Data-Aware Adaptive Compression for Stream Processing

Yu Zhang, Feng Zhang, Hourun Li, Shuhao Zhang, Xiaoguang Guo, Yuxing Chen, Anqun Pan, and Xiaoyong Du

***Abstract*—Stream processing has been in widespread use, and one of the most common application scenarios is SQL query on streams. By 2021, the global deployment of IoT endpoints reached 12.3 billion, indicating a surge in data generation. However, the escalating demands for high throughput and low latency in stream processing systems have posed significant challenges due to the increasing data volume and evolving user requirements. We present a compression-based stream processing engine, called CompressStreamDB, which enables adaptive fine-grained stream processing directly on compressed streams, to significantly enhance the performance of existing stream processing solutions. CompressStreamDB utilizes nine diverse compression methods tailored for different stream data types and integrates a cost model to automatically select the most efficient compression schemes. CompressStreamDB provides high throughput with low latency in stream SQL processing by identifying and eliminating redundant data among streams. Our evaluation demonstrates that CompressStreamDB improves average performance by 3.84× and reduces average delay by 68.0% compared to the state-of-the-art stream processing solution for uncompressed streams, along with 68.7% space savings. Besides, our edge trials show an average throughput/price ratio of 9.95× and a throughput/power ratio of 7.32× compared to the cloud design.**

***Index Terms*—Data compaction and compression, stream processing, edge computing.**

I. Introduction

THE contemporary era of Big Data witnesses extensive use of stream processing technologies [1], [2], [3], [4]. In 2021, active endpoints reached 12.3 billion, reflecting a global 9% increase in connected IoT devices [5]. Notably, low latency and real-time are two of the most prominent features of stream processing, enabling the analysis and querying of vast, continuously incoming data streams, including sensor data [6] and financial transactions [7]. Nevertheless, stream processing systems grapple with the challenge of escalating data volumes as the scale of data streams continues to grow [8]. On one hand, the network bears significant strain from the sheer volume of stream data, hindering real-time functionality in the presence of noticeable transmission delays. On the other hand, a high rate of data arrival can overload server memory, as stream processing systems temporarily store data in memory. Thus, it becomes imperative to explore innovative approaches aimed at alleviating the memory and bandwidth pressures confronting stream processing systems. Data compression, a conventional technique for minimizing file sizes [9], [10], [11], [12], [13], can enhance the efficiency of stream systems and contribute to a reduction in storage requirements when applied in stream processing scenarios.

The utilization of compression in stream processing is pivotal as it enhances the efficiency of stream systems, offering potentially three key advantages. First, stream processing often involves a substantial volume of continuous input data with comparable features, such as timestamps [14], [15], transaction amounts [7], and sensor values [6]. Notably, up to 30% of the data may be duplicated [16]. Through data compression, the redundancy in data can be effectively minimized due to the similarity of input streams, thereby reducing the volume of stream data. Second, in stream processing scenarios, the overhead from memory access and network transfer between nodes surpasses that of computation [17]. Our experiments reveal that transmission can consume up to 70% of the time with a 500 Mbps network. Consequently, it is evident that data compression significantly enhances the efficiency of stream systems. Third, the proven utility of direct computing on compressed data extends to data science applications [18], [19], [20], [21], [22], demonstrating its widespread performance benefits.

However, constructing compressed stream direct processing systems faces three major challenges. First, low latency is crucial for stream processing systems, but the encoding time required by compression methods often introduces significant delays. Experiments in Section II-B reveal that using Gzip may account for up to 90.5% of the overall stream processing time for encoding, an unacceptable overhead. Second, the processing queries and input data in stream processing scenarios are dynamic and subject to modification based on user needs. Some compression algorithms exhibit lower time overheads for compression and decompression, while others offer higher compression ratios.

Manuscript received 7 June 2023; revised 26 February 2024; accepted 4 March 2024. Date of publication 19 March 2024; date of current version 7 August 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62322213 and Grant 62172419, and in part by Beijing Nova Program under Grant 20220484137 and Grant 20230484397. Recommended for acceptance by S. Salihoglu. *(Corresponding author: Feng Zhang.)*

Yu Zhang, Feng Zhang, Hourun Li, Xiaoguang Guo, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China, Beijing 100872, China (e-mail: yu-zhang21@ruc.edu.cn; fengzhang@ruc.edu.cn; lihourun@ruc.edu.cn; xiaoguangguo@ruc.edu.cn; duyong@ruc.edu.cn).

Shuhao Zhang is with the School of Computer Science and Engineering (SCSE), Nanyang Technological University (NTU), Singapore 639798 (e-mail: shuhao.zhang@ntu.edu.sg).

Yuxing Chen and Anqun Pan are with the Database R&D Department, Tencent Inc., Shenzhen 518000, China (e-mail: axingguchen@tencent.com; aaronpan@tencent.com).

Digital Object Identifier 10.1109/TKDE.2024.3377710

To achieve optimal performance, compression must be adaptive to different input workloads, necessitating a careful consideration of the advantages and disadvantages of each algorithm. Third, decompressing data before executing SQL queries can be time-consuming. In our experiments, the time overhead of decompression compared to query execution ranges from 2.09× to 31.37×. This introduces a potential performance impact due to the additional time and space requirements for decompression.

We introduce *CompressStreamDB*, a compression-based stream processing engine designed to overcome the three challenges. First, addressing the need for lightweight and fast compression algorithms to meet low-latency and real-time requirements in stream processing, CompressStreamDB integrates nine lightweight compression methods to enhance efficiency across various input streams. Moreover, deploying the stream processing system on edge devices brings the processor closer to data sources, facilitating accelerated data processing. Second, we introduce a fine-grained adaptive compression algorithm selector capable of dynamically choosing the compression algorithm that provides optimal performance benefits for input streams with varying features. Our system incorporates a cost model that guides the selector's decisions as the workload shifts, considering properties such as the value range and degree of repetition in the input data. This model estimates the time consumed by each compression algorithm, enabling the selector to choose the most efficient one. Third, we propose a method enabling direct querying of compressed data if the data are aligned in memory, thereby avoiding decompression costs. This approach applies query operations to compressed data with minimal modification, if hte data maintain their structure after compression. Additionally, we view lightweight decompression-required techniques as a specific case, integrable into CompressStreamDB. Our preliminary work has been presented in [23]. In this paper, we add new platform, new dataset, new compression algorithm, and new evaluation. Specifically, the new idea of applying edge devices is valuable compared to the cloud. Edge devices show potential in stream processing because they have lower costs and can be deployed close to data sources. We analyze the cost and power benefits of edge devices in detail.

We conducted experiments in both cloud and edge environments, employing four widely-used datasets with varying properties. The cloud platform utilized an Intel Xeon Platinum 8269CY 2.5 GHz CPU, while the edge platform employed a Raspberry Pi 4B. Our experimental results demonstrate that CompressStreamDB outperforms the state-of-the-art stream processing approach, achieving maximum system efficiency. CompressStreamDB exhibits a throughput increase of 3.84× and an average latency reduction of 68.0%. In terms of space savings, CompressStreamDB reduces data storage needs by 68.7%. Furthermore, the edge platform exhibits a throughput/price ratio that is 9.95× higher than the cloud platform, while its throughput/power ratio is 7.32× higher than that of the cloud platform.

Overall, we make the following three major contributions.

- We develop a compressed stream processing engine featuring diverse lightweight compression methods applicable across various scenarios.
- We introduce a system cost model to guide compressed stream processing and design an adaptive compression algorithm selector based on this model.
- We devise a processing approach that directly executes SQL queries on compressed streams and conducts comprehensive experiments to validate its effectiveness.

II. Background

A. Stream Processing and Streaming SQL

Stream processing: Stream processing, a term in data science, focuses on the real-time processing of continuous streams of data, events, and messages. It encompasses various systems, including reactive systems, dataflow systems, and specific classes of real-time systems [31]. The query is the SQL statement used for data processing, which can be further subdivided into different operators. In the stream processing context, a stream comprises a sequence of tuples, where each tuple represents an event with elements like timestamp, amounts, and values. Tuples collectively form batches, which represent processing blocks containing a specific number of tuples. Within a batch, we use the term *column* to denote elements of different tuples in the same field. Stream processing finds extensive applications in scenarios requiring minimal latency, real-time response with minimal overhead (e.g., *risk management* [32] and *credit fraud detection* [14]), and predictable and approximate results (e.g., *SQL queries on data streams* [15] and *click stream analytics* [33]).

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* [34] and *Apache Flink* [35], 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, but to ensure accurate query results, our system exclusively considers lossless compression algorithms. Lossless compression algorithms can be categorized into heavyweight and lightweight compression. Noteworthy heavyweight compression algorithms, such as Lempel-Ziv algorithms [10], [12] and Huffman encoding [9], [36], offer high compression ratios but involve complex encoding and decoding processes, causing significant time overhead. Given the real-time and low-latency

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 [24]	Delete leading zeros of each value with fixed bits.
	Null Suppression with Variable Length [24]	Delete leading zeros of each value with variable bits.
Lazy	Base-Delta Encoding [25]	Encode values as their delta values from base value.
	Run Length Encoding [26]	Encode values with their run lengths.
	Dictionary [27]	Maintain a dictionary of the distinct values.
	Bitmap [13, 28, 29, 30]	Encode each distinct value as a bit-string.
	PLWAH [28, 29]	Compression on bitmap.

requirements of stream processing, which cannot tolerate prolonged delays, our exploration of heavyweight compression algorithms revealed that while they may achieve higher compression ratios, they also result in longer (de)compression times, offering limited improvement in the performance and stability of the stream system. In preliminary experiments, we utilized the commonly used compression tool Gzip in stream processing systems. However, the system with Gzip spent 90.5% of the total time in compression and less than 10% in transmission. Despite its high compression ratio and low transmission time, the compression time overhead could lead to system delays or even pauses. Hence, we advocate the use of lightweight compression algorithms to expedite stream processing.

Lightweight compression: Lightweight compression represents a trade-off between compression ratio and time, employing relatively simple encoding methods. In contrast to heavyweight compression algorithms, lightweight alternatives sacrifice some compression ratio for faster (de)compression times. We examined a range of works [11], [13], [24], [25], [26], [27], [28], [29], [30] on lightweight compression algorithms, covering most commonly used ones. Each algorithm has its unique advantages and disadvantages, which are appropriate to data streams with different characteristics. For instance, Elias Gamma encoding and Elias Delta encoding [11] are suitable for small and large numbers, respectively. Run Length Encoding [26] is effective for data with more repetition. The effectiveness of Null Suppression [13] depends on redundant leading zeros in the elements. Bitmap and its extensions [28], [29], [30] are suitable for compressing data with few distinct values.

Eager and lazy compression: We categorize these lightweight compression algorithms into two groups: eager compression and lazy compression [37]. In Table I, we provide a summary of nine common lightweight compression algorithms with the two categories. Eager compression algorithms compress subsets of input tuples as soon as they arrive, allowing them to process each tuple without waiting. On the other hand, lazy compression algorithms wait for the entire data batch before compression. The advantage of eager algorithms lies in their ability to process each tuple in real-time, while the advantage of lazy algorithms is their capacity to leverage the similar redundancy in large datasets, achieving a higher compression ratio.

C. Edge Computing

Compared to traditional cloud computing architecture, edge computing pushes computing resources and services to the Internet of Things, end devices, and user terminals to achieve real-time data processing and response, reducing the pressure on bandwidth, storage, and computing resources caused by centralized computing. It meets the requirements of low latency, high bandwidth, and data security. With the rapid development of Internet of Things, cloud computing, Big Data, and other relevant technologies, edge computing has been widely used in fields such as smart homes [38], smart cities [39], industrial Internet [40], and intelligent transportation [41].

The lightweight development of edge devices is the current and future trend. With the surge of the mobile Internet, edge computing has extended beyond personal computers and servers to encompass mobile devices, including edge computing platforms based on mobile phones [42]. Following the rapid progress of Internet of Things (IoT) technology, edge computing has expanded into the realm of low-power and embedded devices, such as Raspberry Pi 4B [43] and microcontrollers [44]. These devices, characterized by smaller size, lower power consumption, and versatility to run in various environments, support multiple communication protocols and data processing algorithms. They can perform tasks like anomaly detection [45], exoskeletons [46], voice activation [47], object detection [48], and more. The edge device market is anticipated to grow, driven by the increasing demand for real-time data processing and analysis, as well as the need for low-latency, high-bandwidth, and secure data transmission. According to IDC's forecast, the global number of connected devices is expected to surpass 8 billion by 2025, with approximately 40% of these devices situated at the edge [49].

In stream data processing, edge computing can handle data at the point of generation, alleviating the burden of data transmission and storage. This enables faster real-time analysis of data. For instance, it finds applications in real-time detection systems based on sensors [50], video stream analysis [51], logistics tracking systems [52], network security detection [53], and data processing for autonomous vehicles [48]. Edge devices can efficiently compress and process stream data, meeting the real-time and accuracy requirements of practical tasks.

III. MOTIVATION

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 nature of stream data manifests in three key aspects. First, the attributes of stream data, such as value ranges and repetition degrees, can change unpredictably, impacting the achievable compression ratios of different algorithms. Second, factors like data generation speed and network delays cause fluctuations in the arrival rate of stream data, influencing system waiting times. Third, it is impossible to predict how frequently data properties can change, necessitating a balance between efficiency and overhead when determining the re-decision frequency for dynamic processing.

Differences from column compression in databases: The disparities between stream processing and traditional databases call for innovative compression strategies. Traditional databases perform holistic operations post full data scanning, enabling the selection of compression algorithms based on overall data attributes. In addition, compression in databases focuses more on compression ratios rather than low-latency real-time processing. Conversely, stream processing involves real-time, unbounded data streams that constantly evolve, necessitating dynamic updates to compression methods to adapt to these changes.

C. Case Study

Fig. 1 shows a motivation example to illustrate the comparison between static processing in traditional databases and dynamic processing in a stream scenario. It uses a case study from smart grids [54]. The smart home market is projected to reach a volume of 51.23 billion by 2026, with an estimated 84.9 million active homes and an annual growth rate of 11.7% [55]. This dataset comprises over 4,055 million energy consumption measurements collected from smart plugs installed in private households. It encompasses data from 2,125 plugs distributed across 40 houses over one month. Seven attributes are contained

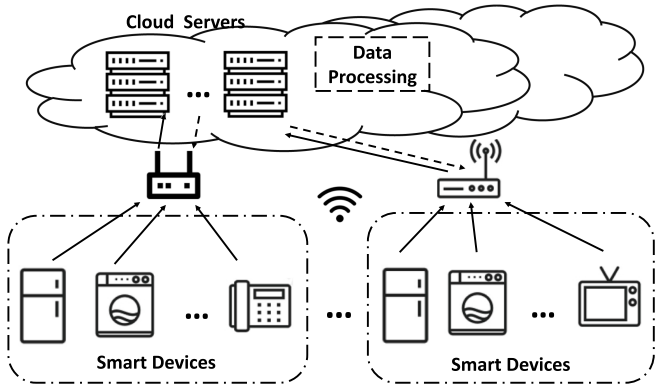

Fig. 1. Example of smart grids.

in the dataset, including timestamps, the measurement value, the ids of the plug and house, etc.

Dynamic characteristics: The characteristics of real-time electricity consumption data in the stream are consistently dynamic. Fluctuations in consumption peaks and troughs, household habits, and usage patterns cause constant shifts in the data stream. For instance, when a household generates substantial power consumption data within a short span, this data may manifest as repeated house IDs with changing plug IDs and values within the stream. Similarly, during peak hours when multiple households are consuming electricity simultaneously, house IDs might frequently change while timestamps remain constant.

Opportunity: In traditional database processing, analysis of the content occurs beforehand, enabling pre-determination of the processing method. Conversely, in a stream scenario, data arrives continuously, necessitating immediate processing as events appear. Stream processing methods lack access to complete information in advance due to the dynamic nature of input data streams. Consequently, real-time processing methods need adaptive changes as the input stream evolves. As demonstrated in Section VII, our solution, CompressStreamDB, significantly outperforms static processing methods, highlighting its superior performance in dynamic environments.

D. Widespread Use of Compressed Stream Direct Processing

Our compressed stream direct processing solution offers extensive applicability across numerous stream applications. We present illustrative examples showcasing its versatility in various scenarios.

- *IoT sensor data* from the smart grid domain [54] is an underlying scenario, which involves the analysis of energy consumption measurements. This aims to offer short-term load predictions and real-time demand management. However, dynamic workload fluctuations present real-time challenges. Leveraging compressed stream direct processing can significantly enhance throughput, enabling efficient processing of large data volumes within short time frames.
- *Real-time decision* in linear road benchmark [56] specifies an expressway variable tolling system. Each vehicle on the

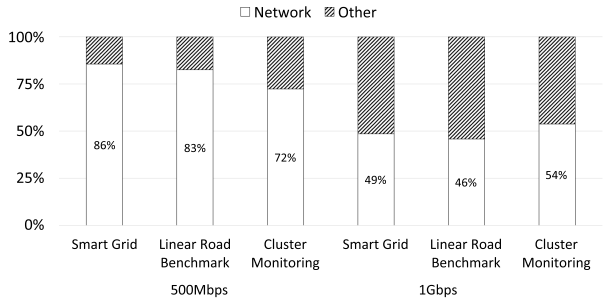

Fig. 2. Total time breakdown.

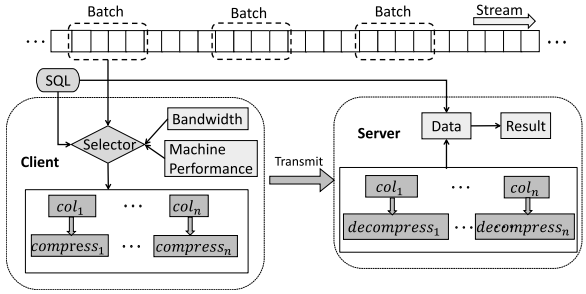

Fig. 3. CompressStreamDB framework.

highway transmits its location through sensors, which are utilized to calculate tolls based on the specific road section. Lower tolls incentivize the use of less congested roads. Our solution enables the system to efficiently process substantial volumes of streaming vehicle location data, facilitating more effective decision-making in toll adjustments.

- *Cluster management* [57] 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.

Various other real-time stream applications, including manufacturing equipment detection [58], ship behavior prediction [59], and temporal event sequence detection [60], necessitate efficient stream processing. Fig. 2 illustrates the breakdown of time utilization in these applications. The complete bar denotes the overall duration of uncompressed stream processing, with the white segment representing the portion of time consumed by network transmission. Notably, with a 500 Mbps bandwidth network, network transmission occupies over 70% of the total time. Even on a 1 Gbps network, transmission still accounts for about 50% of the total time. This highlights the bottleneck created by transmission time in stream applications, underscoring the critical need for the advantages offered by compressed stream direct processing.

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 addresses the challenges mentioned in Section I, effectively mitigating time and space overhead in stream processing. It dynamically selects compression algorithms and seamlessly integrates them into stream processing.

Structure: The CompressStreamDB framework comprises two core components: the client and server, depicted in Fig. 3. The client has a compression algorithm selector based on the cost model. This selector is tasked with data collection and optimal compression algorithm selection. Note that the term "client" refers to devices seeking compressed stream processing in collaboration with the server, encompassing data sources like sensors or smartphones, or intermediate nodes handling data. Leveraging lightweight compression algorithms, even resource-constrained devices like data sources can perform compression. Consequently, our system accommodates a multi-layer architecture with multiple compression client layers, while the client-server setup represents a simplified model. Compression functionalities are deployed on the client side. In a distributed architecture, individual clients perform independent compression without coordination. In scenarios where a single query's input data originates from multiple clients, each client autonomously determines its compression strategy based on the specific data characteristics. The server manages the processing of queries on compressed stream data, housing the kernel functions necessary for executing these queries. It's important to note that while CompressStreamDB is primarily designed for direct processing of compressed stream data, it does not dismiss the inclusion of efficient compression algorithms that require decompression. They can also be integrated into the system and should not be ignored.

Scenario: In a streaming scenario, CompressStreamDB dynamically selects compression algorithms and conducts fine-grained compression-based stream processing based on specified parameters like network throughput and performance metrics of clients and servers. The compression algorithm selection aims to optimize the system's overall performance, specifically to minimize the total processing time.

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. Prior to compression, the selector preloads the data and identifies the compression algorithm that ensures optimal performance. This decision-making process relies on our comprehensive cost model, considering various factors from machine metrics and network conditions to the effectiveness and cost of compression algorithms (refer to Section IV-C for details). Our system operates at batch granularity, employing distinct compression algorithms for each data column, as discussed in Section II. Subsequently, the compressed data is transmitted to the server, where it is processed alongside the corresponding SQL queries.

Batch: In CompressStreamDB, stream data are processed at batch granularity. The batch size operates independently from

the window size in streaming SQL. The window size pertains to a range concept within SQL, whereas the batch size represents the processing granularity of the query engine [1], [14], [15]. It's worth noting that a batch can be smaller than a window or encompass multiple windows. The batch size setting plays a dual role since the growth of batch size can increase both the latency and the compression ratio. We determine the batch size using dynamic sampling, where its overhead can be amortized during stream processing. Users can specify and adjust the batch size based on actual requirements. Experimental insights can be found in Section VII.

Flexibility: CompressStreamDB stands as a highly flexible system, facilitating not only the support for nine existing data compression methods but also the seamless integration of additional compression algorithms. This flexibility is designed to effectively address the increasing demand for stream data processing. The flexibility of CompressStreamDB empowers it to adeptly handle and analyze diverse data streams, varying in types, scales, and rates. This capability allows the system to better align with the demands of real-world tasks.

Portability: The client of CompressStreamDB is highly portable, readily adaptable to diverse devices, including embedded edge devices like Raspberry Pi 4B, requiring minimal modifications. This versatility stems from the lightweight and high-speed algorithms implemented in the client, demanding minimal computational power. The server of CompressStreamDB is portable and can be deployed to diverse high-performance devices. Its direct SQL operators are universally designed and can be adopted to different compression algorithms and platforms. The portability of CompressStreamDB empowers its adaptability across diverse devices and platforms, rendering it well-suited for resource-constrained environments, notably in edge computing scenarios.

B. Compressed Stream Processing

Adaptive processing for dynamic workload: In CompressStreamDB, we dynamically process the input data stream using our selector. As detailed in Section II, stream data processing is achieved through SQL queries, treating a batch as the minimum processing unit. The system predominantly employs 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 system dynamically reselects compression algorithms for data columns using 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. The batch size and window size in our system are independent of each other. Changes in compression do not directly affect the windows. In smaller windows within a batch, reselecting compression affects multiple windows. However, in larger windows that span multiple batches, reselection impacts only subsequent batches, without requiring recompression of previously compressed batches.

Supported data types: CompressStreamDB not only incorporates lightweight compression algorithms for integers, but also supports operations on floating-point numbers and strings. Floating-point items can be converted into integers via multiplying by a factor of 10^n [13]. The n here denotes the maximum number of decimal places within the data column. For instance, in the context of measured values in smart grids [54], the values include numbers such as {3.216, 11.721, 9.8}. With a maximum of 3 decimal places, all data can be scaled by a factor of 10^3, resulting in {3216, 11721, 9800}. Given that data columns typically exhibit closely aligned decimal places, overflow is uncommon in most scenarios. Overflow risks arise only when the converted integer exceeds 2^{31} (the limit of a 32-bit integer). In such cases, we recommend either utilizing a 64-bit integer representation or exploring the option of employing the dictionary encoding method. For strings, they can be mapped to integers using dictionary encoding, which is a widely-used method with marginal overhead [61], [62], [63]. Our evaluation in Section VII covers data types of integer, floating-point, and string, all of which are encoded as integers before loading. After unified data encoding, different types of data can be processed in CompressStreamDB.

Query without decompression: Decompression is employed to restore the original data. CompressStreamDB avoids decompression as much as possible, thus reducing time, memory access, and accelerating 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. Our SQL operators are specially designed for compressed data processing. These operators can accept parameters of the number of bytes each compressed column occupies. For instance, if the original column holds 4 bytes per element but is compressed to 1 B per element, our operators handle this column by reading and writing only 1 B for each entry. Despite various compression algorithms encoding raw data differently, their results ultimately conform to a fixed format. As long as the compressed format meets these three conditions, direct processing of the compressed data is supported. This universal design avoids the complexity of developing separate operator kernels for different compression methods. Our implementation is portable to diverse devices cause it does not require any special hardware support.

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 B, and $col3'$ is 1 B. 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

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 CPU FLOPS of client and server.
B_{client}&B_{server}	The memory bandwidth of client and server.
$T^{com,\tau}_{memory}$&$T^{decom,\tau}_{memory}$	The number of instructions for memory accesses.
$T^{com,\tau}_{operation}$&$T^{decom,\tau}_{operation}$	The number of instructions for computation.

corresponding compressed stream processing operators based on the compression format.

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, enhancing system efficiency. It's important to note that CompressStreamDB accommodates diverse compression schemes. Algorithms requiring lightweight decompression should be considered if their performance benefits outweigh the decompression overhead.

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], [64] 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 consists of four primary stages: compression, transmission, decompression, and query processing. To model the costs across these stages, we develop a system cost model. Table II outlines the key parameters utilized in our system cost model.

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 (1).

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

In the following part of this section, we delve into the considerations of cost model, including machine metrics, network conditions, and the efficiency of compression techniques. We focus on modeling the time consumption across these four aspects.

1) Compression time: For the processing batch, $Size_T$ represents the number of bytes per tuple, while $Size_B$ means the number of tuples per batch, thus there are $Size_T \cdot Size_B$ bytes for each batch. For a chosen compression algorithm τ, $T^{com,\tau}_{memory}$ denotes the number of instructions used for memory accesses during compression, while $T^{com,\tau}_{operation}$ represents the number of instructions used for computation. Then the $t_{compress}$ can be defined by (2).

$$t_{compress} = \alpha \cdot t_{wait} + \max\left(\frac{T^{com,\tau}_{memory} + T^{com,\tau}_{operation}}{N_{client}}, \frac{Size_T \cdot Size_B}{B_{client}}\right). \tag{2}$$

N_{client} means the CPU FLOPS of the client, and B_{client} is the memory bandwidth of the client, which can be obtained from the hardware specifications. The client is responsible for compression. If the compression program is a memory-intensive program for the client, $\frac{Size_T \cdot Size_B}{B_{client}}$ is larger. Otherwise, if it is a compute-intensive program, $\frac{T^{com,\tau}_{memory} + T^{com,\tau}_{operation}}{N_{client}}$ is larger.

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 the variable α is defined in (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 denote the time allocated to transmission as t_{trans}. r represents the compression ratio achieved by the selected algorithm as outlined in Section V. Then, we use $\frac{Size_T}{r}$ to represent the tuple size after compression. Considering $Size_B$ as the number of tuples within a batch, the byte size required for transmitting the compressed batch equates to $\frac{Size_T \cdot Size_B}{r}$. If network bandwidth suffices and queuing delay remains negligible, t_{trans} can be expressed as shown in (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 (5).

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

3) Decompression time: Considering the importance of future-proof compression algorithms that require decompression pre-processing, our system retains the flexibility to incorporate these methods. For a chosen compression algorithm τ, $T^{decom,\tau}_{memory}$ symbolizes the number of instructions executed for memory access during decompression, while $T^{decom,\tau}_{operation}$ denotes the count of computational instructions. This allows us to define $t_{decompress}$ as per (6).

$$t_{decompress} = \beta \cdot \max\left(\frac{T^{decom,\tau}_{memory} + T^{decom,\tau}_{operation}}{N_{server}}, \frac{Size_T \cdot Size_B}{B_{server}}\right). \quad (6)$$

N_{server} relates to the CPU FLOPS of the server, which is similar to N_{client}, and B_{server} means the memory bandwidth of the server. If the decompression program leans towards memory-intensive operations on the server, $\frac{Size_T \cdot Size_B}{B_{server}}$ is larger. Alternatively, if it's more compute-intensive, $\frac{T^{decom,\tau}_{memory} + T^{decom,\tau}_{operation}}{N_{server}}$ is larger. β indicates whether τ needs decompression, which is defined in (7).

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

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 (8).

$$t_{query} = t^{query}_{operation} + \frac{t^{query}_{memory}}{r'}. \quad (8)$$

Please note that r' signifies the compression ratio during the query execution, distinct from the compression ratio denoted as r. The determination of r' hinges on whether the server undertakes decompression as part of the query process. Its definition is represented by (9), indicating its role in the query phase.

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

V. Selected Compression Algorithms

CompressStreamDB incorporates nine lightweight compression algorithms, detailed in Section II. This system dynamically chooses among these options at runtime using our innovative system cost model, and partial parameters of cost model relate to the compression algorithms.

A. Preliminaries

We present the system cost model parameters associated with each compression algorithm, delineating their respective advantages and drawbacks, including parameters such as α, β, r, and r'. The parameters $T^{com,\tau}_{memory}$, $T^{com,\tau}_{operation}$, $T^{decom,\tau}_{memory}$, and $T^{decom,\tau}_{operation}$ pertaining to the number of instructions can be directly derived by inspecting the assembly source code. This approach allows for the construction of a tailored cost model specific to each compression algorithm.

In the rest of this section, we will consistently employ the symbol $Size_B$ to signify the processing batch size. Furthermore, we introduce a new symbol, $Size_C$, representing the size of each element in the column. Our discussion on each compression algorithm will introduce a set of parameters related to dataset properties. These parameters will be explicitly described within the context of each compression algorithm, offering specific insights into their characteristics.

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): The idea of Elias Gamma encoding is to use the number of leading zeros to represent the binary valid bits of the data [11]. Given a positive number x, let $L = \lfloor \log_2 x \rfloor$. The Elias Gamma encoding form of x can be represented as L zero bits followed by the binary form of x. Hence, it uses $2 * L + 1$ bits to represent the number x.

Elias Gamma encoding, being a variable-length encoding, lacks a fixed number of bits. However, as discussed in Section IV-B, we align its encoding to ensure consistent byte representation. For a specific data column, $EGDomain$ signifies the maximum byte requirement for Elias Gamma encoding in that column. Consequently, data within this column involve $EGDomain$ bytes during processing. This encoding method might struggle with outliers, necessitating more uniform input data. An occurrence of significant outliers in the input data could notably escalate redundancy across the entire column, potentially impacting efficiency.

The Elias Gamma encoding method processes only positive integers, while our data comprises non-negative integers. To accommodate this, we increment each integer by 1 before compression and decrement by 1 during decompression. This simple adjustment effectively converts non-negative integers for compatibility with the encoding scheme. The compression ratio r of Elias Gamma encoding can be described in (10).

$$r = \frac{Size_C}{EGDomain}. \quad (10)$$

The storage format of Elias Gamma encoding in CompressStreamDB ensures alignment to a consistent byte length while maintaining the structured nature of compressed data. Utilizing this format, CompressStreamDB bypasses decompression. The parameters associated with Elias Gamma encoding include $\beta = 0, r' = r = \frac{Size_C}{EGDomain}$.

- Pros: 1) It is suitable for scenes where small integers are used frequently. 2) It can avoid the overhead of decompression.
- Cons: 1) The speed of Elias Gamma encoding is relatively slower in contrast to other lightweight algorithms, due to the requirement of performing a logarithmic operation before adding leading zeros. 2) It is not good at handling large outliers.

Elias Delta encoding (ED): Elias Delta encoding, a variant of Elias Gamma encoding, extends the process by performing an additional Elias Gamma encoding on the value derived from the

first encoding [11]. Given a positive number x, let $L = \lfloor \log_2 x \rfloor$, and $N = \lfloor \log_2(L+1) \rfloor$. The Elias Delta encoding form of x has 1) N zero bits, followed by 2) $N+1$ bit binary representation of $L+1$ and 3) the last L bits of the binary form of x. Hence, it uses $2 * N + L + 1$ bits to represent the number x.

Same as Elias Gamma encoding, we extend Elias Delta encoding to make it suitable for non-negative integers. In addition, we have also aligned its encoding results. For a given data column, We use $EDDomain$ to represent the maximum number of bytes required for Elias Delta encoding for the elements of this column. Then, the data in this column are all $EDDomin$ bytes during processing. It requires more bits for compressing small values compared to Elias Gamma encoding, but performs better with larger values. For larger integers, the length of Elias Delta encoding approaches entropy, making it nearly optimal. The compression ratio r for Elias Delta encoding can be expressed using (11).

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

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

- Pros: 1) It is more stable than Elias Gamma encoding. 2) It can handle a larger range of values.
- Cons: 1) Its compression process is more complicated, so the compression speed can be slower. 2) When the value is very small, its performance is not as good as Elias Gamma encoding.

Null suppression with fixed length (NS): The null suppression with fixed length method removes leading zeros from the binary representation of the element, efficiently eliminating the redundancy caused by the data type [24]. "With fixed length" means that the elements of the compressed data have the same number of bits.

To estimate the compression effects of null suppression with fixed length and null suppression with variable length, we introduced the $ValueDomain$ array. The size of this array is the same as the batch size. It records the number of bytes required to represent the valid bits of each element in the column. $ValueDomain_{MAX}$ denotes the bytes used by elements after null suppression with fixed-length. The compression ratio r for this method is given by (12).

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

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

- Pros: Its compression is very convenient and can be performed efficiently.
- Cons: It is not good at handling large outliers.

Null suppression with variable length (NSV): Null suppression with variable length is similar to null suppression with fixed length, achieving compression by removing leading zeros [24]. Unlike the fixed-length method, it doesn't mandate a consistent number of bits in the compressed output. Instead, it records the byte length of each encoded value for decompression. In our design, with values ranging between 1 and 4 bytes, we use two bits to signify the byte count per value. Consequently, every group of four elements requires an extra byte to record their lengths.

Compared with null suppression with fixed length, null suppression with variable length has a better compression ratio in most cases. It has a good effect on elements of different sizes. However, when column elements predominantly fall within a narrow range, the additional bytes used to note their lengths can become an overhead that's difficult to overlook.

Utilizing the introduced $ValueDomain$ array in NS, the total number of bytes needed after compression can be derived by summing the values within $ValueDomain$. The compression ratio r for null suppression with variable length is determined by (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: $\beta = 1$, $r' = 1$.

- Pros: 1) It can make better use of space and achieve a higher compression ratio compared to NS. 2) It can handle the situation of large data changes and is not easily affected by outliers.
- Cons: 1) It needs decompression before processing. 2) It needs extra bytes to record the length.

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): Base-Delta encoding is ideal for scenarios with large values and a limited data range, or when differences between values are considerably smaller than the values themselves [25]. This method selects a base value from a series of values and stores it. Each element is represented by its delta value from this base. If the delta value is significantly smaller than the original element, it can be efficiently represented with fewer bytes.

We designate $BDDomain$ to represent the maximum bytes needed for Base-Delta encoding in a data column. The compression ratio r for Base-Delta encoding can be calculated using (14).

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

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

- Pros: It achieves fast compression due to its reliance on basic vector addition and subtraction operations.
- Cons: It is only suitable for data with a small range of variation.

Run length encoding (RLE): Run Length Encoding is a classical compression algorithm [26] effective for datasets featuring

recurring sequences of elements. It efficiently reduces space by compressing repetitive data that occurs periodically.

Suppose the average run length of a column of data batch is represented by $AverageRunLength$. As run-length encoding requires an additional integer variable (4 bytes) to represent the run length, the compression ratio r for run-length encoding is defined in (15).

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

RLE does not maintain byte alignment and disrupts the original data structure, necessitating decompression before processing. Consequently, its parameters are defined as follows: $\beta = 1$, $r' = 1$.

- Pros: 1) Its compression speed is relatively fast. 2) It can achieve a high compression ratio for data if $AverageRunLength$ is high.
- Cons: 1) It needs decompression before processing the data. 2) It only applies to continuously repeated data.

Dictionary (DICT): The dictionary compression algorithm is commonly used to convert larger data into smaller data by establishing a one-to-one relationship [27]. If the number of data types is denoted as $Kindnum$, the compression ratio r of dictionary encoding can be defined using (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: $\beta = 0$, $r' = r = \frac{Size_C}{\lceil \log_2 Kindnum/8 \rceil}$.

- Pros: It has a relatively high compression ratio.
- Cons: It is appropriate for use when there are only a few types of data.

Bitmap: The bitmap compression algorithm is relatively concise, using a bit string to represent the original data [13], [28], [29], [30]. Each bit in the bit string corresponds to a unique element in the original data. If the number of data types is denoted as $Kindnum$, the compression ratio r of bitmap encoding can be defined using (17).

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

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

- Pros: It has fast compression and decompression speed.
- Cons: It is appropriate for use when there are only a few types of data.

Position list word aligned hybrid (PLWAH): To demonstrate the flexibility of CompressStreamDB in compression algorithms, besides the original compression algorithms, we further extend a highly efficient variant of compressed Bitmap, PLWAH, into our system. The evaluation indicates that PLWAH further improves the performance of our system, detailed in Section VII.

PLWAH is an efficient compressed bitmap data structure with a high compression ratio and fast operation capability [28]. When a sequence of elements filled with 1 s or 0 s appears, the PLWAH algorithm compresses them into a single element, thereby achieving a higher compression ratio. Moreover, the first different value after this sequence will also be compressed into the same element. In other words, PLWAH can merge all elements filled with 0 or 1.

Example: We use a simplified 8-bit example to illustrate the compression scheme of PLWAH. Assume that we have 4 Bitmap entries to compress and the original data is [00000000, 00000000, 00000000, 00100000]. Then the compressed data is an 8-bit entry [1 0 011 011]. The first 1 indicates that this word is a compressed fill word. The second 0 indicates that the fill word is filled with 0. The next three digits "011" indicate a literal word with "1" in the third digit (00100000). The last three digits "011" indicate three consecutive 0 fill words. In this way, the original four items are compressed into one item. In the context of a 32-bit representation PLWAH, the first 2 bits signify the word type, the subsequent 5 bits denote the position of the next literal word's "1", and the final 25 bits indicate the number of merged fill words [28]. Note that our bitmap approach is designed to compress data with specific types of values. The compressed bitmap can only have at most one "1", with the remaining bits set to "0" [13]. PLWAH can be applied in such scenarios.

In practice, the most frequently occurring element can be mapped to the element filled with 0 in the bitmap, and the second most frequently occurring element can be mapped to the element filled with 1 in the bitmap. This strategic mapping significantly reduces space allocation for these two elements in the entire column. If we denote the count of the most frequent element as $MostCount$, and the count of the second most frequent element as $SecondCount$, then the compression ratio r of PLWAH is formally defined in (18).

$$r = \frac{Size_B}{Size_B - MostCount - SecondCount}. \tag{18}$$

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

- Pros: It has much higher compression ratio than bitmap.
- Cons: It is appropriate for use when there are only a few types of data.

VI. Implementation

We implement CompressStreamDB with references to [14], [15], [65]. It comprises two primary modules: a client module housing the stream processing compression algorithms and the adaptive selector, and a server module equipped with fundamental SQL operators to process compressed streams. These operators include selection, projection, groupby, aggregation, and join. The server module takes charge of handling these compressed streams efficiently. Additionally, we integrate a profiler into the server, facilitating the collection of key performance metrics, including (de)compression and transmission times. It's important to note that the compression functionality within CompressStreamDB can be turned off, allowing support for processing uncompressed streams. In scenarios involving small-window queries, like handling a single tuple, CompressStreamDB can seamlessly execute uncompressed stream

processing without waiting for the entire data batch. This hybrid processing mode significantly extends the applicability of system across a broad spectrum of stream processing scenarios.

In our batch implementation, a sliding window can extend across multiple batches. To address this challenge, our system incorporates a batch buffer, temporarily storing data from the previous batch. Upon detecting a sliding window that spans across batches, the system awaits the arrival of the subsequent batch. At this point, it retrieves the previously cached batch from the buffer, facilitating the computation of results across sliding windows that cross batch boundaries.

CompressStreamDB can be deployed among different environments. For example, Apache Storm can customize serializer. We can wrap the compression module of CompressStreamDB into a custom serializer, and then embed it into Storm for use. However, this incurs additional challenges such as model integration and Storm 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 for comparison is CompressStreamDB without compression. Our system offers the ability to disable the compression function, allowing uncompressed stream processing. The comparison against this baseline aims to evaluate whether our solution enhances performance in stream systems. To better demonstrate the benefits of our adaptive compression approach, we conducted a performance comparison between our implementation and two high-performance stream processing systems–Saber [14] and FineStream [15]. Notably, both Saber and FineStream operate without employing compression techniques. We implement the Base-Delta encoding compression with reference to TerseCades [66], denoted as "Base-Delta". TerseCades stands as a pioneering exploration of stream processing with data compression, demonstrating its efficacy in this domain. Our work showcases progressive advancements in performance compared to the Base-Delta encoding employed in TerseCades. As TerseCades isn't open-source, we re-implement its functionalities. Comparing the result of our implementation with those presented in the TerseCades paper [66], we observe similar outcomes. For instance, in [66], the system processed queries on the Pingmesh Data [67] achieving a throughput of 37.5MElems/s. In our implementation using Base-Delta encoding, we accomplish a throughput of 37.2MElems/s when querying the Smart Grid Data [54], showing a comparable performance level. Our study extends further by comparing the adaptive compression stream processing capability of CompressStreamDB across nine lightweight compression algorithms. To exhibit the portability of CompressStreamDB, we conduct experiments on both cloud and edge platforms, analyzing and comparing their throughput, power efficiency, and cost efficiency.

Platforms: Our client is equipped with an Intel Xeon Platinum 8269CY 2.5 GHz CPU and 16 GB memory, running Ubuntu 20.04.3 LTS with Java 8. Our server is deployed on both a cloud platform and an edge platform. Their information is as follows.

- Cloud platform. It has the same configuration as the client. The price of the CPU is about $899. Its TDP is 205 W, which indicates that it cannot be used at the edge because of high power consumption. We mainly conduct system performance experiments on this platform, and use the turbostat tool to monitor its power.
- Edge platform. Our edge device is Raspberry Pi 4 Model B [68] . It is equipped with Quad core ARM Cortex-A72 64-bit SoC and 8 GB memory, running Raspberry Pi OS 5.10 with Java 8. Its price is $75, with maximum 6.4 W power consumption [69] . We use the edge device to prove the portability of our system, and show the advantages of using edge device for stream processing. To detect the Raspberry Pi's power consumption, a power meter is attached.

Datasets: Our evaluation incorporates four datasets widely used in previous studies [14], [15], [61], [70], [71], [72], [73], [74], [75], all of which remain relevant in current discussions. For instance, the smart home market is projected to reach 51.23 billion by 2026, growing at an annual rate of 11.7% [55], emphasizing the continued significance of these datasets. The first dataset originates from energy consumption measurements in smart grids [54], capturing data from various devices within a smart grid to enable load predictions and real-time demand management in energy consumption. The second dataset, compute cluster monitoring [57], is derived from a Google cluster, simulating a cluster management scenario. The third dataset, the linear road benchmark [56], records vehicle position events and models a network of toll roads. The fourth dataset is Star Schema Benchmark (SSB) [76]. It contains one fact table, four dimension tables, and thirteen standard queries. We adjust SSB for stream processing. The adaption for more benchmarks are shown in Section VII-E.

Queries: We utilize eight queries to evaluate the performance of adaptive compression in CompressStreamDB. For each dataset, we execute two queries to derive performance metrics, evaluating various processing methods including the baseline, nine lightweight compression algorithms, and CompressStreamDB. These queries are well-established in prior stream processing studies [14], [15], [72], [73], [74], [75]. The specific details of the eight queries are outlined in Table III. *Q1* and *Q2* analyze the anomaly detection in smart grids dataset. *Q3* and *Q4* operate on the linear road benchmark dataset. Queries *Q5* and *Q6* interact with the Google compute cluster monitoring data. *Q7* and *Q8* tackle the Star Schema Benchmark. To adapt to stream processing; we rewrite Q1.1 and Q1.2 of SSB to adapt to stream processing.

In the Smart Grid and Linear Road Benchmark datasets, a batch encompasses 100 windows, and each window contains 1024 tuples. In the case of the Cluster Monitoring dataset, a batch comprises 200 windows, with each window consisting of 512 tuples. Finally, within the Star Schema Benchmark dataset, each batch contains 100 windows, and each window encompasses 512 tuples. The performance result for each dataset is the average of the results of related queries.

TABLE III
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`
Q7	`select sum(lo_extendedprice * lo_discount) as revenue from lineorder where lo_orderdate >= 19930101 and lo_orderdate <= 19940101 and lo_discount>=1 and lo_discount<=3 and lo_quantity<25 [range 512 slide 1]`
Q8	`select sum(lo_extendedprice * lo_discount) as revenue from lineorder where lo_orderdate >= 19940101 and lo_orderdate <= 19940131 and lo_discount>=4 and lo_discount<=6 and lo_quantity>=26 and lo_quantity<=35 [range 512 slide 1]`

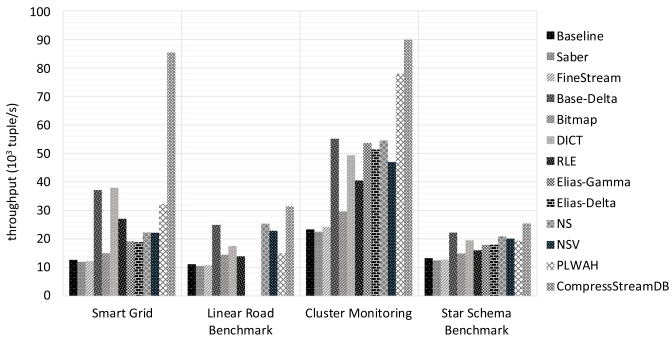

Fig. 4. Throughput of different compression methods.

B. Performance Comparison

Throughput: We delve into the throughput analysis of CompressStreamDB across the eight queries performed on the four datasets. The results are shown in Fig. 4, showcasing the performance of each dataset with distinct processing methods. Our baseline achieves similar throughput in comparison to Saber and FineStream, showcasing a performance similarity between our baseline and other state-of-the-art systems. On average, CompressStreamDB demonstrates a remarkable 3.84× throughput improvement over our baseline. Note that due to the presence of negative numbers in the linear road benchmark dataset, EG and ED cannot be applied to it. Here are our key observations.

First, on the Smart Grid dataset, CompressStreamDB showcases a substantial 6.76× increase in throughput compared to the baseline. Among the individual compression algorithms, DICT encoding stands out, delivering a 3.00× throughput improvement over the baseline. However, CompressStreamDB outperforms DICT encoding, achieving a notable 2.25× improvement in throughput. Second, concerning the linear road benchmark dataset, CompressStreamDB demonstrates a significant 2.83× increase in throughput over the baseline, while NS achieves a 2.28× improvement over the baseline. Notably, CompressStreamDB outperforms NS by 24.1% in system performance. Third, in the context of the Google Cluster Monitoring dataset, CompressStreamDB demonstrates a remarkable 3.85× increase in throughput, while BD achieves a 2.36× improvement. CompressStreamDB achieves a substantial 63.1% throughput improvement over BD. Finally, concerning the Star Schema Benchmark dataset, CompressStreamDB achieves a commendable 1.92× improvement in throughput, compared to BD's 1.68× enhancement. CompressStreamDB outperforms BD by 14.3% in system performance.

Based on the observed throughput outcomes, several significant insights emerge. First, compression can obviously improve the throughput of the stream processing system, which has been demonstrated in [66]. While BD consistently showcases robust system performance across various scenarios, the adaptive compression within CompressStreamDB consistently outperforms it. Second, the impact of compression in stream processing is notably contingent upon dataset properties. For instance, when the $AverageRunLength$ of a dataset is low, RLE fails to deliver substantial performance improvements. Thus, the algorithm selection process emerges as a pivotal factor in enhancing system performance. Third, CompressStreamDB adeptly amalgamates the strengths of diverse algorithms, showcasing remarkable adaptability across varying datasets. Its capability to consistently match or surpass the performance of individual compression algorithms underscores its versatility across datasets of varying complexities.

Latency: Fig. 5 reports the latency of different compression algorithms on the four 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. The latency of our baseline is similar compared to Saber and FineStream, demonstrating similar performance with state-of-the-art systems. On average, CompressStreamDB achieves 68.0% lower latency. Moreover, we have the following observations. First, on the Smart Grid dataset, CompressStreamDB achieves an 85.2% reduction compared to the

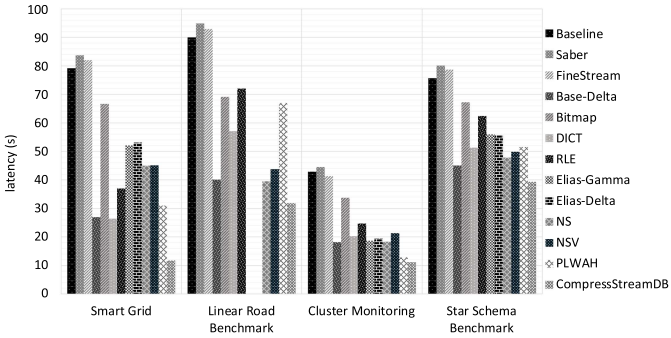

Fig. 5. Latency of different compression methods.

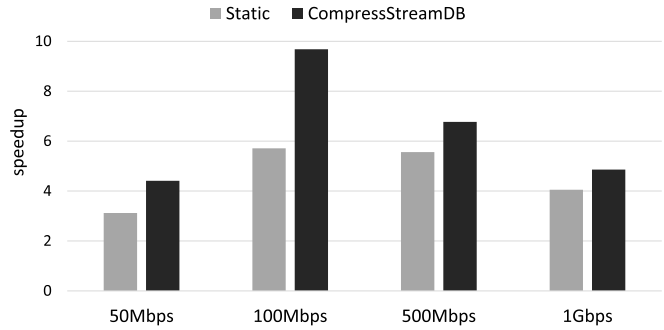

Fig. 6. Speedup with dynamic workload.

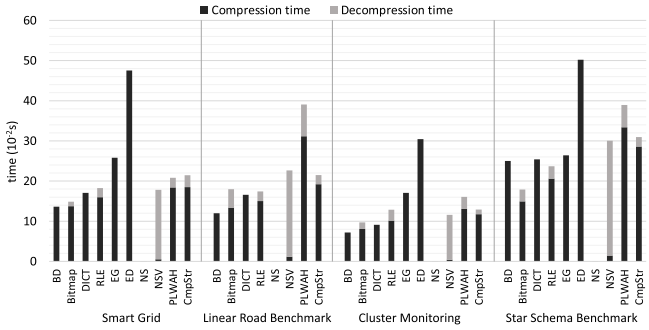

Fig. 7. Time breakdown of compression and decompression. CmpStr is short for CompressStreamDB.

uncompressed system and surpasses the performance of DICT by 55.6% in latency reduction. Second, on the linear road benchmark dataset, CompressStreamDB delivers a 64.6% decrease in latency compared to the baseline and outperforms NS by 19.4%. Third, on the Cluster Monitoring dataset, CompressStreamDB exhibits a 74.0% reduction in latency against the baseline and surpasses BD by 38.7%. Finally, on the Star Schema Benchmark dataset, CompressStreamDB demonstrates a 48.0% decrease in latency compared to the baseline and outshines BD by 12.7%. In summary, CompressStreamDB consistently delivers superior latency performance compared to individual compression algorithms.

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. Using Q1 and Q2 on Smart Grids as an illustrative example, we present speedup comparisons across various network bandwidths in Fig. 6. This trend remains consistent across other cases, showcasing similar performance behaviors. 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. The experimental results demonstrate that CompressStreamDB outperforms the static solution across varying network conditions. Particularly, in a network with 100 Mbps bandwidth, CompressStreamDB exhibits remarkable performance enhancement, achieving a 9.68× speedup over the baseline and 3.97× over the optimal static method. The performance of static method performance remains suboptimal with dynamic workloads due to its rigid data handling approach. In contrast, CompressStreamDB exhibits consistent performance as its adaptive processing method dynamically adjusts based on data characteristics, ensuring stable performance even amidst changes.

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 four lightweight decompression-required methods, RLE, bitmap, NSV, and PLWAH. We conduct experiments on three datasets to statistic the compression time and decompression time during processing. Results are shown in Fig. 7. Our observations are as follows. First, NS is a simple and fast compression technique, exhibiting the shortest sum time of compression and decompression across all four datasets. Notably, NS consistently provides commendable compression ratios across diverse scenarios, ensuring high throughput and latency performance. In contrast, EG, ED, and PLWAH, while categorized as lightweight algorithms, demonstrate relatively slower performance. Their computational intricacies and coding processes contribute to this disparity, potentially impacting their overall efficiency. Second, NSV primarily invests more time in decompression due to the translation process of byte lengths, which influences its behavior. However, it's crucial to note that transmission time constitutes the predominant portion of the total processing time. Specifically, in the case of lightweight compressions, including NSV, decompression time represents less than 1.0% of the total processing time, rendering it negligible. Third, CompressStreamDB does not rank as the most time-efficient method in compression and decompression. It takes a moderate amount of time. However, CompressStreamDB prioritizes the overall system performance rather than focusing solely on optimizing compression efficiency.

Relation between time and compression ratio: Table IV shows the relation between time and compression ratio r. On average, CompressStreamDB saves 68.7% space and 66.1% transmission time. We have the following observations. First, among

TABLE IV
RELATIONS BETWEEN TIME AND COMPRESSION

Dataset	Ratio	Baseline	BD	Bitmap	DICT	RLE	EG	ED	NS	NSV	PLWAH	CompressStreamDB
Smart Grid	trans_time ratio	1	0.341	0.843	0.327	0.453	0.638	0.647	0.559	0.537	0.502	0.178
	1/r	1	0.357	0.866	0.357	0.458	0.643	0.679	0.571	0.571	0.423	0.154
	query_time ratio	1	0.476	0.947	0.938	0.872	0.678	0.694	0.607	1.026	0.977	0.951
	1/r′	1	0.357	1	0.357	1	0.643	0.679	0.571	1	1	1
Linear Road Benchmark	trans_time ratio	1	0.462	0.803	0.630	0.806	\	\	0.431	0.487	0.746	0.353
	1/r	1	0.438	0.797	0.500	0.78	\	\	0.438	0.438	0.762	0.322
	query_time ratio	1	0.476	1.044	1.024	0.936	\	\	0.549	1.030	1.011	0.738
	1/r′	1	0.438	1	0.500	1	\	\	0.438	1	1	0.714
Cluster Monitoring	trans_time ratio	1	0.394	0.743	0.460	0.560	0.395	0.427	0.423	0.520	0.312	0.301
	1/r	1	0.438	0.781	0.438	0.555	0.438	0.438	0.438	0.438	0.513	0.341
	query_time ratio	1	0.824	0.956	0.963	0.920	0.431	0.444	0.474	0.971	0.920	0.932
	1/r′	1	0.438	1	0.438	1	0.438	0.438	0.438	1	1	1
Star Schema Benchmark	trans_time ratio	1	0.598	0.890	0.681	0.825	0.740	0.735	0.632	0.671	0.691	0.524
	1/r	1	0.650	0.885	0.483	0.744	0.767	0.750	0.683	0.684	0.774	0.437
	query_time ratio	1	0.602	0.948	0.568	1.010	0.802	0.831	0.731	0.960	0.102	0.842
	1/r′	1	0.650	1	0.483	1	0.767	0.750	0.683	1	1	0.850

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.

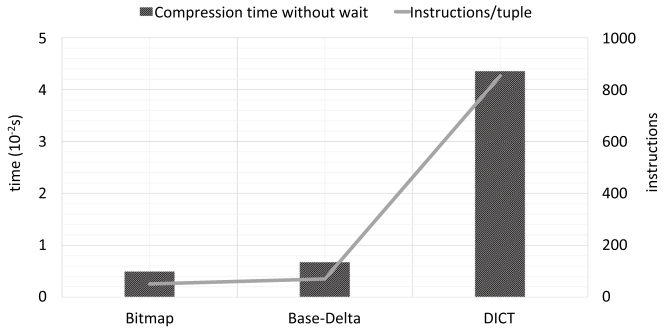

Fig. 8. Relation between compression time and instructions.

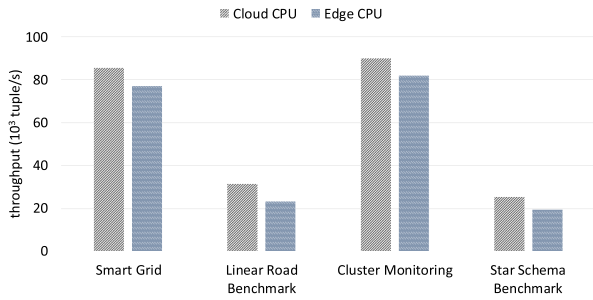

Fig. 9. Comparison of throughput on cloud and edge device.

all processing methods across datasets, CompressStreamDB demonstrates the lowest trans_time ratio. Although BD, as utilized in TerseCades, exhibits a notable advantage of 60.1% on average, CompressStreamDB still manages to surpass BD by 21.6%. Second, CompressStreamDB achieves the highest compression ratio r, or the lowest $1/r$, across all processing methods for each dataset. For instance, on the Smart Grid dataset, CompressStreamDB achieves an r of 6.49, surpassing BD's 2.80. This performance marks a 2.32× higher compression ratio r compared to 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 (4) and (5), a high compression ratio r directly implies lower transmission time. Given our use of lightweight compression methods, the time incurred during compression and decompression remains marginal. Hence, the method with the highest compression ratio can significantly enhance system performance. CompressStreamDB attains its high performance predominantly through its exceptional compression ratio r.

Executed instructions versus compression time: In our cost model, compression time and decompression time relate to the number of executed instructions. To validate this assumption, we conduct an exploration using BD, Bitmap, and DICT on the Smart Grid dataset. Fig. 8 illustrates the relationship between the number of executed instructions and the time spent in compression. Note that waiting time is not factored into this analysis. According to (2), compression time is expected to be proportional to the number of executed instructions for compute-intensive situation. The line in Fig. 8 represents the number of instructions used in compression for processing each tuple. Meanwhile, the bar denotes the compression time without considering the waiting time. Fig. 8 reveals a nearly proportional relationship between the number of instructions executed and the compression time. This aligns closely with the estimations outlined in our cost model.

D. Comparison Between Edge Device and Cloud Device

In this section, we first compare the throughput on the cloud and edge platforms, and then compare their throughput/price ratio and throughput/power ratio, which demonstrate the cost and energy benefits of the edge device.

Throughput: To demonstrate the portability of CompressStreamDB and further explore its performance, we conduct comparative experiments on the edge platform. We use Raspberry Pi 4B as the edge platform for comparison. Fig. 9 illustrates the throughput comparison between the cloud and the edge across four datasets. On average, the cloud exhibits 1.21× the throughput of the edge across these datasets. Consequently, the performance of the cloud surpasses that of the edge, resulting in superior speedup on the cloud platform.

Despite lacking performance advantages compared to cloud platforms, edge devices offer distinct benefits in terms of lower power consumption, reduced costs, and their aptness as localized servers closer to end-users. Therefore, evaluating the efficacy of edge devices should encompass considerations of price and power consumption. To assess this, we introduce two critical ratios: the throughput/price ratio, calculated as the ratio of

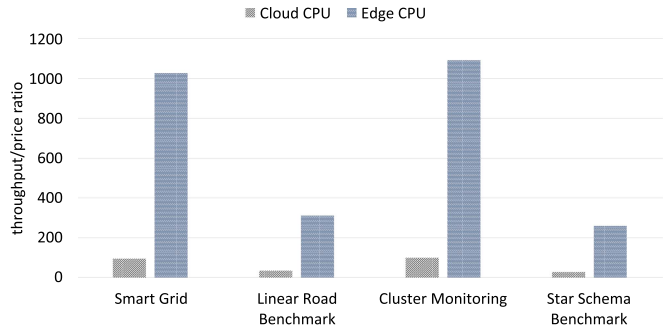

Fig. 10. Comparison of throughput/price ratio on cloud and edge device.

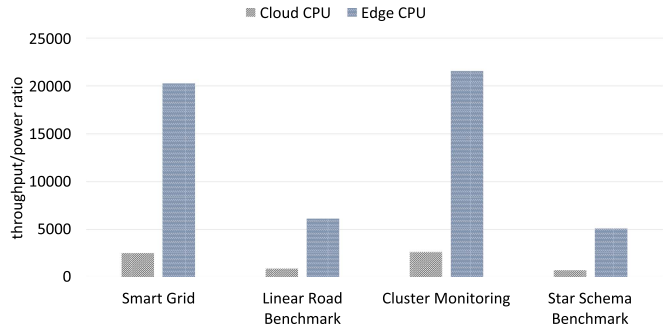

Fig. 11. Comparison of throughput/power ratio on cloud device and edge device.

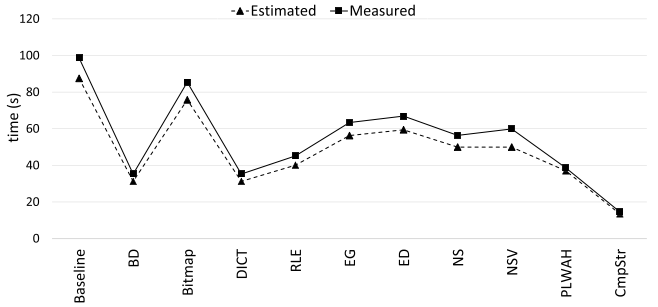

Fig. 12. Accuracy of the cost model. CmpStr is short for CompressStreamDB.

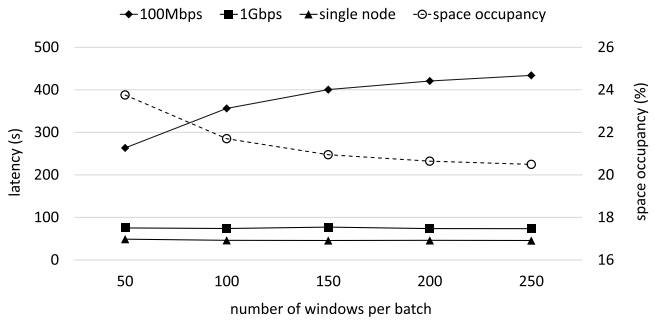

Fig. 13. Effect of batch size on latency and space usage.

throughput to price (USD), and the throughput/power ratio, denoting the ratio of throughput to power (Watt).

Throughput/price ratio: Fig. 10 shows the throughput/price ratios on the cloud server and edge device. In the Smart Grid dataset, the throughput/price ratio of the edge CPU surpasses that of the cloud CPU by 10.81×, and for the remaining three datasets, the ratios stand at 8.90×, 10.91×, and 9.18× respectively. On average, the edge device exhibits a 9.95× higher throughput/price ratio compared to the cloud, showcasing a pronounced cost advantage of utilizing edge devices. These findings affirm the cost-effectiveness inherent in the design purpose of the Raspberry Pi, known for its economical utility.

Throughput/power ratio: In Fig. 11, the throughput/power ratio for both cloud and edge platforms is presented. Notably, the average power consumption of the cloud CPU stands at 33.5 W, whereas the edge device operates at an average power of 3.8 W. In the Smart Grid dataset, the throughput/power ratio on the edge surpasses that of the cloud by 7.95×, while for the other three datasets, these ratios are 6.55×, 8.03×, and 6.75× respectively. On average, the edge platform demonstrates a 7.32× higher throughput/power ratio compared to the cloud. These findings underscore the potential for significant energy savings by leveraging edge devices compared to cloud-based operations.

E. 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 Fig. 12. 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 within CompressStreamDB achieves an accuracy of 89.2%. This level of accuracy suggests that the cost model is reliable and suitable for estimating the cost associated with stream processing integrated with compression techniques.

Batch size: As outlined in Section IV-A, the batch size is a factor influencing both latency and compression ratio. Using the Smart Grid workload as an illustration, where each window contains 1,024 tuples. We depict their interrelation in Fig. 13, considering three distinct network settings: 1 Gbps network, 100 Mbps network, and a single node without network transmission. Our observations are as follows. First, at 100 Mbps, we observe a notable rise in latency corresponding to larger batch sizes. Conversely, within the 1 Gbps network and single-node mode, batch size exhibits comparatively minimal impact on system latency. This divergence arises from the constraints imposed by limited network bandwidth, leading to data queuing before transmission. Consequently, larger batch sizes can induce system pauses. Second, as batch size increases, the space occupancy decreases. This phenomenon is attributed to the improved utilization of data redundancy with larger batch sizes. Third, the absence of an optimal batch size suggests that its determination requires specific situational analysis. Furthermore, we conducted measurements on cross-batch sliding windows, varying the window slide size within the range {1, 128, 256, 512, 1024} across different network settings. We observed nearly

identical performance levels with minimal fluctuation (less than 2%). This consistency is attributed to the ability of our batch buffer to retain critical data without imposing additional strain on network transmission.

Performance on additional benchmark: CompressStreamDB can adapt to various benchmarks. We evaluate it on the TPC-H benchmark to further demonstrate its potential. The TPC-H benchmark contains one fact table and seven dimension tables, with twenty-two standard queries. We adjust TPC-H and rewrite its queries for stream processing. We use the $Q1$, $Q2$, and $Q3$ of its standard queries for evaluation. On average, CompressStreamDB achieves 2.18× throughput compared to our baseline, while the best single compression algorithm BD achieves 1.61× throughput. In terms of latency, CompressStreamDB can reduce the latency by 53.2% compared to baseline, by 26.2% compared to BD.

Impact of parallelism: To investigate the impact of parallelism on performance, we conduct parallel throughput experiments on the cloud platform. On average, the parallel version achieves a throughput improvement of 2.10% compared to the single-core version. Although our cloud platform is equipped with an Intel Xeon Platinum 8269CY CPU and supports 52 threads, the performance improvement of parallelism is limited due to network transmission being a key factor affecting performance in our experimental network environment.

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], [15], [65], [77], [78], [79], [80]. To name a few, Koliousis et al. [14] developed Saber, a window-based hybrid stream processing for discrete CPU-GPU architectures. Zhang et al. [65] introduced BriskStream, an in-memory data stream processing system on shared-memory multi-core NUMA architectures. Zhang et al. [15] proposed a fine-grained query method of stream processing on CPU-GPU integrated architectures. Zhang et al. [81] revisited the design of data stream processing systems on multi-core processors. Scabbard [3] is a recently proposed single node optimized stream processing engine focusing on fault-tolerance aspect. Li et al. [79] proposed a framework called TRACE that allows compression on traffic monitoring streams. Pekhimenko et al. [66] 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.

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 [13], [28], [29], [30], [64], [82], [83], [84], [85], [86], [87], [88], [89], [90] has been proved to be an effective approach to increase the bandwidth utilization and resolve the memory stalls. Wang et al. [30] developed inverted list compression in memory. Deliege and Pedersen [28] optimized space and performance for bitmap compression. Wang et al. [29] conducted an experimental study between bitmap and inverted list compressions. Fang et al. [13] analyzed common compression algorithms, while Przymus and Kaczmarski [64] explored how to select an optimal compression method for time series databases. Sprintz [91] introduces a four part composite compression algorithm for time-series data. Many works [86], [87] used hardware such as GPU and FPGA to optimize data compression. As for processing directly on compressed data [18], [19], [20], [21], [92], [93], [94], this technology can provide efficient storage and retrieval of data. For example, Chen et al. [94] 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. [95], [96], [97] presented compression methods for very large databases, with aggregation operating directly on compressed datasets. Succinct [20] enables efficient queries directly on a compressed representation of data. Other works [18], [19], [98], [99] 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

The demands on stream processing systems continue to surge as the scale of streaming data expands. The growing volume presents considerable challenges in terms of both time efficiency and resource utilization within these systems. We propose CompressStreamDB, which applies compression algorithms in stream processing to improve the system performance. In our implementation, CompressStreamDB integrates nine lightweight compression algorithms, significantly enhancing performance compared to operating without any compression. Our experiments demonstrate that across four real-world datasets, CompressStreamDB achieves a substantial 3.84× increase in throughput while attaining a 68.0% reduction in latency and saving 68.7% space. Moreover, the throughput/price ratio on the edge platform outperforms that of the cloud platform by 9.95×, while the throughput/power ratio on the edge is 7.32× higher than that of the cloud.

References

[1] B. Del Monte et al., "Rhino: Efficient management of very large distributed state for stream processing engines," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2471–2486.
[2] G. Van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Aug. 2020.
[3] G. Theodorakis et al., "Scabbard: Single-node fault-tolerant stream processing," *Proc. VLDB Endowment*, vol. 15, pp. 361–374, 2021.
[4] A. R. M. Forkan et al., "AIoT-citysense: AI and IoT-driven city-scale sensing for roadside infrastructure maintenance," *Data Sci. Eng.*, pp. 1–15, 2023.
[5] State of IoT 2021, 2021. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[6] W. Wingerath et al., "Real-time stream processing for Big Data," *Inf. Technol.*, vol. 58, pp. 186–194, 2016.
[7] B. Gedik et al., "SPADE: The system S declarative stream processing engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1123–1134.
[8] M. Hirzel et al., "Stream processing languages in the Big Data era," *ACM SIGMOD Rec.*, vol. 47, 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 Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.
[11] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. IT-21, no. 2, pp. 194–203, Mar. 1975.
[12] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 5, pp. 530–536, Sep. 1978.
[13] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proc. VLDB Endowment*, vol. 3, pp. 670–680, 2010.
[14] A. Koliousis et al., "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 555–569.
[15] F. Zhang et al., "FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, Art. no. 43.
[16] A third of the internet is just a copy of itself, 2013. [Online]. Available: https://www.businessinsider.com/
[17] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. 22nd Int. Conf. Data Eng.*, 2006, pp. 49–49.
[18] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Enabling efficient random access to hierarchically-compressed data," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1069–1080.
[19] F. Zhang et al., "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proc. VLDB Endowment*, vol. 11, pp. 1522–1535, 2018.
[20] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 337–350.
[21] A. Khandelwal, "Queries on compressed data," University of California, Berkeley, 2019.
[22] F. Zhang et al., "CompressDB: Enabling efficient compressed data direct processing for various databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2022, pp. 1655–1669.
[23] Y. Zhang, F. Zhang, H. Li, S. Zhang, and X. Du, "CompressStreamDB: Fine-grained adaptive stream processing without decompression," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 408–422.
[24] P. A. Alsberg, "Space and time savings through large data base compression and dynamic restructuring," *Proc. IEEE*, vol. 63, no. 8, pp. 1114–1122, Aug. 1975.
[25] 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 *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn.*, 2012, pp. 377–388.
[26] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 671–682.
[27] M. A. Roth and S. J. Van Horn, "Database compression," *ACM SIGMOD Rec.*, vol. 22, pp. 31–39, 1993.
[28] F. Deliège and T. B. Pedersen, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps," in *Proc. 13th Int. Conf. Extending Database Technol.*, 2010, pp. 228–239.
[29] J. Wang et al., "An experimental study of bitmap compression vs. inverted list compression," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 993–1008.
[30] J. Wang et al., "MILC: Inverted list compression in memory," *Proc. VLDB Endowment*, vol. 10, pp. 853–864, 2017.
[31] R. Stephens, "A survey of stream processing," *Acta Inform.*, vol. 34, pp. 491–541, 1997.
[32] P. Córdova, "Analysis of real time stream processing systems considering latency," University of Toronto, 2015.
[33] M. H. Ali et al., "Microsoft CEP server and online behavioral targeting," *Proc. VLDB Endowment*, vol. 2, pp. 1558–1561, 2009.
[34] Apache storm, 2021. [Online]. Available: http://storm.apache.org
[35] Apache flink, 2021. [Online]. Available: http://flink.apache.org
[36] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IEEE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[37] S. Zhang et al., "Parallelizing intra-window join on multicores: An experimental study," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 2089–2101.
[38] H. Yar et al., "Towards smart home automation using IoT-enabled edge-computing paradigm," *Sensors*, vol. 21, p. 4932, 2021.
[39] Z. Lv et al., "Intelligent edge computing based on machine learning for smart city," *Future Gener. Comput. Syst.*, vol. 115, pp. 90–99, 2021.
[40] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial Internet of Things: Challenges, opportunities, and directions," *IEEE Trans. Ind. Informat.*, vol. 14, no. 11, pp. 4724–4734, Nov. 2018.
[41] J. Zhang, F.-Y. Wang, K. Wang, W. -H. Lin, X. Xu, and C. Chen, "Data-driven intelligent transportation systems: A survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 4, pp. 1624–1639, Dec. 2011.
[42] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surv. Tuts.*, vol. 19, no. 4, pp. 2322–2358, Fourth Quarter 2017.
[43] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.
[44] C.-H. Chen, M.-Y. Lin, and C.-C. Liu, "Edge computing gateway of the industrial Internet of Things using multiple collaborative microcontrollers," *IEEE Netw.*, vol. 32, no. 1, pp. 24–32, Jan./Feb. 2018.
[45] B. Hussain, Q. Du, S. Zhang, A. Imran, and M. A. Imran, "Mobile edge computing-based data-driven deep learning framework for anomaly detection," *IEEE Access*, vol. 7, pp. 137656–137667, 2019.
[46] S. Rajesh, V. Paul, V. G. Menon, S. Jacob, and P. Vinod, "Secure brain-to-brain communication with edge computing for assisting post-stroke paralyzed patients," *IEEE Internet Things J.*, vol. 7, no. 4, pp. 2531–2538, Apr. 2020.
[47] S. Aggarwal and S. Sharma, "Voice based deep learning enabled user interface design for smart home application system," in *Proc. 2nd Int. Conf. Commun. Comput. Ind. 4.0*, 2021, pp. 1–6.
[48] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proc. IEEE*, vol. 107, no. 8, pp. 1697–1716, Aug. 2019.
[49] A. Zilberman and L. Ice, "Why computer occupations are behind strong stem employment growth in the 2019–29 decade," *Computer*, vol. 4, no. 5, pp. 11–5, 2021.
[50] D. Park et al., "LiReD: A light-weight real-time fault detection system for edge computing using LSTM recurrent neural networks," *Sensors*, vol. 18, p. 2110, 2018.
[51] X. Jiang, F. R. Yu, T. Song, and V. C. M. Leung, "A survey on multi-access edge computing applied to video streaming: Some research issues and challenges," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 2, pp. 871–903, Second Quarter 2021.
[52] Z. Zhao et al., "IoT edge computing-enabled collaborative tracking system for manufacturing resources in industrial park," *Adv. Eng. Inform.*, vol. 43, 2020, Art. no. 101044.
[53] P. Ranaweera, A. D. Jurcut, and M. Liyanage, "Survey on multi-access edge computing security and privacy," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 2, pp. 1078–1124, Second Quarter 2021.
[54] H. Ziekow and Z. Jerzak, "The DEBS 2014 grand challenge," in *Proc. 8th ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 266–269.
[55] Smart home statistics, 2021. [Online]. Available: https://www.statista.com/outlook/dmo/smart-home/united-states
[56] A. Arasu et al., "Linear road: A stream data management benchmark," in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 480–491.
[57] More Google cluster data, 2011. [Online]. Available: https://ai.googleblog.com/2011/11/more-google-cluster-data.html
[58] V. Gulisano et al., "The DEBS 2017 grand challenge," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, 2017, pp. 271–273.
[59] V. Gulisano et al., "The DEBS 2018 grand challenge," in *Proc. 12th ACM Int. Conf. Distrib. Event-Based Syst.*, 2018, pp. 191–194.
[60] C. Mutschler, H. Ziekow, and Z. Jerzak, "The DEBS 2013 grand challenge," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst.*, 2013, pp. 289–294.
[61] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of GPUs and CPUs for database analytics," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1617–1632.
[62] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endowment*, vol. 4, pp. 539–550, 2011.
[63] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in *Proc. Conf. Innov. Data Syst. Res.*, 2005, pp. 225–237.

[64] 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*, Berlin, Germany: Springer, 2014.
[65] S. Zhang et al., "BriskStream: Scaling data stream processing on shared-memory multicore architectures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2019, pp. 705–722.
[66] G. Pekhimenko et al., "Tersecades: Efficient data compression in stream processing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 307–320.
[67] C. Guo et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 139–152.
[68] Raspberry pi 4 model b, 2022. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
[69] Raspberry pi dramble: Power consumption benchmarks, 2022. [Online]. Available: https://www.pidramble.com/wiki/benchmarks/power-consumption
[70] R. Castro Fernandez et al., "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 725–736.
[71] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 208–221, Second Quarter 2014.
[72] H. Funke et al., "Pipelined query processing in coprocessor environments," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2018, pp. 1603–1618.
[73] J. Li et al., "HippogriffDB: Balancing I/O and GPU bandwidth in Big Data analytics," *Proc. VLDB Endowment*, vol. 9, pp. 1647–1658, 2016.
[74] K. Wang et al., "Concurrent analytical query processing with GPUs," *Proc. VLDB Endowment*, vol. 7, pp. 1011–1022, 2014.
[75] Y. Yuan, R. Lee, and X. Zhang, "The Yin and Yang of processing data warehousing queries on GPU devices," *Proc. VLDB Endowment*, vol. 6, pp. 817–828, 2013.
[76] P. O'Neil et al., "The star schema benchmark and augmented fact table indexing," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*, 2009, pp. 237–252.
[77] X. Ren et al., "LDP-IDS: Local differential privacy for infinite data streams," 2022, *arXiv:2204.00526*.
[78] B. Zhao et al., "EIRES: Efficient integration of remote data in event stream processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 2128–2141.
[79] T. Li et al., "Trace: Real-time compression of streaming trajectories in road networks," *Proc. VLDB Endowment*, vol. 14, pp. 1175–1187, 2021.
[80] Y. Zhou, A. Salehi, and K. Aberer, "Scalable delivery of stream query result," in *Proc. VLDB Endowment*, vol. 2, pp. 49–60, 2009.
[81] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze, "Revisiting the design of data stream processing systems on multi-core processors," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 659–670.
[82] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *Proc. VLDB Endowment*, vol. 8, pp. 329–340, 2014.
[83] K. Sayood, *Introduction to Data Compression*. Burlington, MA, USA: Morgan Kaufmann, 2017.
[84] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Comput. Surv.*, vol. 19, pp. 261–296, 1987.
[85] C. Lin, "Accelerating analytic queries on compressed data," University of California, San Diego, 2018.
[86] C. Rivera et al., "Optimizing huffman decoding for error-bounded lossy compression on GPUs," 2022, *arXiv:2201.09118*.
[87] J. Tian et al., "Optimizing error-bounded lossy compression for scientific data on GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2021, pp. 283–293.
[88] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.
[89] F. Zhang et al., "Zwift: A programming framework for high performance text analytics on compressed data," in *Proc. Int. Conf. Supercomputing*, 2018, pp. 195–206.
[90] X. Huang et al., "Meaningful image encryption algorithm based on compressive sensing and integer wavelet transform," *Front. Comput. Sci.*, vol. 17, no. 3, 2023, Art. no. 173804.
[91] D. Blalock, S. Madden, and J. Guttag, "Sprintz: Time series compression for the Internet of Things," *Proc. ACM Interactive Mobile Wearable Ubiquitous Technol.*, vol. 2, 2018, Art. no. 93.
[92] Z. Pan et al., "Exploring data analytics without decompression on embedded GPU systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1553–1568, Jul. 2022.
[93] F. Zhang et al., "TADOC: Text analytics directly on compression," *VLDB J.*, vol. 30, pp. 163–188, 2021.
[94] X. Chen et al., "HBMax: Optimizing memory efficiency for parallel influence maximization on multicore architectures," 2022, *arXiv:2208.00613*.
[95] J. Li, D. Rotem, and H. K. Wong, "A new compression method with fast searching on large databases," in *Proc. 13th Int. Conf. Very Large Data Bases*, 1987, pp. 311–318.
[96] J. Li, D. Rotem, and J. Srivastava, "Aggregation algorithms for very large compressed data warehouses," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 651–662.
[97] J. Li and J. Srivastava, "Efficient aggregation algorithms for compressed data warehouses," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 3, pp. 515–529, May/Jun. 2002.
[98] W. Fan et al., "Query preserving graph compression," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 157–168.
[99] H. Maserrat and J. Pei, "Neighbor query friendly compression of social networks," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2010, pp. 533–542.

Yu Zhang received the bachelor's degree from the Department of Computer Science and Technology, Tsinghua University, in 2021. He is currently working toward the PhD degree with DEKE Lab and School of Information, Renmin University of China. His major research interests include database systems and parallel computing.

Feng Zhang received the bachelor's degree from Xidian University, in 2012, and the PhD degree in computer science from Tsinghua University, in 2017. He is a professor with DEKE Lab and School of Information, Renmin University of China. His major research interests include database systems, and parallel and distributed systems.

Hourun Li is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), in 2020. His major research interests include database systems, and parallel and distributed systems.

Shuhao Zhang received the bachelor's degree in computer engineering from Nanyang Technological University, in 2014, and the PhD degree in computer science from the National University of Singapore, in 2019. He is currently an assistant professor with Nanyang Technological University. His research interests include high performance computing, stream processing systems, and database system.

Xiaoguang Guo is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), in 2020. His major research interests include database systems and distributed systems.

Anqun Pan is the technical director with the Database R&D Department, Tencent, in China. With more than 15 years of experience, he has specialized in the research and development of distributed computing and storage systems. Currently, he is responsible for steering the research and development of the Tencent distributed database system (TDSQL).

Yuxing Chen received the PhD degree in computer science from the University of Helsinki, Finland, in 2021. He currently works as a senior research engineer with the Database R&D Department, Tencent, China. His research interests focus on database performance and evaluation, HTAP database design, and distributed system design.

Xiaoyong Du received the BS degree from Hangzhou University, Zhejiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.