Living on the Edge: Efficient Handling of Large Scale Sensor Data

Roman Karlstetter
Technical University of Munich
IfTA GmbH
Garching/Puchheim, Germany
roman.karlstetter@tum.de

Amir Raoofy
Technical University of Munich
Garching, Germany
amir.raoofy@tum.de

Martin Radev
Technical University of Munich
Garching, Germany
martin.radev@tum.de

Carsten Trinitis
Technical University of Munich
Garching, Germany
carsten.trinitis@tum.de

Jakob Hermann
IfTA GmbH
Puchheim, Germany
jakob.hermann@ifta.com

Martin Schulz
Technical University of Munich
Garching, Germany
martin.w.j.schulz@tum.de

***Abstract*—Real-time sensor monitoring is critical in many industrial applications and is, e.g., used to model and predict operating conditions to optimize operations as well as to prevent damage in machinery and systems. In many cases, this data is generated by a myriad of sensors and stored or transmitted for post-processing by data analysts. Handling this data near its origin—on the edge—imposes significant challenges for storage and compression: it is necessary to store it in a format that is suitable for large data analytics algorithms, which in most cases means columnar storage. Furthermore, to provide efficient storage and transmission of such sensor data, it must be compressed efficiently. However, existing solutions do not address these challenges sufficiently. In this work, we present a holistic approach for fast streaming of large scale sensor data directly into columnar storage and integrate it with a proven compression scheme. Our approach uses a pipelined scheme for streaming and transposing the data layout, combined with a byte-level transformation of data representation and compression, which we evaluate in comprehensive experiments. As a result, our approach enables transformation of large scale sensor data streams into an efficient, analytics-friendly format already at the sensor site, i.e., on the edge, at data ingestion time. By implementing our optimized approach in the open and widely used columnar storage format Apache Parquet, which we already partly upstreamed, we ensure its accessibility to the community.**

***Index Terms*—sensor data streaming, edge computing,**

I. Introduction

Industrial installations use an ever-growing number of sensors to monitor machine health and operational state. The petroleum industry [1], water supply and distribution networks [2], power generation, e.g., wind turbines [3] and gas-fired power plants [4], HPC centers [5], and many more: they all benefit from the data generated by an enormous number of sensors in their monitoring systems, which they analyze to optimize operations, detect potential problems, and prevent failures. In many cases, these sensors measure physical phenomena or other derived values and deliver continuous streams of sensor data. These streams often include data sampled at a very high data rate, which is especially necessary for monitoring and studying the underlying physical processes that generate high-frequency oscillations. Further, for each installation, monitoring systems measure and analyze the signals from many sensors simultaneously, leading to numerous amounts of parallel and fast data streams (see Figure 1 ②).

These parallel streams are processed in real-time to adjust machine parameters, prevent machine failures or to detect potential problems (see Figure 1 ③). Additionally, many scenarios require storing this data for later use, either on site or—if network bandwidth allows it—in the cloud. For example, the collected data is used to design new and more expressive metrics to indicate the health of underlying machinery and to improve the monitoring systems' real-time failure detection mechanisms. Moreover, in case of damage to the machinery, the collected data is used for root cause analysis, i.e., to understand failures and their reasons. Traditionally, such analyses do not happen on site, but are rather moved to an off-site data center, where data from many installations can be combined (e.g., see Figure 1 ⑥).

Handling the large scale sensor data streams for such analysis scenarios imposes *two* main challenges:

First, data streams are typically ingested and stored in a *row-oriented* (time-ordered) layout (Figure 1 ④), which is the straightforward approach for high-throughput data stream storage, as the streams are directly written to permanent storage without the need for any further *layout processing* after ingestion. However, for most analysis algorithms, data stored in a *column-oriented* (sensor-ordered) storage format leads to more efficient analyses [6]. This underlines the need for an automated layout processing step, i.e., conversion from row-oriented to column-oriented layout. Ideally, this is done before persisting the data to permanent storage, close to the data origin, i.e., on the edge device. However, the processing

This work was supported by *Bayerische Forschungsstiftung* under the research grant *Optimierung von Gasturbinen mit Hilfe von Big Data* (AZ-1214-16), a collaboration project of *TU München* and *IfTA Ingenieurbüro für Thermoakustik GmbH*.

978-1-7281-9586-5/21/$31.00 ©2021 IEEE
DOI 10.1109/CCGrid51090.2021.00010

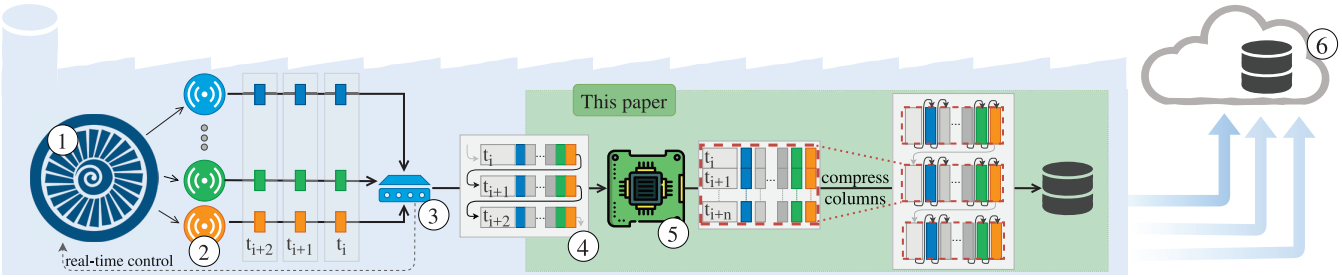

Fig. 1: Deployment scenario for monitoring industrial assets, here at the example of a gas turbine for power generation ①. A limited time frame of data from several sensors ② is combined and analyzed on an embedded digital signal processor without any storage capabilities ③. This happens in real-time to protect the machinery. In order to store the data for further offline analysis, the row-oriented (time-ordered) data stream ④ is sent to an edge-system ⑤, still on site. This edge-system stores the sensor data and provides real-time analysis results. A subset of the data stored on site is sent to a central storage system ⑥, where data from other, similar installations is gathered as well. Our contributions address efficient handling of this sensor data stream by efficiently transforming it to a column-oriented (sensor-ordered) format as well as optimizing compression.

resources at this stage of data processing, i.e., at the edge, are scarce, rendering this a challenging task.

Second, yet perhaps even more important, these sensor data streams generate *huge volumes* of data. Optimizing compression directly impacts the needed storage space and network bandwidth for data transmission. It is thus essential to consider data compression from the beginning and integrate it directly into the ingestion system at the edge, near the machinery.

While there are systems handling these individual aspects, there is no holistic approach that optimizes the entire data flow from sensor to columnar storage; and there is also no study to quantify the handling of *fast* sensor data in a practical real-world scenario (see Figure 1).

In this paper, we present a novel end-to-end solution to address these challenges. Our approach extends the pipeline with two optimized, complementary components that enable both efficient layout processing and sensor data compression on the edge device. We demonstrate the feasibility of our approach by implementing a fully working system that extends the Apache Parquet [7] storage format. We upstreamed our implementation as part of Apache Parquet so that the research community can easily reproduce and apply our approach to their data sets and use it in their projects.

Additionally, we comprehensively evaluate the handling of fast and large scale sensor streams with our system. For these evaluations, we rely on a real-world use case, exploit realistic settings, and take the unique advantage of using data acquired from a real-world industrial monitoring system to quantify the performance and practical resource requirements on different hardware platforms. Finally, using a case study, we show that our system is also applicable to low-power edge systems, which is an important usage scenario in industrial settings.

In summary, we make the following key contributions:

- We design, implement and optimize an *end-to-end* system, from sensor to compressed columnar storage.
- We provide a comprehensive evaluation of the developed system on multiple platforms.
- Using a real-world data set, we show that our approach improves both compression ratio and sustained throughput for floating-point sensor data compared to prior art.
- We integrate our approach into an industrial system and demonstrate its functionality and efficiency in a real-world setting.

The remainder of this paper is organized as follows: We first give an overview of the context and define the problem statement in Section II. In Section III, we describe our holistic approach of streaming sensor data to columnar storage and how to efficiently compress these sensor data streams, and in Section IV we detail our implementation. We provide an extensive evaluation of our system together with a discussion in Section V. In Section VI we demonstrate the applicability of our approach in a real-world use case, and we discuss related work Section VII. Finally, Section VIII concludes our paper.

II. Problem Statement

Streams of sensor data are ubiquitous, representing the backbone of virtually every modern intelligent monitoring system, as they are used in industrial (e.g., Industry 4.0 applications), home/consumer (e.g., smart homes and vehicles) and research (e.g., large scale experimental setups) environments alike. As one concrete example, which is representative of many practical scenarios in scale and resource requirements, we describe a combustion monitoring and protection system for heavy-duty gas turbines used for power generation (see Figure 1). The combustion process in current generations of gas turbines is monitored using up to 32 high-frequency pressure oscillation sensors, each of which generates a stream of single-precision floating-point values at a rate of 25.6 kHz (see Figure 2). Further, monitoring systems generate not only raw sensor data: they also integrate operational information, derive extra information such as frequency spectra of raw data as well as other results of application-specific real-time analyses, which are represented as additional data streams and can easily double the amount of produced data. In total, a single turbine alone

generates more than 500 GBytes of uncompressed single-precision floating-point data per day.

State-of-the-art systems handle this data by storing it in a row-oriented format (in most cases, binary), as this is the most natural and efficient way of streaming it to permanent storage. As data analysis and visualization often requires the processing of individual sensor streams, a columnar data storage format speeds up analysis workloads considerably [4], [6]. For example, pre-filtering or simple analysis of individual signals, could already be performed on the edge, avoiding unnecessary data transmission to cloud systems. In other words, a column-oriented layout makes data analytics more straightforward, since data is already in a suitable format for analysis. Further, column-oriented layout enables compression algorithms to work more efficiently, as alike data is implicitly grouped together. Consequently, we need to be able to convert the incoming data stream from the row-oriented layout—as the sensor processing system generates it—to a column-oriented format usable for analysis workloads directly already on the edge device near the origin of data.

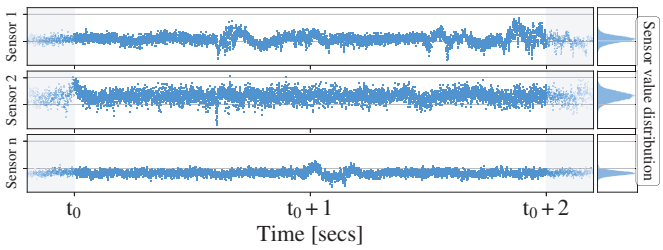

Fig. 2: Example of raw sensor data streams for three different sensors, sampled synchronously. This two-seconds excerpt shows more than 50 000 sensor readings in each plot. The right part of the plot shows a histogram of the value distribution.

However, transferring and storing this data—even after such transformations—puts a lot of pressure on network and storage systems; it is thus necessary to compress it with a suitable compression method, and do so as soon as possible in the data pipeline. We show an example of sensor values for the described use case in Figure 2. It can be clearly seen that there is a lot of noise in the sensor signal, a combination of measurement noise and physically existing process noise of the monitored asset. On the other hand, there are limitations on the range and resolution of possible sensor values, creating opportunities for optimizing compression, specializing them specifically for such sensor systems. Additionally, during normal operation, there is only a low dynamic range in these values (as indicated by the histogram on the right of Figure 2). Consequently, in this paper we show that, by taking these data properties into consideration in the design of our compression stack, it is possible to substantially improve both compression ratio and throughput, while taking the limited processing capabilities on the edge into account.

Overall, we therefore propose an end-to-end system, which 1) transforms the data layout from the row-oriented input stream into a column-oriented analytics format, 2) efficiently handles compression, specifically targeted at sensor data and 3) show that all this can be done on a low-power edge device.

III. Approach

We start by describing the core ideas of our approach in designing an end-to-end system: the two aspects of streaming data to columnar storage and compressing floating-point data more efficiently are discussed separately in the following two subsections. Although the design criteria for these two aspects in our end-to-end approach are conceptually orthogonal, the resulting effects are intertwined, and both need to be considered together to reach an optimal sensor handling system suitable for a large range of sensor streams, including the one discussed in Section II. We start with explaining the core idea of converting row-oriented data into column-oriented format.

A. Streaming to Columns

As shown in Figure 1, in a typical monitoring system for industrial assets, sensor readings arrive in parallel for one particular instant t_i. The data for these sensor readings is sent to an edge computer, which handles permanent storage. State-of-the-art systems store the arriving data streams to permanent storage *as is*, i.e., they continuously append the row-oriented sensor values at instant t_i to a file. This is typically followed by encoding, compression and manual conversion steps.

As this layout is not helpful for most analytics operations, we transpose the data stream and structure it into a columnar layout. In the remainder of this paper, we will call this step *layout transformation*. Since we do this layout transformation on the fly and at the edge to avoid high transformation costs needed when applied during post-processing, we need to take the limited compute and memory resources on such edge devices into account.

We employ a double-buffered and pipelined streaming scheme for the layout transformation and apply it before the encoding and compression step. As the length of the data stream can be assumed to be infinite (systems are operating 24/7) and the memory resources of any computing device are limited, the streaming scheme relies on assembling the arriving streams into buffers with predefined capacities. As a consequence, only a limited number of consecutive rows from the row-oriented data stream are buffered in memory. We call this set of consecutive rows buffered together a *row group* (see dashed red rectangles in Figure 1).

We assume there is enough physical RAM to buffer two of these row groups completely in memory. While buffering on different persistent storage technologies (either explicitly via memory mapped files or implicitly via system swap) is potentially possible as well, it creates a variety of additional challenges that are out of scope of this paper. We assume further that the number of sensors does not change dynamically—resulting in a rigid data schema—and that all sensors are sampled synchronously by a signal acquisition system[1] (cf. Figure 1 ③). Such a system typically processes a window

[1]These are valid assumptions for many monitoring facilities, especially those introduced in Section II.

of fixed (small) length and sends the raw sensor data (see Figure 2) together with several analysis results.

Based on these assumptions, the actual approach for buffering a row group is the following: since we know the set of columns (i.e., number and types of sensors data streams) and the *row group size* (`rgs`: the number of rows in a row group, can be configured based on the memory limitations on the device), we can calculate the buffersize for one row group as

$$\text{buffersize} = \texttt{rgs} \times \sum_{\texttt{c} \in \text{Columns}} \texttt{sizeof}(\texttt{type}(\texttt{c})).$$

For the pipelined layout transformation, we allocate two of these buffers and use them in an alternating fashion for data buffering. These buffers have a transposed layout in which different columns are structured at different offsets. A row that arrives at instant t_i is cut up and scattered to the right spots in the buffer.

Once a buffer is completely filled (or, to ensure timely streaming, when a certain time interval has passed), it is terminated, *prepared for serialization* and written to a file or to the network. We execute this preparation and writing phase using a background thread to decouple buffering from data writing. This enables us to avoid disruption in data ingestion, as we are dealing with a continuous stream of values arriving from the sensors, and since preparing and writing data can be time consuming (see Section V). Having data in column-oriented format, we then also enable benefits like improved compressibility due to the implicitly achieved grouping, as subsequent values in the data storage come from the same sensor. It also enables further optimizations, which are discussed in Section V-F. After a predefined number of row groups have been streamed to a file, this file is terminated and new data goes into a new file.

To keep up with the incoming data stream, all parts of the pipeline need to support the throughput of the arriving data stream, including the two software components layout transformation and preparing data for serialization.

B. Two-Step Floating-Point Compression

We base the compression component of our system on a proven compression approach, consisting of 1) a fast reversible reorganization step and 2) the usage of a proven general-purpose compressor. This base scheme is thereby similar to the Blosc library [8] or the shuffle filter in HDF5 [9].

The rationale behind our approach is to prepare the stream of floating-point numbers so that the general-purpose compressor is much more efficient on parts of the reorganized data stream. For this, we take a window of fixed length of the buffered stream of one sensor and reorganize the floating-point binary representations of this single sensor stream, using a specialized scheme we call *byte stream split*, into multiple, more "similar" streams (see Figure 3). These intermediate streams are then concatenated and compressed using a general-purpose compressor (e.g., zstd).

For reconstruction, we just decompress the compressed stream and combine the bytes from the resulting streams to assemble values and reconstruct the original floating-point stream. This reconstruction requires information about how many values from the stream are split up in one pass; we call this *blocksize* in later parts of this paper.

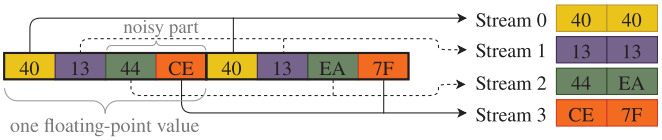

Fig. 3: Example floating-point *byte stream split*. Here we are transforming a simple stream of two floating-point values 2.3010745 = `0x401344CE` and 2.3111875 = `0x4013EA7F` into four separate streams.

C. Combining Streaming and Compression

Although the streaming and compression components' design and optimization are conceptually orthogonal, combining them in a complete end-to-end system creates synergies, as converting data streams first to a columnar layout is beneficial for compression. Specifically, for compression, we are explicitly relying on the "similarity" of streams in column-oriented layout after the on-the-fly layout transformation step, as such similarity is less likely for streams in row-oriented layout. Consequently, our approach enables efficient compression of data streams by exploiting a layout transformation step, early on, at the data ingestion point. Therefore, the combination of streaming and compression steps makes deployment on an edge system practical.

IV. Implementation in Apache Parquet

To demonstrate the feasibility of our end-to-end approach, we implement a fully working system based on *Apache Parquet*, which is a widely used columnar storage format in the data analytics community. We start with a short introduction to Apache Parquet and argue why it is a suitable skeleton for our system. Next, we discuss implementation details of streaming sensor data to Apache Parquet and then present our efficient two-step compression implementation. Finally, we sketch how both ideas are combined into a fully working system.

A. Apache Parquet

Apache Parquet [7] is a column-oriented storage format integrated into popular computing and data analytics frameworks such as Apache Spark, Apache Arrow and Pandas, and is suitable for efficient representation of tabular data. The format splits all rows into smaller chunks called *row groups*; in fact, we borrowed this term from the Apache Parquet format specification in Section III-A. A serialized Apache Parquet file consists of so-called *pages*, which contain the actual data values (at least one page for each column in each row group) and meta data information. A page also serves as the smallest unit of compression.

Our implementation is based on the C++ implementation of Apache Parquet, which is part of *Apache Arrow* [10].

Since its version 0.11.0, Apache Arrow offers two possible API-alternatives to write to a row group. In the first approach, the developer calls `AppendRowGroup()` on a `ParquetFileWriter` instance, providing data for one row group in a column-by-column order. The other approach, `AppendBufferedRowGroup()`, internally buffers the complete row group until it is terminated, so that data can be appended in an out-of-order fashion to all the columns.

The Apache Parquet specification describes two possibilities of compressing data pages. First, *encodings* provide a sort of lightweight compression, which may already considerably reduce data size. Second, encoded pages may be compressed using one of several supported general-purpose compressors, like zstd or Brotli [11], [12]. These two complementary steps in the Apache Parquet specification match our two-step compression approach: we insert our data reorganization scheme as an additional encoding and combine it with a general-purpose compressor available in Parquet.

B. Streaming to Apache Parquet

When using `AppendBufferedRowGroup()` for buffering, the Arrow implementation relies on the use of one separate memory allocation per column and growing all these allocations as data is ingested. This approach imposes no memory consumption restriction early on, and the memory consumption grows with ingesting more data, resulting in a very flexible buffering scheme. However, it potentially requires repeated memory reallocation calls, to serve the memory demand on growing row groups. Consequently, operating system overheads in managing all these buffers grow when streaming to a large number of columns. Furthermore, for streaming one row to Apache Parquet, only a single value is added to every column, which amplifies any overhead the library calls entail. This version of the API is thus not well suited for the layout transformation step.

To avoid these performance problems, we allocate two large chunks of memory, each with enough space for all rows and columns in the row group, and use a logical column-wise layout on top of each. Adding new values to this buffer as they arrive from the stream one by one now only incurs the cost of copying the data to the buffer with offsets that are computed based on the logical column-wise layout.

Once all data for one row group is buffered, we swap buffers and continue with layout transformation on the other buffer.

As described before, the buffer that has been fully filled is now prepared for serialization in a background thread. To do so, we call the Apache Arrow API `AppendRowGroup()` to create a new row group and put the data into this newly generated row group of the Apache Parquet file, with a single call to the library for each column. This invokes several data serialization steps. First, the data is grouped into data-pages, which subsequently are encoded and compressed one-by-one. Meta information describing the column is added last in another data page.

It is important to note that the buffers for layout transformation are allocated only once, since allocating and deallocating them repeatedly induces an unnecessary overhead.

C. Two-Step Compression

For the implementation of our two-step compression scheme, we use the encoding and compression functionalities of Apache Parquet implemented in Apache Arrow. With that, we rely on the general-purpose compressors integrated into Apache Arrow which simplifies our implementation.

We implement *byte stream split* as a new *encoding* in Apache Parquet. In contrast to other encodings in Apache Parquet, this encoding does not reduce the data size on its own, but prepares the input for compression. The implementation consists of an *encoder* and *decoder* for writing and reading[2] respectively. Again, we use the existing infrastructure in Apache Parquet that stores the size of blocks (in our case the *page size*), which is necessary for decompression.

To ensure that the additional *byte stream split* step does not slow down the compression stack, we evaluated three alternative implementations: a simple implementation using two nested loops and two manually vectorized implementations using SSE and AVX2 instruction sets. The simple implementation loops over the input, splits a single element and scatters the bytes to the corresponding streams. While this simple implementation serves as a baseline which can be used on any system, we determined that the compiler does not efficiently auto-vectorize all code paths. The SSE and AVX2 implementations use a combination of `shuffle`, `unpack` and `permute` instrinsics with different lane and stride sizes.

As an example, we describe the single-precision version utilizing `SSE`-intrinsics (see Figure 4), using an optimal sequence of dependent stages. This transformation sequence requires four stages for the encode-transformations and two stages for the decode-transformations. It processes 16 consecutive single-precision values at the same time, split up in four 128-bit SSE registers. After loading data into the registers, the encoder applies a series of interleavings using `unpack` intrinsics to distribute the bytes. The encoding transformation finishes after four stages where each SSE register contains the bytes of 16 values for each corresponding output stream. Then, the encoder stores the registers to the intermediate buffers. The decoder works analogously, but only requires two stages.

Our performance comparison on the systems in Table I shows a few important characteristics. First, all (automatically or manually) vectorized code is bound by memory bandwidth. This implies that smaller block sizes benefit from the CPUs cache hierarchy and thus can yield higher throughput, similar to how `memcpy` behaves. Next, the achievable throughput for the handcrafted vectorized version is at least one order of magnitude higher than what compressors like zstd can achieve.

[2] The implementation of the decoder is not a part of streaming, but we are including it as it is required for the use in a complete system that later also reads the data, e.g., reading data in a data analytic application.

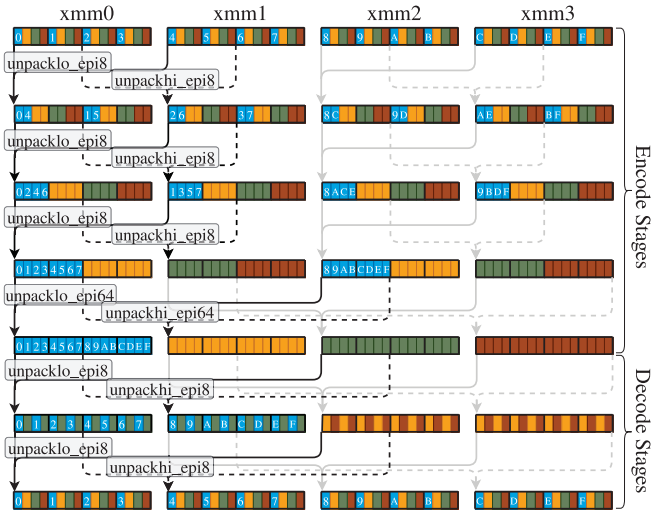

Fig. 4: Single-precision floating-point *byte stream split* encoder-decoder transformation sequences using `unpack` intrinsics. This processes 16 single-precision floating-point values simultaneously. Note that the encoding and decoding are independent operations. For simplicity and ease of explanation, they are visualized back-to-back here. The grayed out arrows represent the same operations in the respective stages, and the annotations are left out for clarity.

V. Experimental Evaluation

In this section, we provide an empirical evaluation of our implementation. We start by proposing a number of evaluation questions that help to characterize the various aspects of our approach's performance. Then we describe our experimental setup and go over the evaluation aspects one by one. We finish this section with a short discussion of our results and findings.

A. Evaluation Questions

As in the previous sections, we first analyze the streaming and compression aspects of our approach individually. Then, we look at the combination of the two aspects. Our experiments are organized around the following *question sets* (QSx):

- QS1: What is the impact of the number of sensors and row group size on layout transformation's performance?
- QS2: How much can *byte stream split* improve compression ratio of general-purpose compressors? How much does it speed up compression and decompression? Which compression algorithms and settings work best?
- QS3: Since our approach is primarily targeting streaming systems at the edge, we raise and answer these questions: What is the sustained throughput achieved by our complete system for various realistic scenarios? What impact do the storage medium options have on the overall performance?

B. Experimental Setup

Since our approach is meant to apply to both high-end server systems and low-power edge computers, i.e., close to the machine or system producing the sensor data, we evaluate performance on two types of systems: One is equipped with a powerful server CPU, the other is a passively cooled low-power edge system, as it is typically found in industrial installations. The details of those two systems are in Table I.

TABLE I: Configurations of the systems used for evaluations

	Server System	Edge System
CPU	Intel® Xeon® Gold 6136 CPU @ 3.00GHz	Intel® Atom™ x5 Processor E3940 @ 1.60GHz
RAM	4 × 16 GiB DDR4 @ 2666 MT/s	2 × 4 GiB DDR3 @ 1866 MT/s
TDP	150 W	9.5 W
ISA	SSE & AVX	SSE
Network	10 GBit Ethernet	1 GBit Ethernet

Both systems run Ubuntu 20.04 and use the *performance* CPU frequency scaling governor. For implementing our streaming approach, we use C++, compile with g++ compiler version 9.3.0 and use the options '`-O3 -march=native`'. As our implementation of *byte stream split* is upstream in the Apache Arrow and Parquet-MR libraries, we use the Conda packages `arrow-cpp` and `pyarrow` version 0.17.1 and link against this version of the library for the streaming experiments. Unless specified otherwise, we use default parameter values in the libraries for our experiments.

C. QS1: Layout Transformation Performance

In the first experiment, we assess the impact of the number of sensors (columns) and row group size on the performance of layout transformation. For our test, we create a small driver program that runs the experiments. This driver program uses pre-recorded floating-point sensor values and feeds them into our layout transformation procedure. This procedure carries out the layout transformation and fills a pre-allocated floating-point buffer with the columnar layout: For each row, it iterates over all sensor values in the row-oriented input stream and appends one value to each column. To make the results easily reproducible, and to avoid any influence of the underlying storage system, we disable serialization to persistent storage for these experiments.

With Apache Parquet for our implementation, it is essential to also consider the performance impacts of Apache Parquet serialization, which happens right after the layout transformation. For this reason, we conduct two experiments for every combination of the number of columns and row group size that we test: 1) we only conduct the layout transformation, and 2) we additionally serialize to Apache Parquet.

Our evaluation in Figure 5 shows that the configurations with a larger number of columns have lower throughput, and those with a medium number of columns yield the best throughput. This behavior directly stems from the performance of the layout processing step and its efficiency in using in the cache hierarchy of the system, as transforming the layout of the arriving sensor values almost exclusively consists of `memcpy` operations.

Another observation in the plots is that except for very small row group sizes, serializing data into Apache Parquet (using

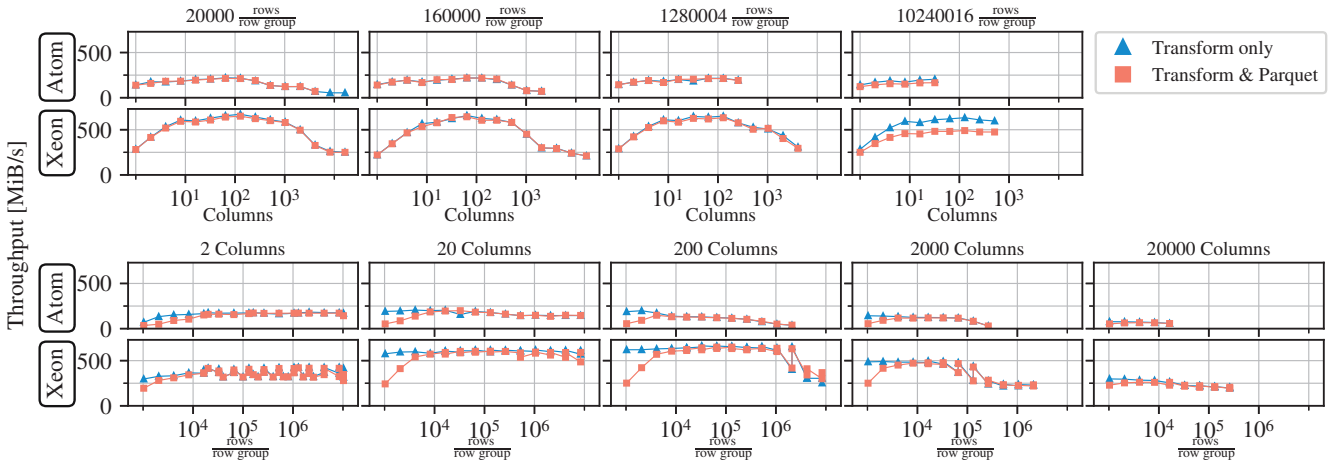

Fig. 5: Layout transformation performance for varying rows per row group and number of columns.

the background thread) does not bring additional overhead. This is also expected since we do not have compression or encoding enabled for this experiment yet, so preparing the Apache Parquet format is a matter of several `memcpy` operations. For small row group sizes, the overhead of these many small `memcpy` operations becomes noticeable, though.

For certain row group sizes, a cache eviction effect considerably decreases the performance for layout transformation. Such performance degradations happen when the number of columns is greater than the number of cache sets, and the addresses of "neighboring" elements in each row fall into the same cache set in the transposed layout. In our experiments, we mainly use row group sizes that avoid this problem, so this effect can only be seen for number of columns above 200 and large row group sizes in Figure 5. Additionally, this problem can be avoided by padding the allocation of row groups, so that we do not hit the same cache lines.

D. QS2: Compression Performance

In this section, we investigate the overall performance of our two-step compression approach, which includes the study of compression metrics. We use our raw sensor data stream (see Figure 2) as benchmark data set.

The different encoding/compression configurations are tested with the following approach: Using python as test driver, we load 1 GB of data (250M single precision values) into an Apache Arrow `Table` object, and call `write_table()` from the `pyarrow.parquet` package with the respective encoding and compression settings. This results in all data being put into a single row group. Data is then written to a file on a local SSD-drive. After that, we clear the Linux page cache and read the file that has been just written to measure read performance. The compression ratio is computed as the number of bytes in the Apache Arrow table divided by the size of the resulting Apache Parquet file. For throughput measurements, we average our results across 10 runs.

The results of this compression performance evaluation are visualized in Figure 6. They show that our approach

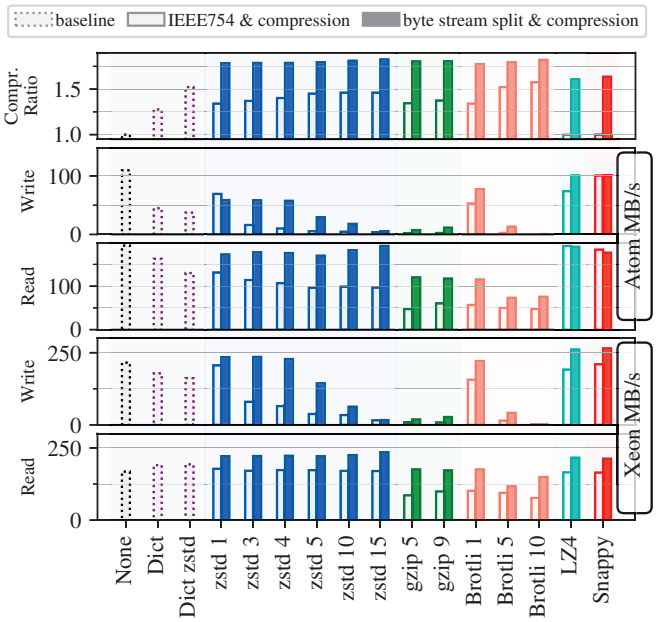

Fig. 6: Comparison of compression ratio, write and read throughput for Atom and Xeon platforms across a set of encoding/compression combinations. The first measurement (None) is there for reference, providing a baseline of the I/O-system's speed. We are using *dictionary encoding*—a fast and lightweight alternative encoding in Apache Parquet—as an additional baseline.

improves the best state-of-the-art variants in all three metrics, compression ratio, write and read throughput, and does so on both hardware platforms under test. Further, it is apparent that the performance of our approach does not depend on any particular general-purpose compressor, as it improves the compression ratio for all tested compression algorithms, regardless of compression level, and improves write and read throughput in almost all cases. On the other hand, our approach

enables significant improvements for compression algorithms like Snappy and LZ4, which almost do not compress the data streams at all without preprocessing. These now provide significant gains in compression ratio, more than any other unmodified state-of-the-art algorithms we tested. Yet, they still provide highest compression and decompression speeds among all examined alternatives. For those compression algorithms that have a compression level setting (zstd, gzip, and Brotli in Figure 6), using higher compression levels does not yield significant compression ratio improvements when compared to additional computational effort that has to be invested. Adding the *byte stream split* reorganization step already improves the compression ratio more than any compression level increase could yield. This suggests that our method is especially well-suited for simple, high-throughput compression algorithms.

The plot also shows that both tested systems benefit from our two-step approach, for compression and decompression.

E. QS3: Sustained Throughput Performance

In the last experiment, we evaluate the overall performance for streaming sensor data to Apache Parquet combining all ideas presented in this paper. In this experiment, we take a buffer in memory of row-oriented floating-point values originating from multiple sensors as the input, using the data from Section II. We take values from this row-oriented buffer and feed them to our layout transformation step, as described before. In the next step, we perform *byte stream split* in combination with zstd on the columns with default compression level of 1. This is followed by persistent storage of the result into an Apache Parquet file. Since we are interested in sustained throughput, we set the Linux kernel parameter `vm.dirty_bytes` [13] to a small value of 100 MB. This ensures that our driver process blocks when it is waiting for the data to be flushed to storage. To compensate for any OS-buffering effects that might happen regardless, we write files with a size of 20 GiB for these experiments. We run the experiments for a realistic configuration of 200 sensors and a row group size of 500 000. To illustrate the impact of the underlying storage medium on throughput, we write into various mediums: SSD, HDD, and a comparably powerful 10 GBit NAS, as three typical mediums used for back-end storage. Like before, we also include results when not persisting at all as a "best case". As baseline, we examine what throughput the Apache Arrow `BufferedRowGroupWriter` solution can deliver. Additionally, we perform one large experiment for 20 000 sensors and a row group size of 20 000.

We obtain various insights from the results in Figure 7, and start by examining compression ratios. We achieve a compression ratio of 1.34 with zstd, while compressing *byte stream split* encoded streams with zstd results in the much better compression ratio of 1.78. These ratios correspond to the compression of 20 GiB raw sensor streams into an Apache Parquet file of size 14.95 GiB and 11.24 GiB, respectively. In addition to this considerable reduction in disk storage utilization, the additional improvement in compression ratio has another performance implication: As a result of this higher

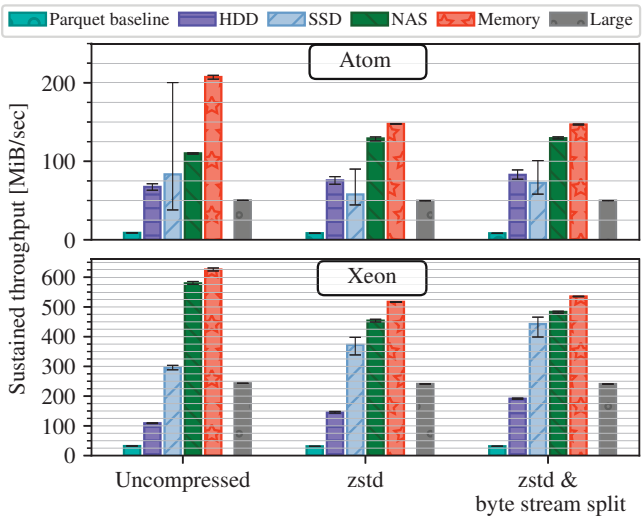

Fig. 7: Sustained end-to-end performance for a configuration of 200 columns and a row group size of 500 000. *Large* shows the results for a configuration of 20 000 columns and a row group size of 20 000. Note that the y-axes have different scales for the two test systems, annotated by Atom and Xeon. Error-bars indicate min and max throughput of experiment runs.

compression ratio, less amount of data passes I/O, which can be a potential bottleneck in many systems.

We next look at the results on the edge system. Since the 10 GBit NAS is only connected via the 1 GBit network interface of the edge system, we hit the limit of this Ethernet interface in the uncompressed setting. We also observe that compression even reduces the overall throughput here, since the processor cannot compress the columns fast enough (see Section V-F for a potential solution to this problem). Even though the large configuration delivers a throughput that is considerably lower, it would still be fast enough for typical deployment scenarios.

For the Xeon system, we note that writing uncompressed to local persistent storage is bound by write throughput of the HDD and SSD, respectively. We highlight here that compression improves the overall performance, and the higher compression ratio that can be achieved by the *byte stream split* approach has a positive influence on sustained performance for local storage. In case of the NAS, adding compression slightly reduces throughput, since the storage system is faster than zstd throughput in this case. We discuss a potential improvement to avoid this compression bottleneck in Section V-F.

In all cases, our approach clearly outperforms the Apache Arrow baseline, which is bound by some implementation-specific overhead.

F. Discussion

a) Streaming performance and memory consumption: While Apache Arrow supports buffering data since version 0.11, our approach outperforms this implementation by one order of magnitude (see Figure 7). However, we trade that performance by having less flexibility in the final size of the

row-group. That being said, the fixed row group sizes is not a huge limiting factor by itself for our use case: row group sizes can be pre-configured based on the system's memory resources, which effectively compensate for the inflexibility.

b) Row group size: The maximum possible row group size is tuned based on the amount of memory the system can use for writing. In addition to performance aspects when writing, the row group size also impacts read performance. Larger row group sizes are generally preferred for data analysis, since decompression performance is higher on larger buffers, while small row group sizes increase the share of meta data information.

Another aspect that is affected by row group size is latency of writing data to and reading it from permanent storage. Larger row group sizes mean that data takes longer until it can be serialized to the storage system. If latency is a concern, however, the system needs to be augmented with an orthogonal component capable of reading the row group buffers.

c) Compression performance: Our buffering approach enables performing the compute-intensive tasks on all the columns in parallel, on the available processing units in a system. More specifically, as our experiments show, compression is the dominating component in preparing serialization to disk on the edge system (see Section V-E). This opens up exploiting parallel processing for an originally sequential data stream. While we postpone this idea to future work, it gives additional potential for optimization.

VI. End-to-End Case Study

In this section, we show how our system can be used in a realistic end-to-end setting. We describe a typical setup for the example scenario we introduced in Section II and prove the feasibility of this scenario for the edge-system from Table I.

For the setup we evaluate in this section, we simulate multiple row-oriented data streams (④ in Figure 1) on a dedicated machine. Each such data stream generates its own time stamps and gets converted into a separate columnar layout. We generate single-precision floating-point data samples at a rate of 25 600 Hz from a real recording (see Figure 2), which is important for reproducing compression behavior. One stream simulates 32 such sensors, representing an edge-case for state-of-the-art monitoring systems (typical setups in this domain often still have fewer sensors). Together, each simulated turbine thus produces a data stream with a raw data rate (not considering time stamps and other meta data) of:

$$25\,600 \frac{\text{Samples}}{\text{sec and Sensor}} \times 4 \frac{\text{Bytes}}{\text{Sample}} \times 32 \text{ Sensors} \approx 3.28 \frac{\text{MByte}}{\text{sec}}$$

A data generator sends these streams via TCP over a GBit Ethernet connection to the edge device (Edge-System in Table I). There, for each stream, we apply our complete stream handling pipeline, consisting of layout transformation, encoding and compression and store the resulting parquet files to an HDD[3]. We let this pipeline run long enough to ensure that for each stream, eight row-groups are filled completely.

We keep increasing the *number of simulated turbines* connected to the one system handling this data, up to the point where it fails to handle all data streams. At this point, the generator is forced to pause data generation, resulting in a gap in the data stream (backpressure via TCP). Choosing a row group size of 800 000 rows, which results in a maximum required buffer size of roughly 6 GBytes[4] for layout transformation, is a viable configuration for the Edge-System from Table I.

For analysis, we evaluate the *maximum gap*[5] between consecutive samples for any of the simultaneous streams, induced by a bottleneck in stream handling.

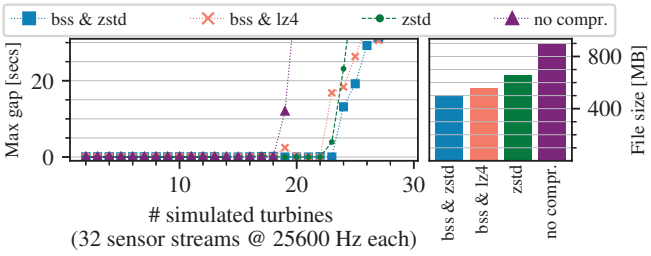

Fig. 8: The maximum gap remains zero up to a certain number of simulated turbines where some part of the pipeline cannot handle the amount of data anymore.

Our results in Figure 8 show that our system can seamlessly handle up to 23 simulated turbines, equivalent to the overall throughput of ≈ 75 MByte/sec. Further analysis shows the limit here is disk throughput, which underlines that both *byte stream split* and compression are essential steps in the pipeline. Otherwise, the system's capacity in handling data streams will be hit already for fewer simulated turbines, as indicated by the curve for the uncompressed stream in Figure 8. We further measured the resulting file size, which shows that the proposed compression approach shrinks the required storage space considerably compared to existing compression alternatives.

VII. Related Work

a) Time series storage systems: There are several existing solutions that handle storage of time series streams. InfluxDB provides a time series database which uses a custom storage backend [14]. To tackle their problems with some workloads, they work on a new database core called *IOx*, which also uses Apache Parquet for persisting data [15]. TimescaleDB, an extension on PostgresQL, utilizes so called Hypertables to manage one consistent view over multiple chunks of data [16]. While this increases performance of typical workloads for time series data, it still employs row-oriented storage. Building on top of HBase, OpenTSDB provides a solution targeting distributed setups [17]. All of these solutions

[3]HDDs still have a more competitive price point and their performance is sufficient in current setups. To be as close to the real setup as possible, we hence decided to evaluate the setup for HDDs.

[4]For 29 parallel data streams:
29×2 Buffers $\times$ 800 000 $\frac{\text{Rows}}{\text{Buffer}} \times 32$ Sensors $\times$ 4 Bytes $\approx$ 6GByte.

[5]A forced pause in data generation since data handling cannot keep up with the data rate.

require some kind of data serialization and parsing to and from text to ingest data, which is prohibitive for edge systems where processing resources are scarce.

b) Distributed and event stream storage systems: A lot of research effort has been spent on distributed streaming storage and processing systems. While some of the use cases and problems discussed in such works [2], [18] are similar or related to our use case, distributed stream processing is not what we aim for with this paper. Furthermore, many publications regarding distributed stream processing consider event streams and the challenges that come with it [19]–[24]. This is fundamentally different from our usage scenario in both data type and data rate or regularity. An event is usually much more complex (it may arrive in the form of text or structured text) than a single sensor reading (one 4-byte floating-point value) and arrives with a much more irregular frequency.

c) Compression: While there is a huge amount of work on data compression in industry and academia, we want to highlight shuffle filters in HDF5 [9] and the blosc library [8]. These two approaches are use a similar approach for improving general purpose compression.

VIII. Conclusions

We presented a holistic approach for handling large and fast sensor data in modern monitoring systems through efficient streaming into a columnar data layout combined with an effective data representation and compression scheme. Our compression implementation is based on Apache Parquet and has been upstreamed, making it possible for the broader community to reuse. Our in-depth investigations show that our approach is practicable on low-power edge devices, providing data in an analytics-ready format directly at the data generation site. By converting the data into a columnar format on the edge, followed by efficient data reorganization and compression, our approach improves both efficiency and effectiveness when compared to existing solutions, and it can be easily generalized to other and even lossy compression schemes. This opens up a set of new possibilities for analytic use-cases in large scale industrial settings, providing new opportunities such as effective predictive maintenance or dynamic operational optimizations based on the rich data streams available already today from such systems, but which often are left unused due to the missing processing options. Our approach allows future optimizations like parallelizing compression workloads, further enhancing sensor monitoring and acquisition systems. Thus, it brings significant improvements to systems for streaming of sensor data and has a direct influence on real-time monitoring systems across the industry.

References

[1] A. Baaziz and L. Quoniam, "How to use big data technologies to optimize operations in upstream petroleum industry," *International Journal of Innovation*, vol. 1, 12 2014.

[2] X. Ren, O. Curé, L. Ke, J. Lhez, B. Belabbess, T. Randriamalala, Y. Zheng, and G. Kepeklian, "Strider: An adaptive, inference-enabled distributed rdf stream processing engine," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1905–1908, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137805

[3] A. Aguilera, R. Grunzke, D. Habich, J. Luong, D. Schollbach, U. Markwardt, and J. Garcke, "Advancing a gateway infrastructure for wind turbine data analysis," *J. Grid Comput.*, vol. 14, no. 4, p. 499–514, Dec. 2016. [Online]. Available: https://doi.org/10.1007/s10723-016-9376-9

[4] R. Karlstetter, R. Widhopf-Fenk, J. Hermann, D. Rouwenhorst, A. Raoofy, C. Trinitis, and M. Schulz, "Turning Dynamic Sensor Measurements From Gas Turbines Into Insights: A Big Data Approach," ser. Turbo Expo: Power for Land, Sea, and Air, vol. Volume 6: Ceramics; Controls, Diagnostics, and Instrumentation; Education; Manufacturing Materials and Metallurgy, 06 2019, v006T05A021.

[5] A. Netti, M. Müller, A. Auweter, C. Guillen, M. Ott, D. Tafani, and M. Schulz, "From facility to application sensor data: Modular, continuous and holistic monitoring with DCDB," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 64:1–64:27.

[6] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05. VLDB Endowment, 2005, p. 553–564.

[7] D. Vohra, *Apache Parquet*. Berkeley, CA: Apress, 2016, pp. 325–335. [Online]. Available: https://doi.org/10.1007/978-1-4842-2199-0_8

[8] The Blosc Developers. (2020) What Is Blosc? Accessed December 2020. [Online]. Available: https://blosc.org/pages/blosc-in-depth/

[9] The HDF Group. (2020) HDF5 User's Guide. Accessed December 2020. [Online]. Available: https://support.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide.pdf

[10] The Apache Software Foundation. (2020) Apache Arrow. Accessed April 2020. [Online]. Available: https://arrow.apache.org/

[11] Facebook. (2020) Zstandard. Accessed April 2020. [Online]. Available: https://facebook.github.io/zstd/

[12] J. Alakuijala and Z. Szabadka, "Brotli Compressed Data Format," RFC 7932, Jul. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7932.txt

[13] van Riel, Rik and Peter W. Morreale. (2008) Linux Kernel Documentation for /proc/sys/vm/*. Accessed September 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/sysctl/vm.txt

[14] InfluxData Inc. (2020) InfluxDB. Accessed April 2020. [Online]. Available: https://docs.timescale.com/latest/main

[15] Paul Dix, InfluxData Inc. (2020) Announcing InfluxDB IOx - The Future Core of InfluxDB Built with Rust and Arrow. Accessed December 2020. [Online]. Available: https://www.influxdata.com/blog/announcing-influxdb-iox/

[16] Timescale Inc. (2020) TimescaleDB. Accessed April 2020. [Online]. Available: https://docs.timescale.com/latest/main

[17] The OpenTSDB Authors. (2020) OpenTSDB. Accessed April 2020. [Online]. Available: http://opentsdb.net/

[18] M. Zhang, T. Wo, T. Xie, X. Lin, and Y. Liu, "Carstream: an industrial system of big data processing for internet-of-vehicles," *Proc. VLDB Endow.*, vol. 10, pp. 1766–1777, 08 2017.

[19] M. Borkowski, C. Hochreiner, and S. Schulte, "Minimizing cost by reducing scaling operations in distributed stream processing," *Proc. VLDB Endow.*, vol. 12, no. 7, p. 724–737, Mar. 2019. [Online]. Available: https://doi.org/10.14778/3317315.3317316

[20] M. Hoffmann, A. Lattuada, and F. McSherry, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *Proc. VLDB Endow.*, vol. 12, no. 9, p. 1002–1015, May 2019. [Online]. Available: https://doi.org/10.14778/3329772.3329777

[21] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "A holistic view of stream partitioning costs," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1286–1297, Aug. 2017.

[22] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1825–1836, Aug. 2017.

[23] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1634–1645, Aug. 2017.

[24] Q. Huang and P. P. C. Lee, "Toward high-performance distributed stream processing via approximate fault tolerance," *Proc. VLDB Endow.*, vol. 10, no. 3, p. 73–84, Nov. 2016. [Online]. Available: https://doi.org/10.14778/3021924.3021925