CompressStreamDB: Fine-Grained Adaptive Stream Processing without Decompression

Yu Zhang[◇], Feng Zhang[◇], Hourun Li[◇], Shuhao Zhang[+], Xiaoyong Du[◇]

[◇]Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China
[+]Information Systems Technology and Design Pillar, Singapore University of Technology and Design
{yu-zhang21,fengzhang, lihourun}@ruc.edu.cn, shuhao_zhang@sutd.edu.sg, duyong@ruc.edu.cn

***Abstract*—Stream processing prevails and SQL query on streams has become one of the most popular application scenarios. For example, in 2021, the global number of active IoT endpoints reaches 12.3 billion. Unfortunately, the increasing scale of data and strict user requests place much pressure on existing stream processing systems, requiring high processing throughput with low latency. To further improve the performance of current stream processing systems, we propose a compression-based stream processing engine, called CompressStreamDB, which enables adaptive fine-grained stream processing directly on compressed streams, without decompression. Particularly, CompressStreamDB involves eight compression methods targeting various data types in streams, and it also provides a cost model for dynamically selecting the appropriate compression methods. By exploring data redundancy among streams, CompressStreamDB not only saves space in data transmission between client and server, but also achieves high throughput with low latency in SQL query on stream processing. Our experimental results show that compared to the state-of-the-art stream processing system on uncompressed streams, CompressStreamDB achieves 3.24× throughput improvement and 66.0% lower latency on average. Besides, CompressStreamDB saves 66.8% space.**

I. Introduction

Stream processing technology is becoming increasingly prevalent in the current big data era [1, 2, 3, 4]. In 2021, the global number of connected IoT devices grows 9% and there are 12.3 billion active endpoints [5]. Stream processing can perform analysis and queries on a large number of continuously arriving tuples of data streams, such as sensor data [6] and financial transactions [7], with low latency and real-time requirements as main characteristics. However, as the scale of data streams continues to increase, stream processing systems are facing the pressure of increasing data volumes [8]. On the one hand, a large amount of stream data bring great pressure to the transmission bandwidth. If an apparent delay exists in network transmission, the real-time performance of stream processing cannot be guaranteed. On the other hand, stream processing systems often temporarily store data in memory, which also puts huge pressure on the memory usage of the server if the data arrival rate is high. Therefore, it is both necessary and essential to explore new methods to reduce the memory and bandwidth pressures faced by stream processing systems. Data compression [9, 10, 11, 12, 13], as a common method to reduce data size, can be applied to stream processing situations to improve the performance of stream systems and save time and space overhead.

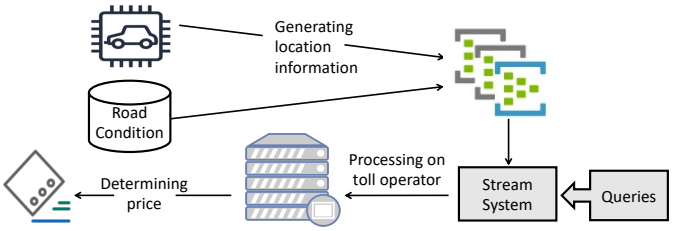

Fig. 1. Use case of linear road system.

The use of compression in stream processing is of great importance, which can help improve the performance of stream systems. It can bring three major benefits. First, in stream processing, a large amount of continuously input data with similar characteristics are being processed, such as timestamps [14, 15], transaction amounts [7], and sensor values [6]. Specifically, 30% data can be duplicated [16]. Due to the similarity of input streams, data redundancy can be effectively reduced through data compression, thereby reducing the volume of the stream data. Second, in stream processing scenarios, a large part of the overhead is caused by memory access and network transmission [17, 18] between nodes, instead of computation. Our experiments show that with 500Mbps network, the transmission time can occupy more than 70%. Therefore, the reduction of data size by compression can obviously improve the performance of stream systems. Third, direct computing on compressed data has been proved to be very useful in data science applications [19, 20, 21, 22, 23, 24], which sheds light on applying compression in stream systems. Overall, compression can bring performance gains.

We show a use case of linear road system [25] in Figure 1, which uses the information from the sensors on vehicles to generate data stream of traffic conditions and can determine road tolls. At peak hours, millions of registered cars utilize the expressways, and each is equipped with a sensor that reports its position and speed every 30 seconds. Accordingly, the volume of data is vast. With real-time changes, the system must compute with minimal latency in order to make toll decisions. Because traditional approaches cannot meet the rigid requirement, compressed stream processing is then urgently needed. Moreover, the stream data contain a

variety of attributes, and are changing dynamically. Because no compression algorithm is suitable for all situations, we need to consider comprehensive compression algorithms with different characteristics.

There is a large literature on stream processing systems, but few studies have focused on the comprehensive compression methods in stream processing, especially for SQL-based compressed stream processing. Supporting SQL in stream processing helps users process data in a user-friendly manner, and direct SQL processing on compressed data helps to further improve the system performance [26]. Among the existing stream engines, TerseCades [27] ensures the benefits of compressed stream processing through optimizations in terms of hardware architectures, but it uses only an simple integer compression method and a floating-point compression method, without comprehensive performance model for guidance. Scabbard [3] targets fault-tolerance in stream processing engine. It aims at single-node stream processing and complications like networking are not its focus. TRACE [28] is a stream framework that allows compression on traffic monitoring data, but it considers only the special speed-based representation of data. Different from these works, we study the fine-grained adaptive compressed stream direct processing with comprehensive cost model supporting diverse compression methods.

However, designing compressed stream direct processing systems faces three major challenges. First, stream processing systems requires low latency. Unfortunately, compression algorithms often take time to encode, causing severe delays. For example, in our experiments in Section II-B, when using Gzip as a compression tool, compression can occupy 90.5% of the total stream processing time, which is an intolerable overhead. Second, in stream processing scenarios, the processing queries and input data can change at any time according to different user requirements. Moreover, part of compression algorithms can achieve higher compression ratios, while other compression algorithms have less time overhead for compression and decompression. For different input workloads, the compression needs to be adaptive to achieve better performance. We need to weigh the pros and cons of choosing the compression algorithm. Third, the compressed data can require a decompression process before the SQL query execution. In our experiments, the time overhead of decompression compared to query execution ranges from 2.09× to 31.37×. Decompression can bring additional time and space overhead, which can bring down the performance.

We develop a compression-based stream processing engine, called **CompressStreamDB**, which can address the above three challenges through the special designs of the system. First, to ensure low latency and real-time requirements in stream processing, the compression algorithms need to be lightweight and fast. In CompressStreamDB, eight lightweight compression algorithms are adopted in the stream system, which can achieve significant performance improvements on diverse input streams. Second, for input streams with different characteristics, we design a fine-grained adaptive compression algorithm selector, which can dynamically choose the compression algorithm that can bring the greatest performance improvement. In detail, as the workload changes, the selector makes decisions based on the characteristics of the input data (including the value range, the degree of repetition, etc.) and given parameters, with the help of our system cost model. The model can estimate the time consumed by each compression algorithm, thus selecting the compression algorithm that can achieve the optimal performance. Third, to handle the decompression overhead, we propose a solution that can directly query the compressed data if the data are aligned in memory. In detail, if the data are still structured after compression, query operation can be applied to the compressed data with minimal modification. Moreover, we regard efficient lightweight decompression-required methods as a special case, which can also be integrated into CompressStreamDB.

We conduct experiments in a cloud environment, and use three real-world datasets with different properties. The platform is equipped with an Intel Xeon Platinum 8269CY 2.5 GHz CPU. Experiments show that compared to the state-of-the-art stream processing method, CompressStreamDB can achieve the highest system efficiency, with 3.24× throughput improvement and 66.0% lower latency on average. Besides, CompressStreamDB can generate 66.8% space savings.

Overall, we make the following three major contributions.

- We develop a compressed stream processing engine, which includes diverse lightweight compression methods, suitable for different scenarios.
- We propose a system cost model to guide the compressed stream processing, and design an adaptive compression algorithm selector based on the cost model.
- We design a processing method of directly performing SQL queries on compressed streams, which can avoid the overhead caused by decompression.

II. Background

A. Stream Processing and Streaming SQL

Stream processing. Stream processing is a terminology in data science that focuses on the real-time processing of continuous streams of data, events, and messages. In detail, stream processing generically refers to the study of a number of disparate systems, including dataflow systems, reactive systems, and certain classes of real-time systems [29]. The query is the SQL statement used for data processing, which can be further subdivided into different operators. In the context of stream processing, a stream consists of a sequence of tuples, and a tuple is a record that represents an event including elements such as timestamp, amounts, and values. The tuples constitute batches. A batch represents the processing block that contains a certain number of tuples. In a batch, we use *column* to represent the elements of different tuples in the same field. Unlike database processing referring to relatively more complex and time-consuming analysis of enormous data that have already been stored over a period of time, stream processing emphasizes micro-batch, dynamic, and continuous processing. Stream processing is an ideal method to promptly obtain approximate or speculative results from unbounded streams

of data. Therefore, stream processing has been widely applied in particular scenarios that requires minimal latency, real-time response with tiny overhead (e.g., *risk management* [30] and *credit fraud detection* [14]), and predictable and approximate results (e.g., *SQL queries on data streams* [15] and *click stream analytics* [31]).

Streaming SQL. Among various fields of stream processing, Streaming SQL is one of the emerging hot research topics. Streaming SQL can be perceived as the streaming version of SQL processing on streams of data, instead of the database. Traditional SQL queries process the complete set of available data in the database and generate definite results. In contrast, streaming SQL needs to continuously process the arriving data, and the result is non-determined and constantly changing. As a result, this can raise a number of issues, such as how to reduce the response time. Streaming SQL owns declarative nature similar to SQL, and provides an effective stream processing technology, which largely saves the time and elevates the productivity in stream data analysis. Besides, many stream systems have been proposed, such as *Apache Storm* [32] and *Apache Flink* [33], whose relational API is suitable for stream analysis, providing a solid development foundation and productive tools.

B. Compression Algorithms

Various compression algorithms have been proposed. To ensure that the query results in stream processing are accurate, only lossless compression algorithms are considered in our system. Lossless compression algorithms can be divided into heavyweight compression and lightweight compression. Popular heavyweight compression algorithms include Lempel-Ziv algorithms [10, 12] and Huffman encoding [34, 9], etc. Although their compression ratio is relatively high, the encoding and decoding processes are too complicated, which brings serious time overhead. Stream processing has the requirements of real-time and low latency. It cannot tolerate long delays during processing. After exploring heavyweight compression algorithms in stream processing systems, we find that they can lead to a higher compression ratio together with longer (de)compression time, which does not significantly improve the performance and stability of the stream system. In our preliminary experiment, we apply a commonly used compression tool Gzip in stream processing systems. The compressed processing system with Gzip spends 90.5% of the total time in compression, and less than 10% time in transmission. Though it can provide high compression ratio and low transmission time, the compression time overhead can cause the system delay or even pause. Hence, we use lightweight compression algorithms to accelerate stream processing.

Lightweight compression. The lightweight compression is a trade-off scheme between compression ratio and performance. The encoding methods of the lightweight compression are relatively simple. Compared to heavyweight compression algorithms, they trade a lower compression ratio for faster compression and decompression time. We have investigated a series of works [11, 13, 35, 36, 37, 38, 39, 40, 41] on lightweight compression algorithms, including most of the commonly used lightweight compression algorithms. Each compression algorithm has its own advantages and disadvantages, and they are suitable for data streams with different characteristics. For example, Elias Gamma encoding and Elias Delta encoding [11] are respectively suitable for small numbers and large numbers. Run Length Encoding [37] is suitable for data with more repetition. The compression effect of Null Suppression [13] depends on the redundant leading zeros in the element. Bitmap and its extensions [39, 40, 41] are suitable for data compression with few values.

Eager and lazy compression. We divide these lightweight compression algorithms into two categories: eager compression and lazy compression [42]. We summarize eight common lightweight compression algorithms of the two categories in Table I. The eager compression algorithms compress as soon as a subset of input tuples arrives. In contrast, the lazy compression algorithms wait for the entire data batch to enter before compressing. The advantage of the eager algorithms is that they can process each tuple in time without waiting, and the advantage of the lazy algorithms is that they can better utilize the similar redundancy of a large amount of data to achieve a higher compression ratio.

TABLE I
EAGER AND LAZY COMPRESSION METHODS IN LIGHTWEIGHT COMPRESSION

	Compression Method	Description
Eager	Elias Gamma Encoding [11]	Encode each value with unary and binary bits.
	Elias Delta Encoding [11]	Encode each value with unary and binary bits.
	Null Suppression with Fixed Length [36]	Delete leading zeros of each value with fixed bits.
	Null Suppression with Variable Length [36]	Delete leading zeros of each value with variable bits.
Lazy	Base-Delta Encoding [35]	Encode values as their delta values from base value.
	Run Length Encoding [37]	Encode values with their run lengths.
	Dictionary [38]	Maintain a dictionary of the distinct values.
	Bitmap [13, 41, 39, 40]	Encode each distinct value as a bit-string.

III. MOTIVATION

In this section, we analyze the necessities in compressed stream processing, and show examples of its widespread use.

A. Problem Definition and Basic Idea

Problem definition. We show the problem definition of processing compressed stream as follows. The input data streams are unbounded sequences of tuples, which are generated from the data source. The data block to be processed in the stream is referred to as window w, containing a sequence of tuples of a preset size. We use SQL queries to handle these streams. Each query contains different operators, including $projection$, $aggregation$, $groupby$, etc. Given a chosen compression algorithm τ, the stream is compressed at source, and the compressed stream is denoted as R'. Finally, compressed streams and queries are transmitted to the processor. The result of compressed stream processing consists of tuples in stream after a series of queries. Our optimization aims at minimizing latency while increasing throughput.

Basic idea of compressed stream direct prcocessing. To solve the problem, our basic idea is compressed stream direct prcocessing. In detail, we develop a fine-grained adaptive

model to select appropriate compression schemes and perform mapping between compressed data and operators. For each streaming SQL operator, we modify the number of bytes it reads and compress the values it uses. In this way, data can be queried without decompression, thus saving both time and space. Note that efficient lightweight decompression-required methods, which can bring significant benefits, should also be considered. In our scenario, we treat it as a special case.

B. Dynamic Characteristics in Stream Processing

Special dynamism in stream processing. The dynamic characteristics of stream data are mainly reflected in the following three aspects. First, the properties of the stream data can change at any time, including the value range, the degree of repetition, etc. As the data attributes change, the compression ratio that different compression algorithms can achieve also varies. Second, affected by factors such as the data generation speed and network delay, the rate at which stream data comes can change at any time, affecting the system's waiting time. Third, it is impossible to predict how frequently data properties can change. Thus, the re-decision frequency for dynamic processing needs to be considered, which is a balance between efficiency and overhead.

Differences from column compression in databases. The characteristics of stream processing are significantly different from those of traditional databases, which require novel compression designs. In traditional database processing, holistic operations can be conducted after scanning all the data, so the compression algorithms can be selected according to the overall data characteristics. Also, compression in databases focuses more on compression ratios rather than low-latency real-time processing. In contrast, in stream processing, an unbounded stream of data arrives in real-time changes. The compression method needs to be updated dynamically to adapt to changes in stream data.

C. Case Study

We show a motivation example in Figure 2 to illustrate the comparison between the static processing in traditional databases and the dynamic processing in stream situation. In Figure 2, we study the case from smart grids [43]. The smart home market is expected to reach a market volume of 51.23 billion by 2026, with an estimated 84.9 million active homes and an annual growth rate of 11.7% [44]. The dataset of smart grids is collected from smart plugs deployed in private households, which contains over 4,055 millions of energy consumption measurements for 2,125 plugs distributed across 40 houses in a period of one month [43]. Seven attributes are contained in the dataset, including timestamps, the measurement value, the ids of the plug and house, etc.

Dynamic characteristics. For the real-time generated electricity consumption data, the characteristics of the stream data are constantly changing. The peaks and troughs of electricity consumption, as well as the different electricity habits of diverse households, can lead to constant changes in the data stream. When a family generates a large amount of electrical power consumption data within a short period of time, the data can appear in the stream as continuously repeated house ids, and changing plug ids and values. When many families consume electricity at the same time, such as the peak hours at night, the house ids can vary frequently, while the timestamps remain unchanged.

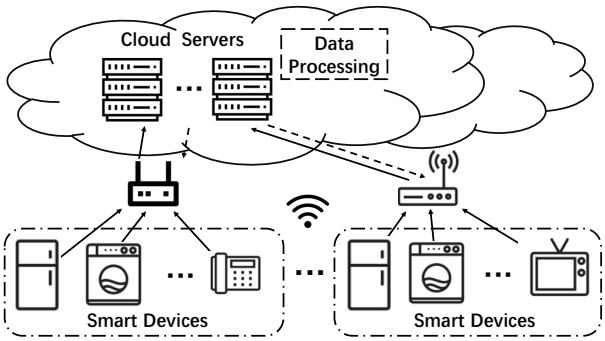

Fig. 2. Example of smart grids.

Opportunity. Traditional processing in databases can analyze the content to be processed in advance, so as to predetermine the processing method. In contrast, in stream scenario, its data come in a streaming fashion, so the stream system needs to process as the events appear. For stream processing methods, it is not available to know all the information to be processed in advance. As the input data stream changes dynamically, its real-time processing method needs to change adaptively. We in Section VII show that our solution, CompressStreamDB, can achieve much better performance over the static processing.

D. Widespread Use of Compressed Stream Direct Processing

A large number of stream applications can benefit from our solution of compressed stream direct processing. We provide some examples to illustrate that our engine can be applied to multiple scenarios.

- **IoT sensor data** from the smart grid domain [43] is an underlying scenario, which involves the analysis of energy consumption measurements. The goal is to provide short-term load predictions and load data for real-time demand management. However, workloads fluctuate dynamically in real time. If we enable compressed stream direct processing, we can increase throughput and process large amounts of data in short time.
- **Real-time decision** in linear road benchmark [25] specifies an expressway variable tolling system. Every automobile on the highway has a sensor that reports its location. These data are used to calculate tolls for the specific section of road that the vehicle is on. Reduced tolls can encourage people to utilize less crowded roadways. With our solution, the system can process large volumes of stream data of vehicle locations more efficiently and provide decision effectively.
- **Cluster management** [45] can monitor the execution of computation tasks. The coming data relate to the status of the cluster, including task submission, state of success or failure, etc. The anomaly detection with unexpected failures should emit as soon as possible. Our solution can provide more rapid response for anomaly detection.

Moreover, other stream applications, such as manufacturing equipment detection [46], ship behavior prediction [47], and temporal event sequence detecting [48], all need real-time stream processing. We demonstrate the time breakdown for these applications in Figure 3. The whole bar represents the total duration of uncompressed stream processing, while the white part represents the time percentage taken by the network transmission. With 500Mbps bandwidth network, the network transmission time occupies over 70% of the total time. Even with 1Gbps bandwidth network, the transmission time still occupies about 50% of the total time. Thus, transmission time is the bottleneck of stream applications, which significantly needs the benefits of compressed stream direct processing.

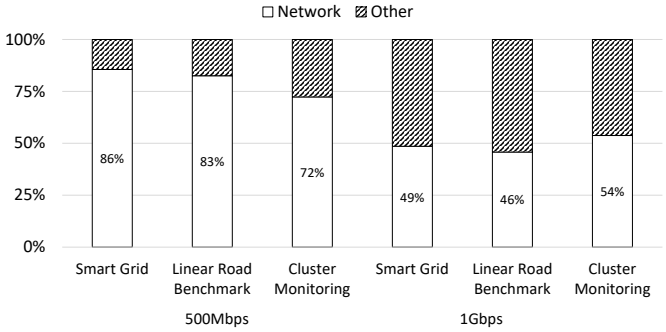

Fig. 3. Total time breakdown.

IV. CompressStreamDB Framework

We propose a fine-grained compressed stream processing framework, called CompressStreamDB, and we show our system design in this section.

A. Overview

CompressStreamDB solves the challenges mentioned in Section I, and can effectively reduce the time and space overhead in stream processing. It can adaptively select compression algorithms and apply them to stream processing.

Structure. The CompressStreamDB framework consists of two parts: client and server, as shown in Figure 4. The client has a compression algorithm selector based on the cost model. The selector is responsible for collecting the data to be processed and selecting the optimal compression algorithm. Note that the term "client" refers to a device that requires compressed stream processing with the server. It can be a data source, such as a sensor or smart phone, or an intermediate node that collects data. Because our compression algorithms are lightweight, compression can be performed on resource-constraint devices like data sources. As a result, our system can be expanded to a multi-layer architecture with multiple layers of compression clients, while the client-server architecture represents a simplified model. Compression are deployed on the client, and the server is responsible for query processing of compressed stream data. The server includes the kernel functions of queries. Note that although CompressStreamDB is designed for the direct processing of compressed stream data, we do not exclude efficient compression algorithms that need decompression. They can also be integrated into the system and should not be ignored.

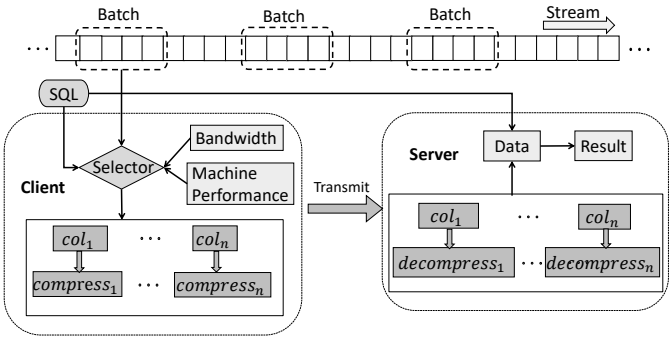

Fig. 4. CompressStreamDB framework.

Scenario. In stream situation, given a series of parameters, such as network throughput and performance metrics of clients and servers, as input, CompressStreamDB can adaptively select the compressions and perform fine-grained compression-based stream processing according to the characteristics of the data stream. The principle of compression algorithm selection is to optimize the performance of the entire system, that is, to minimize the total processing time of the system.

Workflow. After the data are generated in the client of CompressStreamDB, the data mainly undergo a series of processes including compression, transmission, decompression, and query, which is also the basis of our proposed system cost model. Before compression, the selector preloads the data and provides the compression algorithm that can achieve the optimal performance. The decision process is based on our cost model. A wide range of factors are taken into consideration in building our model, from machine metrics to network conditions, as well as the effectiveness and cost of compression algorithms, detailed in Section IV-C. We process the input data at batch granularity, and use different compression algorithms for each column of data, as mentioned in Section II. The compressed data are transmitted to the server, and the server handles the compressed data with corresponding SQL queries.

Batch. In CompressStreamDB, stream data are processed at batch granularity. Batch size setting is a double-edged sword, since the growth of batch size can increase both the latency and the compression ratio. We determine the batch size through dynamic sampling, where its overhead can be amortized during stream processing. Furthermore, it can be specified by users and adjusted according to the actual needs. More experimental details can be found in Section VII.

B. Compressed Stream Processing

Compression. The compression module in CompressStreamDB involves eight different compression algorithms with diverse characteristics. Among the eight compression algorithms, four belong to fixed-length encoding, while the others belong to variable-length encoding. However, variable length coding can bring additional difficulties to transmission and processing. Because for variable length coding, flag bits need to be added to determine the division between elements, and the input stream data need decompression before processing, or at least, segmentation. This greatly affects the efficiency of processing. Therefore, for the compression algorithms in CompressStreamDB, we

use a certain degree of alignment to ensure that the final data is transmitted in whole bytes, which brings convenience for processing.

Adaptive processing for dynamic workload. We use our selector to dynamically process the input data stream in CompressStreamDB. As mentioned in Section II, we use SQL to process stream data, and regard a batch as the minimum processing block. We mainly consider common relational operators, including *projection*, *selection*, *aggregation*, *group-by*, and *join*. Stream processing is performed through query statements composed of these operators with a given size of the sliding window. After a preset number of batches, the compression algorithms for data columns are reselected with the system cost model. CompressStreamDB then scans the next five batches to predict the data properties of the follow-up stream, uses the system cost model to calculate latency with the properties, and finally identifies the new processing method with the lowest total processing time. Considering that the compression algorithms we use are all lightweight, the overhead of dynamic reselection can be negligible.

Query without decompression. Decompression is used to restore the original data. Regarding the cost of decompression, CompressStreamDB avoids decompression as much as possible, thus reducing the decompression time and memory access overhead, and accelerates the query process. In our design, we can directly query the compressed data when the compressed stream meets the following three conditions. First, the compressed data are similar to the data before compression, and are still structured. Second, the compressed stream data should be aligned. Third, the compression does not affect the order of the stream and the process of kernel operation. This universal design avoids the complexity of developing separate operator kernels for different compression methods.

Example. Assume that the stream data includes three columns: $col1$ is 8 bytes, $col2$ is 4 bytes, and $col3$ is 4 bytes. After compression, $col1'$ is 2 bytes, $col2'$ is 1 byte, and $col3'$ is 1 byte. A query like "select col1, avg(col2) from data group by col3" can be mapped to "select col1′, avg(col2′) from data group by col3′". In this way, we only need to update the number of bytes to be read for each corresponding column in the operator. The original stream processing operators are mapped to the corresponding compressed stream processing operators according to the compression.

With such designs, we can compare and calculate the compressed values directly, allowing us to perform queries directly on compressed data. Therefore, the result of compression can be applied to the entire query execution, including intermediate results, which benefits the system efficiency. Note that CompressStreamDB is positioned to integrate diverse compression schemes. For the lightweight decompression-required algorithms, if the performance gain they bring can offset the decompression overhead, they should also be considered.

C. System Cost Model

To guide the system to automatically select a suitable compression algorithm at runtime, we propose a cost model for stream processing systems with compression. Previous works [13, 49] provide the cost models only for the compression algorithms, but not stream processing. As far as we know, our work is the first to provide the cost model for compressed stream processing. The difficulty of proposing a cost model for compressed stream processing lies in the complexity of processing procedures and scenarios. In our processing scenario, we take the machine metrics, network conditions, and other extensive factors into consideration, and solve the above-mentioned difficulties through a multi-step cost model.

The process of CompressStreamDB mainly includes four stages: compression, transmission, decompression, and query processing. We develop a system cost model for the compressed stream processing in these four stages. The main parameters used in our system cost model are listed in Table II.

TABLE II
SYMBOLS AND MEANINGS.

Symbol	Description
α	The compression algorithm is lazy or eager.
β	Whether the compression needs decompression.
r	The compression ratio in transmission step.
r'	The compression ratio in query step.
τ	The compression algorithm.
$Size_T$	The number of bytes per tuple.
$Size_B$	The number of tuples per batch.
N_{client}&N_{server}	The machine performance.
$T^{com,\tau}_{memory}$&$T^{decom,\tau}_{memory}$	The number of instructions for memory accesses.
$T^{com,\tau}_{operation}$&$T^{decom,\tau}_{operation}$	The number of instructions for computation.

We represent the time of compression, data transmission, decompression, and query processing by $t_{compress}$, t_{trans}, t_{decom}, and t_{query}, respectively. The whole process time is represented as t. Based on the above analysis, we can represent the system cost of compressed stream processing in Equation 1.

$$t = t_{compress} + t_{trans} + t_{decom} + t_{query} \tag{1}$$

In the following part of this section, We consider the factors including machine metrics, network conditions, and efficacy of compression methods, in building our cost model. We specifically model the time consumption of the above four aspects.

1) Compression time. For a selected compression algorithm τ, we use $T^{com,\tau}_{memory}$ to represent the number of instructions used for memory accesses during compression, and $T^{com,\tau}_{operation}$ to represent the number of instructions used for computation. Then, $t_{compress}$ can be defined as Equation 2.

$$t_{compress} = \alpha \cdot t_{wait} + \frac{T^{com,\tau}_{memory} + T^{com,\tau}_{operation}}{N_{client}} \tag{2}$$

N_{client} relates to the performance of the client. If the program is a memory-intensive program, N_{client} represents the access speed of the memory of the client. Otherwise, if it is a compute-intensive program, N_{client} represents the CPU frequency of the client. As mentioned in Section II-B, the eager compression algorithms compress data immediately, while the lazy compression algorithms need to wait until the whole data batch arrives. Hence, if we use t_{wait} to represent the time spent waiting for a data batch, we can use $\alpha \cdot t_{wait}$ to

calculate the time that τ spends on waiting, where α is defined in Equation 3.

$$\alpha = \begin{cases} 1, & \text{if the compression algorithm } \tau \text{ is lazy;} \\ 0, & \text{if the compression algorithm } \tau \text{ is eager.} \end{cases} \tag{3}$$

2) Transmission time. We use t_{trans} to represent the time spent on transmission. $Size_T$ represents the number of bytes per tuple, which is decided by the input data, and r represents the compression ratio of the selected algorithm. Then, we use $\frac{Size_T}{r}$ to represent the tuple size after compression. The compression ratio r is detailed in Section V. $Size_B$ represents the number of tuples per batch. Accordingly, for a compressed batch, the size of bytes we need to transmit is $\frac{Size_T \cdot Size_B}{r}$. When the network bandwidth is sufficient and the queuing delay does not need to be considered, t_{trans} can be represented in Equation 4.

$$t_{trans} = \frac{Size_T \cdot Size_B}{r} \cdot latency \tag{4}$$

When the network bandwidth is fully occupied, the calculation of t_{trans} can be given as Equation 5.

$$t_{trans} = \frac{Size_T \cdot Size_B}{r \cdot bandwidth} \tag{5}$$

3) Decompression time. Many future-proof efficient compression algorithms still need decompression before processing, and our system should not exclude them. Moreover, it is not complicated to add the factor of decompression time in the adaptive performance model. For a selected compression algorithm τ, $T^{decom,\tau}_{memory}$ represents the number of instructions used for memory accesses during decompression, and $T^{decom,\tau}_{operation}$ represents the number of instructions used for computation. Then, $t_{decompress}$ can be defined in Equation 6.

$$t_{decompress} = \beta \cdot \frac{T^{decom,\tau}_{memory} + T^{decom,\tau}_{operation}}{N_{server}} \tag{6}$$

N_{server} relates to the performance of the server, which is similar to N_{client}, and β indicates whether τ needs decompression, which is defined in Equation 7.

$$\beta = \begin{cases} 1, & \text{if the compression algorithm } \tau \text{ needs decompression;} \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

4) Query time. The query is executed on the server through kernel functions. The compression process mainly affects the efficiency of kernel functions on memory read and write, but does not affect the computation process. Because CompressStreamDB reads and writes in bytes, the memory read and write time is proportional to the number of bytes occupied in memory. We use $t^{query}_{operation}$ to represent the computation time of a query, and t^{query}_{memory} to represent the query time spent on memory read and write. Note that both of them represent the processing time in the uncompressed condition. We can obtain t_{query} by Equation 8.

$$t_{query} = t^{query}_{operation} + \frac{t^{query}_{memory}}{r'} \tag{8}$$

Note that r' is the compression ratio in the query step, which is different from the compression ratio r, and r' can be determined by whether the server performs decompression or not. Accordingly, it can be defined by Equation 9.

$$r' = \begin{cases} 1, & \text{if the compression algorithm needs decompression;} \\ r, & \text{otherwise.} \end{cases} \tag{9}$$

V. Selected Compression Algorithms

CompressStreamDB involves eight lightweight compression algorithms, as mentioned in Section II. CompressStreamDB can adaptively select these alternatives at runtime. The selection is based on our novel system cost model, and partial parameters of the system cost model relate to the compression algorithms.

A. Preliminaries

We show the corresponding parameters of each compression algorithm in the system cost model, with the advantages and disadvantages of these algorithms. These parameters include α, β, r, and r'. As for the four parameters about the number of instructions, $T^{com,\tau}_{memory}$, $T^{com,\tau}_{operation}$, $T^{decom,\tau}_{memory}$, and $T^{decom,\tau}_{operation}$, they can be directly obtained by reading the assembly source code. In this way, a specific cost model corresponding to each compression algorithm can be built.

In the rest part of the section, the previously mentioned symbol $Size_B$ shall be continually used to represent the processing batch size. In addition, we use $Size_C$ to represent the size of each element in the column. According to the characteristics of different compression algorithms, we design a series of parameters about the properties of the dataset. These parameters shall be detailed in the description of each compression algorithm.

B. Eager Compression

Eager Compression algorithms compress the arrived elements immediately and do not need to wait for the whole batch. Hence, their α value in the system cost model is 0.

Elias Gamma encoding (EG). For a given data column, we use $EGDomain$ to represent the maximum number of bytes required for Elias Gamma encoding forthis column. The compression ratio r of Elias Gamma encoding can be described in Equation 10.

$$r = \frac{Size_C}{EGDomain} \tag{10}$$

Its storage format is aligned so that the encoded results have the same number of bytes, while the compressed data is still structured. CompressStreamDB can avoid decompression using this format. Then, we have its parameters: $\beta = 0$, $r' = r = \frac{Size_C}{EGDomain}$.

Elias Delta encoding (ED). We use $EDDomain$ to represent the maximum number of bytes required for Elias Delta encoding for the elements of this column. Similar to Elias Gamma encoding, the compression ratio r of Elias Delta encoding can be defined in Equation 11.

$$r = \frac{Size_C}{EDDomain} \tag{11}$$

Its parameters are: $\beta = 0$, $r' = r = \frac{Size_C}{EDDomain}$.

Null suppression with fixed length (NS). To estimate the compression effects of null suppression with fixed length and null suppression with variable length, we introduced another dataset property, the $ValueDomain$ array. The size of this array is the same as the batch size, and is used to record the number of bytes required to represent the valid bits of each element in the column. In this method, the maximum value in the array, $ValueDomain_{MAX}$, is the number of bytes used by elements after null suppression with fixed length compression. The compression ratio r of null suppression with fixed length can be represented as Equation 12.

$$r = \frac{Size_C}{ValueDomain_{MAX}} \tag{12}$$

Similarly, its parameters are: $\beta = 0$, $r' = r = \frac{Size_C}{ValueDomain_{MAX}}$.

Null suppression with variable length (NSV). With the array $ValueDomain$ introduced in NS, we can obtain the total number of bytes needed for compression by the sum of $ValueDomain$. The compression ratio r of null suppression with variable length can be defined in Equation 13.

$$r = \frac{Size_C \cdot Size_B}{Size_B/4 + \sum_{i=1}^{Size_B} ValueDomain_i} \tag{13}$$

Because the compressed elements are not byte-aligned, they have to be decompressed before processing. We have its parameters: $\beta = 1$, $r' = 1$.

C. Lazy Compression

Lazy compression algorithms wait until the entire input batch arrives, and then compress the whole batch. Hence, their α value in the system cost model is 1.

Base-Delta encoding (BD). For a given data column, we use $BDDomain$ to represent the maximum number of bytes required for Base-Delta encoding. The compression ratio r of Base-Delta encoding can be defined in Equation 14.

$$r = \frac{Size_C}{BDDomain} \tag{14}$$

It can avoid decompression in CompressStreamDB. Then, we have its parameters: $\beta = 0$, $r' = r = \frac{Size_C}{BDDomain}$.

Run length encoding (RLE). Suppose the average run length of a column of data batch is represented by $AverageRunLength$. Since run length needs to be represented by an extra int variable (4 bytes), the compression ratio r of run length encoding is defined in Equation 15.

$$r = \frac{Size_C \cdot AverageRunLength}{Size_C + 4} \tag{15}$$

RLE is not byte-aligned, and it breaks the original data structure. Hence, it needs decompression before processing. Therefore, we have its parameters: $\beta = 1$, $r' = 1$.

Dictionary (DICT). Assuming that the number of data types is $Kindnum$, then the encoding compression ratio r of dictionary is defined in Equation 16.

$$r = \frac{Size_C}{\lceil \log_2 Kindnum/8 \rceil} \tag{16}$$

It is byte-aligned and structured, so it can avoid decompression. Accordingly, its parameters are: $\beta = 0$, $r' = r = \frac{Size_C}{\lceil \log_2 Kindnum/8 \rceil}$.

Bitmap. Assuming that the number of data types is $Kindnum$, then the encoding compression ratio r of bitmap is defined in Equation 17.

$$r = \frac{Size_C}{2^{\lceil \log_2 Kindnum) \rceil}/8} \tag{17}$$

It destroys the data structure of the original data, so: $\beta = 1$, $r' = 1$.

VI. Implementation

We implement CompressStreamDB with reference to [15, 14, 50]. CompressStreamDB consists of a client module and a server module. The client module includes the compression algorithms applied in stream processing, as well as the adaptive selector. The server module supports common SQL operators including selection, projection, groupby, aggregation, and join. The server is responsible for processing compressed streams. The server also involves a profiler, which can be used to collect performance data, such as (de)compression and transmission time. Note that the compression function of CompressStreamDB can be turned off so that it supports processing uncompressed streams as well. For small-scale stream scenarios, such as processing a single tuple, CompressStreamDB can perform uncompressed stream processing directly without waiting for the entire data batch. Such a hybrid processing mode makes our system applicable to a wide range of stream processing scenarios. For batch implementation, sliding window can span multiple batches. To solve this problem, our system provides a batch buffer and stores the data from the previous batch in it temporarily. When a cross sliding window is detected, the system has to wait for the following batch. Then, previous cached batch of data can be retrieved from the buffer, and the cross sliding window results can be computed.

CompressStreamDB can be deployed among different environments. For example, Flink can customize serializer. We can wrap the compression module of CompressStreamDB into a custom serializer, and then embed it into Flink for use. However, this incurs additional challenges such as model integration and Flink internal implementation overhead, especially in distributed environments. Because our work focuses on the adaptive selection of compressions in stream processing, we leave the adaptation to other systems as future work.

VII. Evaluation

A. Experimental Setup

Methodology. The baseline used in the comparison is CompressStreamDB without compression. As mentioned in Section VI, CompressStreamDB can turn off the compression function for direct stream processing. Our comparison

TABLE III
THE QUERIES USED IN EVALUATION.

Query	Detail
Q1	`select timestamp, avg (value) as globalAvgLoad from SmartGridStr [range 1024 slide 1]`
Q2	`select timestamp, plug, household, house, avg(value) as localAvgLoad from SmartGridStr [range 1024 slide 1] group by plug, household, house`
Q3	`(select timestamp, vehicle, speed, highway, lane, direction, (position/5280) as segment from PosSpeedStr [range unbounded]) as SegSpeedStr -- select distinct L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment from SegSpeedStr [range 30 slide 1] as A, SegSpeedStr [partition by vehicle rows 1] as L where A.vehicle == L.vehicle`
Q4	`select timestamp, avg(speed), highway, lane, direction from PosSpeedStr [range 1024 slide 1] group by highway,lane,direction`
Q5	`select timestamp, category, sum(cpu) as totalCPU from TaskEvents [range 512 slide 1] group by category`
Q6	`select timestamp, eventType, userId, max(disk) as maxDisk from TaskEvents [range 512 slide 1] group by eventType, userId`

to baseline examines whether our solution can help deliver better performance on stream systems. To further understand the advantage of our adaptive compression, we implement the Base-Delta encoding compression with reference to TerseCades [27], which is denoted as "Base-Delta". TerseCades is the first systematic investigation of stream processing with data compression, demonstrating the effectiveness of compression in stream processing. We show our performance progressiveness over the Base-Delta encoding algorithm used in TerseCades. TerseCades is not open source so we reimplement it. We have compared the results of our implementation with the TerseCades paper [27] and the results are similar. For example, in [27], it can process queries on the Pingmesh Data [51] with the throughput of 37.5MElems/s. As in our implementation of Base-Delta encoding, we can process queries on the Smart Grid Data[43] with the throughput of 37.2MElems/s, which is similar. Moreover, we also compare the adaptive compression stream processing of CompressStreamDB with the eight lightweight compression algorithms.

Platform. We perform experiments in the Alibaba cloud [52]. Both the server and the client are equipped with an Intel Xeon Platinum 8269CY 2.5 GHz CPU and 16GB memory, running Ubuntu 20.04.3 LTS with Java 8. Our platform can provide network bandwidth ranging from 0 to 1Gbps between the server and the client.

Datasets. We use three real-world datasets in evaluation. All these datasets have been widely used in previous studies [53, 54, 14, 15] and they are still prevalent topics. For example, the smart home market is estimated to reach 51.23 billion by 2026, with an 11.7% annual growth rate [44]. The first dataset is energy consumption measurement in smart grids [43], which comes from different devices in a smart grid and focuses on load predictions and real-time demand management in energy consumption. The second dataset is compute cluster monitoring [45], which comes from a cluster at Google, and emulates a cluster management scenario. The third dataset is linear road benchmark [25], which denotes the position events of vehicles, and models a network of toll roads.

Benchmarks. We use six queries to evaluate the performance of adaptive compression in CompressStreamDB. We use two queries on each dataset to obtain the performance metrics, and evaluate the performance of different processing methods, including the baseline, eight light-weight compression algorithms, and CompressStreamDB. These queries are commonly used in previous studies [14, 15]. The details of the six queries are shown in Table III. *Q1* and *Q2* are conducted on the dataset of anomaly detection in smart grids. *Q3* and *Q4* are for the linear road benchmark dataset. *Q5* and *Q6* are for the dataset from Google compute cluster monitoring. For the datasets of Smart Grid and Linear Road Benchmark, each batch contains 100 windows and each window contains 1024 tuples. For the Cluster Monitoring dataset, each batch contains 200 windows and each window contains 512 tuples. The performance result for each dataset is the average of the results of related queries.

B. Performance Comparison

Throughput. We explore the throughput of CompressStreamDB for the six queries on the three datasets. The results are shown in Figure 5, each dataset with ten different processing methods. On average, CompressStreamDB achieves 3.24× throughput improvement. Note that the linear road benchmark dataset has negative numbers inside, so Elias Gamma encoding and Elias Delta Encoding cannot be performed on it. We have the following observations. First, on the Smart Grid dataset, CompressStreamDB improves the system performance with a 4.80× increase in throughput over the baseline. The optimal single compression algorithm is DICT encoding, which can achieve 3.00× throughput improvement over the baseline. Compared to DICT, CompressStreamDB achieves 1.60× improvement in throughput. Second, on the linear road benchmark dataset, the throughput of CompressStreamDB is 2.38× higher than that of baseline, while NS is 2.28× than the baseline. The system performance of CompressStreamDB is 4.4% higher than NS. Third, on the Google Cluster Monitoring dataset, CompressStreamDB achieves 2.55× improvement in throughput, while Base-Delta achieves 2.36× improvement. CompressStreamDB reaches 8.1% throughput improvement over Base-Delta.

From the above throughput results, we can obtain the following revelations. First, compression can obviously improve the throughput of the stream processing system, which has been demonstrated in [27]. Although Base-Delta can often pro-

vide good system performance in most situations, the adaptive compression used in CompressStreamDB still achieves better performance. Second, the effect of compression in stream processing is significantly influenced by the dataset properties. For example, if $AverageRunLength$ of the dataset is low, RLE cannot provide enough performance. Hence, the selection of algorithms is an important factor to improve performance. Third, CompressStreamDB can combine the advantages of different algorithms, and adapt to various situations. No matter what the dataset is, CompressStreamDB can achieve similar or better performance than any single compression algorithm.

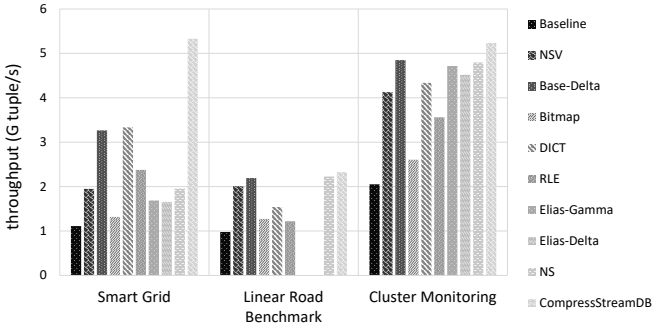

Fig. 5. Throughput of different compression methods.

Latency. Figure 6 reports the latency of different compression algorithms on the three datasets. In our work, latency represents the time from data input to the query result output. Similar to throughput, latency is an important target of the system performance. On average, CompressStreamDB achieves 66.0% lower latency. Moreover, we have the following observations. First, on the Smart Grid dataset, the latency of CompressStreamDB is 79.2% lower than that of the system without compression, and 37.5% lower than DICT. Second, on the linear road benchmark dataset, the latency of CompressStreamDB is 58.0% lower than that of the baseline. The latency of NS is 56.1% lower than that of the baseline. The latency of CompressStreamDB is 4.2% lower than that of NS. Third, on the Cluster Monitoring dataset, CompressStreamDB achieves 60.8% reduction in latency over that of the baseline, while Base-Delta achieves 57.7% reduction. The latency of CompressStreamDB is 7.4% lower than that of NS. In short, CompressStreamDB can provide better latency than any single compression algorithm.

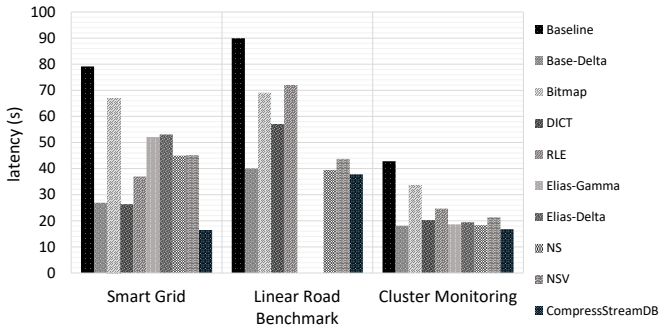

Fig. 6. Latency of different compression methods.

Dynamic workload. To illustrate the comparison between the static processing solution and the dynamic processing in CompressStreamDB, we use the datasets and benchmarks to generate dynamic workloads and evaluate on them. We use Q1 and Q2 on smart grids as an example, and show the speedups with different network bandwidths in Figure 7. For the other cases, the performance behaviors are similar. We denote "Static" for the static compressed processing method with the optimal performance on the dynamic workload, while CompressStreamDB, denoted as "CompressStreamDB", applies our dynamic design. Experimental results show that under different network conditions, CompressStreamDB always achieves much higher performance than the static solution. Under the network with 100Mbps bandwidth, CompressStreamDB exhibits the highest performance improvement, with 9.68× speedup over the baseline, 3.97× over the optimal static method. The performance of the static method cannot achieve the optimal with the dynamic workload, because the method it handles data cannot be changed. As for CompressStreamDB, its adaptive processing method can be dynamically adjusted according to the data characteristics, so its performance keeps stable with the changes.

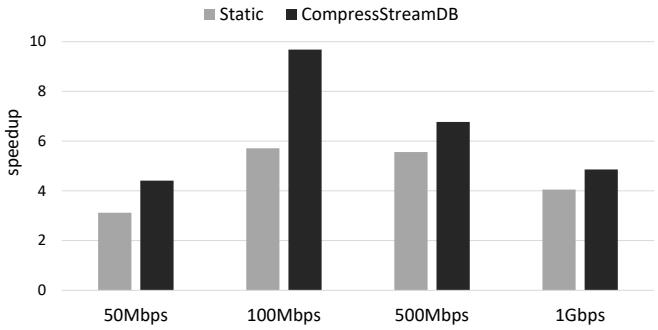

Fig. 7. Speedup with dynamic workload.

C. Analysis of Time and Space Savings

(De)compression time. As we mentioned in Section IV-C, CompressStreamDB targets diverse lightweight fast compression methods. For the compressions that can bring significant performance benefits, even decompression is required, we still involve them in CompressStreamDB. In our experiments, we include three lightweight decompression-required methods, RLE, bitmap, and NSV. The compression time and decompression time during the whole process are shown in Figure 8. Our observations are as follows. First, NS is a simple and fast compression algorithm, with the lowest sum time of compression and decompression in all three datasets. NS can generate a good compression ratio in most situations. Hence, it can achieve high throughput and latency performance. Compared to the other lightweight algorithms, the performance of Elias Gamma encoding and Elias Delta encoding is relatively slow. Though they are also lightweight, their computation and coding process is still more complicated than the others. This can be a major factor affecting their performance. Second, null suppression with variable length (NSV) spends most time on decompression. The process of translating bytes of length takes time, thus influencing its behavior. Even so, note that

TABLE IV
RELATIONS BETWEEN TIME AND COMPRESSION. *Trans_time*: TRANSMISSION TIME OF A SELECTED METHOD DIVIDED BY THE TRANSMISSION TIME OF BASELINE. *Query_time*: QUERY TIME OF A SELECTED METHOD DIVIDED BY THE QUERY TIME OF BASELINE.

Dataset	Ratio	Baseline	BD	Bitmap	DICT	RLE	EG	ED	NS	NSV	CompressStreamDB
Smart Grid	trans_time ratio	1	0.341	0.843	0.327	0.453	0.638	0.647	0.559	0.537	0.198
	1/r	1	0.357	0.866	0.357	0.458	0.643	0.679	0.571	0.571	0.217
	query_time ratio	1	0.476	0.947	0.938	0.872	0.678	0.694	0.607	1.026	0.934
	1/r'	1	0.357	1	0.357	1	0.643	0.679	0.571	1	0.890
Linear Road Benchmark	trans_time ratio	1	0.462	0.803	0.630	0.806	\	\	0.431	0.487	0.420
	1/r	1	0.438	0.797	0.500	0.78	\	\	0.438	0.438	0.405
	query_time ratio	1	0.476	1.044	1.024	0.936	\	\	0.549	1.030	0.657
	1/r'	1	0.438	1	0.500	1	\	\	0.438	1	0.625
Cluster Monitoring	trans_time ratio	1	0.394	0.743	0.460	0.560	0.395	0.427	0.423	0.520	0.380
	1/r	1	0.438	0.781	0.438	0.555	0.438	0.438	0.438	0.438	0.375
	query_time ratio	1	0.824	0.956	0.963	0.920	0.831	0.934	0.868	0.971	0.853
	1/r'	1	0.859	1	0.859	1	0.859	0.859	0.859	1	0.906

query time accounts for the majority of the total time; the decompression time of all lightweight compressions, including NSV, accounts for less than 1.0% of the total time, which can be ignored. Third, CompressStreamDB is not the method that spends the least time on compression and decompression. It takes a moderate amount of time, 26.7% longer than those of fast compression algorithm like DICT. CompressStreamDB does not aim at the optimal performance of the compression aspect, but pays more attention to the overall performance of the system.

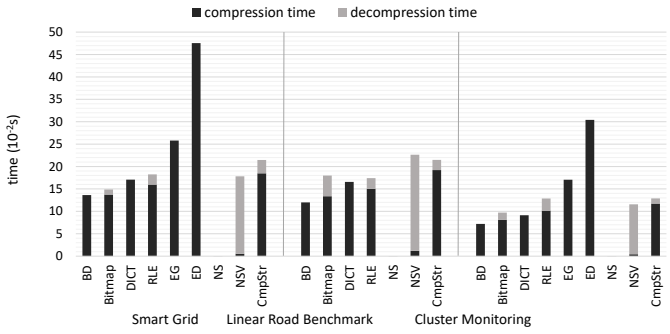

Fig. 8. Time breakdown of compression and decompression. CmpStr is short for CompressStreamDB.

Relation between time and compression ratio. Table IV shows the relation between time and compression ratio r. On average, CompressStreamDB saves 66.8% space and 66.7% *trans_time*. We have the following observations. First, CompressStreamDB achieves the lowest *trans_time* ratio compared to the other processing methods among all datasets. BD, adopted by TerseCades, also achieves a clear advantage of 60.1% on average. However, CompressStreamDB still achieves a 16.6% improvement over that of BD. Second, CompressStreamDB achieves the highest compression ratio r, or the lowest $1/r$, among all processing methods of each dataset. For example, on the Smart Grid dataset, r of CompressStreamDB is 4.61, followed by 2.80 of Base-Delta. CompressStreamDB achieves 1.65× compression ratio r over the optimal single compression algorithm. Third, transmission time ratio and $1/r$ are positively correlated and change proportionally, so as to the ratio of query time and $1/r'$. According to Equation 4 and Equation 5, high compression ratio r indicates low transmission time. Because we use lightweight compression methods, the compression and decompression incur only marginal time. Accordingly, the method with the highest compression ratio can increase the system performance most. CompressStreamDB achieves high performance mainly by its high compression ratio r.

D. Design Tradeoffs and Discussion

Model accuracy. We verify the accuracy of our system cost model in this part. We use the example of the Smart Grid dataset for illustration, as shown in Figure 9. The dashed line and the solid line show the estimated time and measured time respectively. All estimated values are slightly less than the actual values because of additional overhead caused by the system operation. On average, the system cost model of CompressStreamDB can achieve an accuracy of 88.2%, which implies that the cost model is accurate and can be applied to estimate the cost of stream processing with compression.

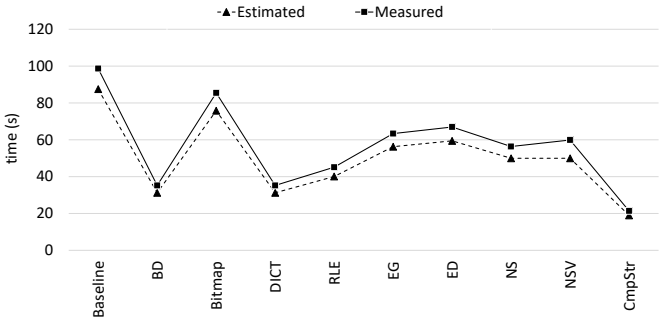

Fig. 9. Accuracy of the cost model. CmpStr is short for CompressStreamDB.

Batch size. As discussed in Section IV-A, batch size influences both latency and compression ratio. We use the Smart Grid workload for illustration, and each window has 1024 tuples. We show their relationship in Figure 10, which includes three network settings: 1Gbps network, 100Mbps network, and single node without network transmission. We have the following observations. First, for 100Mbps, the latency increases with the batch size. In contrast, for 1Gbps network and single node mode, batch size has relatively little effect on system latency. The reason is that when network bandwidth is limited, the data have to be queued before transmission, and thus large batch can result in system pauses. Second, the space occupancy decreases as the batch size increases. The reason is that larger batch size can better utilize data redundancy. Third, batch size has no optimal value, and needs to be analyzed in

specific situations. Moreover, we also measure the cross-batch sliding window, and we alter the window slide size in the range {1, 128, 256, 512, 1024} at different network settings. We find that the performance with different slides are nearly the same (with less than 2% fluctuation). The reason is that our batch buffer can preserve essential data, with no extra burden on network transmission.

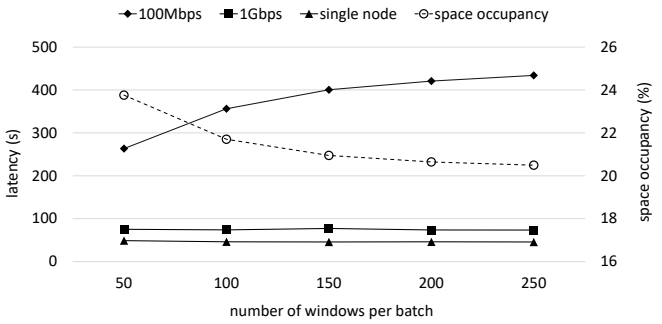

Fig. 10. The effect of batch size on latency and space usage.

Integration of diverse compression schemes. CompressStreamDB is an open system positioned to integrate diverse compression schemes. Although CompressStreamDB currently contains only eight representative compression algorithms covering most cases, more compression schemes can be easily integrated into our system, as long as they are beneficial to the system. For example, for bitmap encoding, we can integrate its further extension, PLWAH [41, 39], into our system. Experiments show that if we use only PLWAH as our single compression method, its transfer time is 30.2% longer than our current design. When employing PLWAH in the adaptive compression model, we can reduce 10.0% transmission time, and improve the overall performance of the system by 13.4%.

VIII. Related Work

Data stream optimization. Due to the increasing demand for real-time processing of vast amounts of data, many studies have been devoted to optimize the performance of stream systems [3, 55, 56, 28, 57, 58, 59, 50, 15]. To name a few, Koliousis et al. [14] developed Saber, a window-based hybrid stream processing for discrete CPU-GPU architectures. Scabbard [3] is a recently proposed single node optimized stream processing engine focusing on fault-tolerance aspect. Li et al. [28] proposed a framework called TRACE that allows compression on traffic monitoring streams. Pekhimenko et al. [27] proposed TerseCades, adopting an integer compression method and a floating-point number compression method to enable direct processing on compressed data. However, none of them utilize the diversity of lightweight data compression technology in stream processing and take multi-layer transmission scenario with complex factors into consideration. This is one of the biggest differences between CompressStreamDB and previous works.

Processing on compressed data. CompressStreamDB's direct SQL query processing on compressed data is a main feature that significantly reduce both time and space overhead in stream processing. Data compression [60, 49, 13, 61, 62, 39, 40, 41, 63, 64, 65, 66, 67, 68, 69] has been proved to be an effective approach to increase the bandwidth utilization and resolve the memory stalls. Wang et al. [40] developed inverted list compression in memory. Deliège and Pedersen [41] optimized space and performance for bitmap compression. Wang et al. [39] conducted an experimental study between bitmap and inverted list compressions. Fang et al. [13] analyzed common compression algorithms, while Przymus and Kaczmarski [49] explored how to select an optimal compression method for time series databases. Picewise linear approximation (PLA) [70] can be applied to approximately answer continuous queries directly in compressed streams, Sprintz [71] introduces a four part composite compression algorithm for time-series data. Many works [65, 67, 72, 66, 69, 68] used hardware such as GPU and FPGA to optimize data compression. As for processing directly on compressed data [20, 21, 19, 22, 23, 73, 74, 75, 76, 77], this technology can provide efficient storage and retrieval of data. For example, Chen et al. [77] proposed a memory-efficient optimization approach for large graph analytics, which compresses the intermediate vertex information with Huffman coding or bitmap coding and queries on the partially decoded data or directly on the compressed data. Li et al. [78, 79, 80] presented compression methods for very large databases, with aggregation operating directly on compressed datasets. Succinct [22] enables efficient queries directly on a compressed representation of data. Other works [81, 82, 19, 21] focused on the direct processing of other compressed storage structures such as graphs. Different from these studies, our work is the first fine-grained stream processing engine that can query compressed streams without decompression.

IX. Conclusion

Stream processing technology is prevalent in the big data area. With the increasing scale of stream data, stream processing systems face tremendous time and space pressure. We propose CompressStreamDB, which applies compression algorithms in stream processing to improve the system performance. In detail, we involve eight lightweight compression algorithms in CompressStreamDB, and can obtain better performance than that without compression. Experiments show that on three real-world datasets, CompressStreamDB can achieve 3.24× throughput improvement and 66.0% lower latency, along with 66.8% space savings.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (62172419, 61732014, and 62072458), CCF-Tencent Open Research Fund, and Beijing Nova Program. This work is also supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Programme FCP-SUTD-RG-2022-005, FCP-SUTD-RG-2022-006, and a SUTD Start-up Research Grant (SRT3IS21164). Feng Zhang is the corresponding author of this paper.

References

[1] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, "Rhino: Efficient management of very large distributed state for stream processing engines," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2471–2486.

[2] G. Van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020.

[3] G. Theodorakis, F. Kounelis, P. Pietzuch, and H. Pirk, "Scabbard: Single-node fault-tolerant stream processing," *Proc. VLDB Endow.*, vol. 15, no. 2, p. 361–374, oct 2021.

[4] F. Zhang, C. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "Fine-grained multi-query stream processing on integrated architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2303–2320, 2021.

[5] "State of IoT 2021," https://iot-analytics.com/number-connected-iot-devices/, 2021.

[6] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter, "Real-time stream processing for big data," *it-Information Technology*, vol. 58, no. 4, pp. 186–194, 2016.

[7] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The system s declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1123–1134.

[8] M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Akrivi Vlachou, "Stream processing languages in the big data era," *ACM Sigmod Record*, vol. 47, no. 2, pp. 29–40, 2018.

[9] P. Deutsch *et al.*, "Gzip file format specification version 4.3," 1996.

[10] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[11] P. Elias, "Universal codeword sets and representations of the integers," *IEEE transactions on information theory*, vol. 21, no. 2, pp. 194–203, 1975.

[12] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[13] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 670–680, 2010.

[14] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 555–569.

[15] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 633–647.

[16] "A third of the internet is just a copy of itself," https://www.businessinsider.com/30-percent-of-the-internet-is-duplicates-2013-12, 2013.

[17] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer, "Network-aware operator placement for stream-processing systems." in *ICDE*, vol. 6, 2006, p. 49.

[18] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: A stream-processing framework for interactive rendering on clusters," *ACM Trans. Graph.*, vol. 21, no. 3, p. 693–702, jul 2002.

[19] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1522–1535, 2018.

[20] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Zwift: A programming framework for high performance text analytics on compressed data," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 195–206.

[21] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Enabling efficient random access to hierarchically-compressed data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1069–1080.

[22] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 337–350.

[23] A. Khandelwal, *Queries on Compressed Data*. University of California, Berkeley, 2019.

[24] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du, "CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases," in *SIGMOD*, 2022.

[25] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *PVLDB*, 2004.

[26] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.

[27] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "Tersecades: Efficient data compression in stream processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 307–320.

[28] T. Li, L. Chen, C. S. Jensen, and T. B. Pedersen, "Trace: Real-time compression of streaming trajectories in road networks," *Proceedings of the VLDB Endowment*, vol. 14, no. 7, pp. 1175–1187, 2021.

[29] R. Stephens, "A survey of stream processing," *Acta*

Informatica, vol. 34, no. 7, pp. 491–541, 1997.

[30] P. Córdova, "Analysis of real time stream processing systems considering latency," *University of Toronto patricio@ cs. toronto. edu*, 2015.

[31] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov *et al.*, "Microsoft cep server and online behavioral targeting," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1558–1561, 2009.

[32] "Apache Storm," http://storm.apache.org/., 2021.

[33] "Apache Flink," http://flink.apache.org/., 2021.

[34] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[35] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *2012 21st international conference on parallel architectures and compilation techniques (PACT)*. IEEE, 2012, pp. 377–388.

[36] P. A. Alsberg, "Space and time savings through large data base compression and dynamic restructuring," *Proceedings of the IEEE*, vol. 63, no. 8, pp. 1114–1122, 1975.

[37] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 671–682.

[38] M. A. Roth and S. J. Van Horn, "Database compression," *ACM Sigmod Record*, vol. 22, no. 3, pp. 31–39, 1993.

[39] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 993–1008.

[40] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson, "MILC: Inverted list compression in memory," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 853–864, 2017.

[41] F. Deliège and T. B. Pedersen, "Position list word aligned hybrid: optimizing space and performance for compressed bitmaps," in *Proceedings of the 13th international conference on Extending Database Technology*, 2010, pp. 228–239.

[42] S. Zhang, Y. Mao, J. He, P. M. Grulich, S. Zeuch, B. He, R. T. Ma, and V. Markl, "Parallelizing intra-window join on multicores: An experimental study," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2089–2101.

[43] H. Ziekow and Z. Jerzak, "The DEBS 2014 grand challenge," in *DEBS*, 2014.

[44] "Smart home statistics," https://www.statista.com/outlook/dmo/smart-home/united-states, 2021.

[45] "More google cluster data," https://ai.googleblog.com/2011/11/more-google-cluster-data.html, 2011.

[46] V. Gulisano, Z. Jerzak, R. Katerinenko, M. Strohbach, and H. Ziekow, "The debs 2017 grand challenge," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 271–273.

[47] V. Gulisano, Z. Jerzak, P. Smirnov, M. Strohbach, H. Ziekow, and D. Zissis, "The debs 2018 grand challenge," in *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 191–194.

[48] C. Mutschler, H. Ziekow, and Z. Jerzak, "The debs 2013 grand challenge," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, 2013, pp. 289–294.

[49] P. Przymus and K. Kaczmarski, "Compression planner for time series database with gpu support," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 2014, pp. 36–63.

[50] S. Zhang, J. He, A. C. Zhou, and B. He, "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 705–722.

[51] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 139–152.

[52] "Alibaba cloud," https://www.alibabacloud.com, 2022.

[53] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013.

[54] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," *IEEE Transactions on Cloud Computing*, 2014.

[55] X. Ren, L. Shi, W. Yu, S. Yang, C. Zhao, and Z. Xu, "Ldp-ids: Local differential privacy for infinite data streams," *arXiv preprint arXiv:2204.00526*, 2022.

[56] B. Zhao, H. van der Aa, T. T. Nguyen, Q. V. H. Nguyen, and M. Weidlich, "Eires: Efficient integration of remote data in event stream processing," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2128–2141.

[57] Y. Zhou, B. C. Ooi, and K.-L. Tan, "Disseminating streaming data in a dynamic environment: an adaptive and cost-based approach," *The VLDB Journal*, vol. 17, no. 6, pp. 1465–1483, 2008.

[58] C. Lin, E. Boursier, and Y. Papakonstantinou, "Plato: approximate analytics over compressed time series with tight deterministic error guarantees," *arXiv preprint arXiv:1808.04876*, 2018.

[59] Y. Zhou, A. Salehi, and K. Aberer, "Scalable delivery of

stream query result," in *Proceedings of 35th International Conference on Very Large Data Bases (VLDB 2009)*, no. CONF. VLDB, 2009.

[60] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled cpu-gpu architectures," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 329–340, 2014.

[61] K. Sayood, *Introduction to data compression*. Morgan Kaufmann, 2017.

[62] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Computing Surveys (CSUR)*, vol. 19, no. 3, pp. 261–296, 1987.

[63] C. Lin, *Accelerating Analytic Queries on Compressed Data*. University of California, San Diego, 2018.

[64] C. Lin, J. Wang, and Y. Papakonstantinou, "Data compression for analytics over large-scale in-memory column databases," *arXiv preprint arXiv:1606.09315*, 2016.

[65] J. Tian, S. Di, X. Yu, C. Rivera, K. Zhao, S. Jin, Y. Feng, X. Liang, D. Tao, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data on gpus," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 283–293.

[66] C. Rivera, S. Di, J. Tian, X. Yu, D. Tao, and F. Cappello, "Optimizing huffman decoding for error-bounded lossy compression on gpus," *arXiv preprint arXiv:2201.09118*, 2022.

[67] C. Zhang, S. Jin, T. Geng, J. Tian, A. Li, and D. Tao, "Ceaz: accelerating parallel i/o via hardware-algorithm co-designed adaptive lossy compression," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.

[68] S. Jin, C. Zhang, X. Jiang, Y. Feng, H. Guan, G. Li, S. L. Song, and D. Tao, "Comet: a novel memory-efficient deep learning training framework by using error-bounded lossy compression," *arXiv preprint arXiv:2111.09562*, 2021.

[69] S. Jin, S. Di, J. Tian, S. Byna, D. Tao, and F. Cappello, "Improving prediction-based lossy compression dramatically via ratio-quality modeling," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 2494–2507.

[70] Y. Zhou, Z. Vagena, and J. Haustad, "Dissemination of models over time-varying data," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 864–875, 2011.

[71] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time series compression for the internet of things," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, pp. 1–23, 2018.

[72] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, 2016.

[73] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 459–475, 2021.

[74] Z. Pan, F. Zhang, Y. Zhou, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Exploring data analytics without decompression on embedded GPU systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1553–1568, 2021.

[75] F. Zhang, Z. Pan, Y. Zhou, J. Zhai, X. Shen, O. Mutlu, and X. Du, "G-TADOC: Enabling efficient GPU-based text analytics without decompression," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1679–1690.

[76] F. Zhang, J. Zhai, X. Shen, D. Wang, Z. Chen, O. Mutlu, W. Chen, and X. Du, "TADOC: Text analytics directly on compression," *The VLDB Journal*, vol. 30, no. 2, pp. 163–188, 2021.

[77] X. Chen, M. Minutoli, J. Tian, M. Halappanavar, A. Kalyanaraman, and D. Tao, "Hbmax: Optimizing memory efficiency for parallel influence maximization on multicore architectures," *arXiv preprint arXiv:2208.00613*, 2022.

[78] J. Li, D. Rotem, and H. K. Wong, "A new compression method with fast searching on large databases," 1987.

[79] J. Li, D. Rotem, and J. Srivastava, "Aggregation algorithms for very large compressed data warehouses," in *VLDB*, vol. 99, 1999, pp. 651–662.

[80] J. Li and J. Srivastava, "Efficient aggregation algorithms for compressed data warehouses," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 3, pp. 515–529, 2002.

[81] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 157–168.

[82] H. Maserrat and J. Pei, "Neighbor query friendly compression of social networks," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 533–542.