatible with other applications sharing the same resources. Decreased memory usage reduces the likelihood of system crashes due to memory overflow, particularly when processing larger, complex, or highly irregular data sets.

These improvements do not operate in isolation but have a synergistic effect with other parts of MAKER as well. The less memory the system uses, the more room there is to choose a higher **Buffer**, allowing for more data models to be stored. This is tied directly to our ability to manage the data transmission **Budget** better, giving the user more flexibility when choosing between discarding old models or exceeding the **Budget**.

By enhancing processing speeds and reducing memory usage, we have created a system that promises greater stability and resource efficiency, all within the constraints of real-world scenarios.

C. Considering irregularity as an importance metric

While this paper has discussed at length the potential significance of irregularities in time series data, we have not formally addressed irregularity as an importance metric, as outliers have been. Incorporating irregularity as an importance metric could potentially add another layer of sophistication to our system, helping us identify more important data points and improve prioritisation of the data transmission. The presumption that irregularity could be a sign of importance has already been established in the paper, as irregular data points often signal a departure from the norm and therefore could contain valuable information.

However, there are several challenges and considerations associated with this approach. For one, how do we objectively measure irregularity, and how do we determine which irregular data points are more important than others? While the Z-scores used in the *Importance Determiner* component provide a measure of how far a data point deviates from the mean, this may not directly translate to some hypothetical measure of irregularity. More research would need to be done to develop an effective metric for irregularity, and it might require different techniques depending on the specific characteristics of the data set.

A critical case to consider is when data sets are inherently irregular. In such cases, irregularity would be the norm rather than the exception, and it could be challenging to distinguish important irregularities from ordinary ones. We would need to determine how to measure relative irregularity within a data set and how to adjust our importance scoring accordingly.

In conclusion, while incorporating irregularity as an importance metric could potentially enhance our system's ability to identify and prioritise important data points, it would also introduce several complexities and challenges that would need to be carefully addressed. Future work could explore this idea in more detail, investigating how an irregularity metric could be implemented and what impact it might have on the overall performance and functionality of our system.

XII. CONCLUSION

In this paper, we have detailed the design and capabilities of MAKER, a time series compression and transmission system. This system improves upon the groundwork set by the model-based compression system MOBY[12], enhancing it by introducing the ability to adapt error bounds in real time according to a user-specified data budget.

At its core, MAKER differentiates between important and ordinary data points, concentrating on maintaining low error bounds for the significant data. It also identifies areas of the data set where larger error bounds might lead to considerable savings, further enhancing its ability to adhere to budget constraints. A key feature is the preservation of timestamps in their original form instead of interpolating/discarding them, as data irregularity can denote importance, and preservation of the timeline at which data arrived decreases the risk of inconclusive data analysis.

This is made possible through three core components. The *Importance Determiner* component employs Welford's online algorithm and Z-scores to identify outliers and mark important data points. The *Chunk-based Scheduler* component then segments the data stream into manageable chunks, facilitating continuous monitoring and intelligent adjustment of error bounds. Finally, the *Timestamp Compressor* component, through a blend of offset lists constructed with run-length encoding on the offsets and Huffman coding to further enhance the compression ratio, provides lossless compression for timestamps and thereby captures the irregularity.

Our evaluation demonstrates that MAKER generally operates within the defined data budget by intelligently adjusting error bounds, only faltering in extreme cases where budgets are unrealistically low based on the incoming data. In cases where there is a superfluous budget, the system will maximise the data accuracy as much as possible across the data set, such that the budget is utilised to its full potential. On the performance and memory front, MAKER exhibits an impressive improvement over MOBY. These combined improvements enhance MAKER's adaptability, resilience, and suitability for diverse scenarios, especially in remote systems where resource constraints are a common reality.

While there is always room for enhancement and additional fine-tuning, MAKER represents a step forward in managing the delicate balance between data integrity, resource usage, and adherence to data budget constraints. Future work might explore additional strategies and models to further refine the system's capabilities.

XIII. ACKNOWLEDGEMENT

We thank Torben Bach Pedersen for his time and supervision. We also thank Cobham SATCOM for their time, sharing knowledge, and for lending equipment for test, and lastly, we thank NASA for elaborating on the *REMS* data set.

References

[1] Kiriakos Alexiou, Efthimios G. Pariotis, and Helen C. Leligou. "Sensor Data Quality in Ships: A Time Series Forecasting Approach to Compensate for Missing Data and Drift in Measurements of Speed through Water Sensors". In: *Designs* 7.2 (2023). ISSN: 2411-9660. DOI: `10.3390/designs7020046`. URL: `https://www.mdpi.com/2411-9660/7/2/46`.

[2] Jason Brownlee. *How To Resample and Interpolate Your Time Series Data With Python*. Accessed on 04/10/2023. Dec. 2016. URL: `https://machinelearningmastery.com/resample-interpolate-time-series-data-python`.

[3] Javier Burgués, Juan Manuel Jiménez-Soto, and Santiago Marco. "Estimation of the limit of detection in semiconductor gas sensors through linearized calibration models". In: *Analytica Chimica Acta* 1013 (2018), pp. 13–25. ISSN: 0003-2670. DOI: `https://doi.org/10.1016/j.aca.2018.01.062`. URL: `https://www.sciencedirect.com/science/article/pii/S0003267018301673`.

[4] Javier Burgués and Santiago Marco. "Multivariate estimation of the limit of detection by orthogonal partial least squares in temperature-modulated MOX sensors". In: *Analytica Chimica Acta* 1019 (2018), pp. 49–64. ISSN: 0003-2670. DOI: `https://doi.org/10.1016/j.aca.2018.03.005`. URL: `https://www.sciencedirect.com/science/article/pii/S0003267018303702`.

[5] Giorgio Buttazzo and Luca Abeni. "Adaptive rate control through elastic scheduling". In: *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*. Vol. 5. Dec. 2000, 4883–4888 vol.5. DOI: `10.1109/CDC.2001.914704`.

[6] Huamin Chen, Jian 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 (IEEE Cat. No.04EX975)*. Oct. 2004, pp. 124–133. DOI: `10.1109/MAHSS.2004.1392089`.

[7] Giacomo Chiarot and Claudio Silvestri. "Time Series Compression Survey". In: *ACM Comput. Surv.* 55.10 (Feb. 2023). ISSN: 0360-0300. DOI: `10.1145/3560814`. URL: `https://doi.org/10.1145/3560814`.

[8] Marco Dalai and Riccardo Leonardi. “Approximations of One-Dimensional Digital Signals Under the$l^i nfty$Norm”. In: *IEEE Transactions on Signal Processing* 54.8 (Aug. 2006), pp. 3111–3124. ISSN: 1941-0476. DOI: 10.1109/TSP.2006.875394.

[9] Dell. *Hard Drive - Why Do Solid State Devices (SSD) Wear Out*. Accessed on 10/06/2023. Sept. 2021. URL: https://www.dell.com/support/kbdoc/da-dk/000137999/hard-drive-why-do-solid-state-devices-ssd-wear-out?lang=en.

[10] Frank Eichinger et al. “A Time-Series Compression Technique and Its Application to the Smart Grid”. In: *The VLDB Journal* 24.2 (Apr. 2015), pp. 193–218. ISSN: 1066-8888. DOI: 10.1007/s00778-014-0368-8. URL: https://doi.org/10.1007/s00778-014-0368-8.

[11] Hazem Elmeleegy et al. “Online Piece-Wise Linear Approximation of Numerical Streams with Precision Guarantees”. In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 145–156. ISSN: 2150-8097. DOI: 10.14778/1687627.1687645. URL: https://doi.org/10.14778/1687627.1687645.

[12] Frederik Agneborn, Emil L. Bech, Teis V. Harrington, Martin O. Lykkegaard, Daniel Vilslev. “MOBY: MOdel-Based compression sYstem”. Aalborg University, 9th Semester Project/Pre-Master Thesis, group cs-22-dt-9-01. 2023.

[13] Wang Fuzong, Guo Helin, and Zhao Jian. “Dynamic data compression algorithm selection for big data processing on local file system”. In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. 2018, pp. 110–114.

[14] Robert van de Geijn and Margaret Myers. *ALAFF Catastrophic cancellation*. Accessed on 15/06/2023. Apr. 2023. URL: https://www.cs.utexas.edu/users/flame/laff/alaff/a2appendix-catastrophic-cancellation.html.

[15] Carlos Gonzalez and David Lara Arango. “Techniques for the Automated Detection of Anomalies and Assessment of Quality in High-Frequency Data Collection Systems”. In: *4th Hull Performance & Insight Conference*. 2019, pp. 143–152.

[16] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[17] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. “Modelardb: Modular model-based time series management with spark and cassandra”. In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1688–1701.

[18] Lasse V. Karlsen. *Efficient way of storing Huffman tree (Lasse V. Karlsen’s reply)*. Accessed on 09/05/2023. Apr. 2009. URL: https://stackoverflow.com/questions/759707/efficient-way-of-storing-huffman-tree.

[19] Iosif Lazaridis and Sharad Mehrotra. “Capturing sensor-generated time series with quality guarantees”. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. Mar. 2003, pp. 429–440. DOI: 10.1109/ICDE.2003.1260811.

[20] Hussein Sh. Mogahed and Alexey G. Yakunin. “Development of a Lossless Data Compression Algorithm for Multichannel Environmental Monitoring Systems”. In: *2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE)*. Oct. 2018, pp. 483–486. DOI: 10.1109/APEIE.2018.8546121.

[21] NASA. Accessed on 07/06/2023. Mar. 2023. URL: https://atmos.nmsu.edu/data_and_services/atmospheres_data/MARS/curiosity/rems.html.

[22] Robert C. Nelson. *Debian 11.x (Bullseye) Minimal Snapshot*. Accessed on 15/06/2023. May 2023. URL: https://forum.beagleboard.org/t/debian-11-x-bullseye-monthly-snapshots/31280.

[23] Tuomas Pelkonen et al. “Gorilla: a fast, scalable, in-memory time series database”. In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1816–1827. ISSN: 2150-8097. DOI: 10.14778/2824032.2824078.

[24] Piotr Przymus and Krzysztof Kaczmarski. “Dynamic Compression Strategy for Time Series Database Using GPU”. In: *New Trends in Databases and Information Systems*. Ed. by Barbara Catania et al. Cham: Springer International Publishing, 2014, pp. 235–244. ISBN: 978-3-319-01863-8.

[25] Sheldon M. Ross. “Chapter 2 - Descriptive statistics”. In: *Introduction to Probability and Statistics for Engineers and Scientists (Sixth Edition)*. Ed. by Sheldon M. Ross. Sixth Edition. Academic Press, 2021, pp. 11–61. ISBN: 978-0-12-824346-6. DOI: https://doi.org/10.1016/B978-0-12-824346-6.00011-9. URL: https:

//www.sciencedirect.com/science/article/pii/B9780128243466000119.

[26] Raimund Seidel. “Small-Dimensional Linear Programming and Convex Hulls Made Easy.” In: *Discrete & computational geometry* 6.5 (1991), pp. 423–434. URL: http://eudml.org/doc/131168.

[27] Julien Spiegel, Patrice Wira, and Gilles Hermann. “A Comparative Experimental Study of Lossless Compression Algorithms for Enhancing Energy Efficiency in Smart Meters”. In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. July 2018, pp. 447–452. DOI: 10.1109/INDIN.2018.8471921.

[28] Robert Underwood et al. “FRaZ: A Generic High-Fidelity Fixed-Ratio Lossy Compression Framework for Scientific Floating-point Data”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 567–577. DOI: 10.1109/IPDPS47924.2020.00065. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS47924.2020.00065.

[29] B.P. Welford. “Note on a Method for Calculating Corrected Sums of Squares and Products”. In: *Technometrics* 4.3 (1962), pp. 419–420. DOI: 10.1080/00401706.1962.10490022. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022. URL: https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022.

[30] Ian H. Witten, Radford M. Neal, and John G. Cleary. “Arithmetic Coding for Data Compression”. In: *Commun. ACM* 30.6 (June 1987), pp. 520–540. ISSN: 0001-0782. DOI: 10.1145/214762.214771. URL: https://doi.org/10.1145/214762.214771.

XIV. Appendices

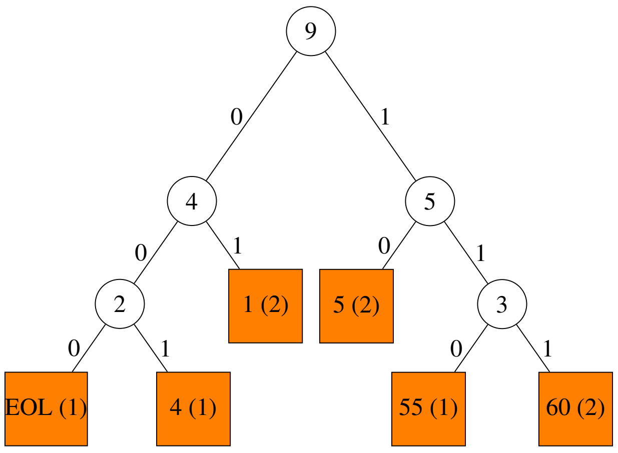


Figure 15: Huffman coding tree based on the global offset list show in Table 2

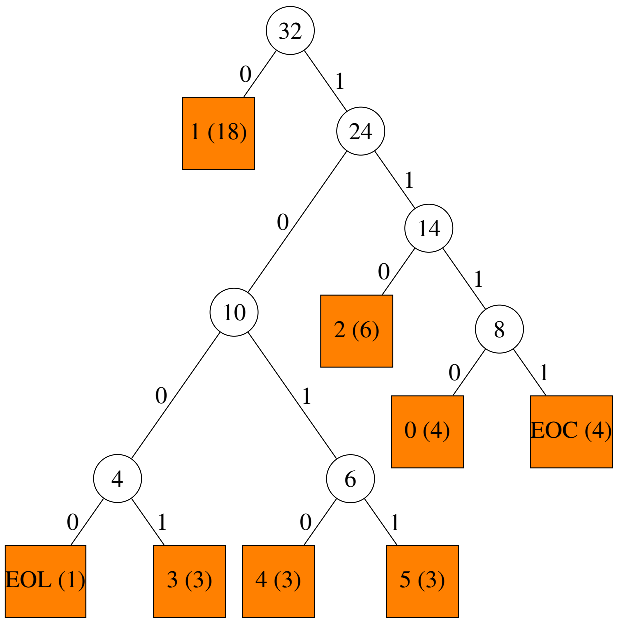


Figure 16: Huffman coding tree based on the local offset list show in Table 3

Table 8: Huffman codes for global offset list

Value	Control code
1	01
4	001
5	10
55	110
60	111
EOL	1000

Table 9: Huffman codes for local offset list

Value	Control code
0	1110
1	0
2	110
3	1001
4	1010
5	1011
EOTS	1111
EOL	1000

Algorithm 5 DecodeTree

```
Input: Bit string Encoding
Output: Tree node
 1: if (next bit of Encoding is 1) then
 2:     Value ← read value from Encoding
 3:     Tree node Node
 4:     Node.Value ← Value
 5:     return Node
 6: else
 7:     Tree node leftChild ← DecodeTree(Encoding)
 8:     Tree node rightChild ← DecodeTree(Encoding)
 9:     Tree node newNode
10:     newNode.leftChild ← leftChild
11:     newNode.rightChild ← rightChild
12:     return newNode
13: end if
```