Future Generation Computer Systems 152 (2024) 138–151

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

An adaptive load balancing strategy for stateful join operator in skewed data stream environments

Dawei Sun [a,*], Chunlin Zhang [a], Shang Gao [b], Rajkumar Buyya [c]

[a] School of Information Engineering, China University of Geosciences, Beijing, 100083, China
[b] School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia
[c] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Keywords:
Big data
Distributed stream processing
Gated recurrent unit
Skewed data streams
Load balancing strategy
Apache storm

ABSTRACT

As one of the most computationally intensive operations in stream processing applications, join operation can cause severe load imbalance problem when dealing with skewed data. Most of the popular solutions focused on monitoring-based dynamic balancing strategies, making it difficult to quickly adapt to the changing frequency of data stream, and sometimes failing the balancing strategies that try to address the skewed load in the cluster. To address these issues, we propose to use the prediction results of a deep reinforcement learning model and adjust the grouping strategy in advance before the frequency change of data stream. It will enable the system to quickly adapt to data stream fluctuation, while managing the resources for effective resource utilization. The following contributions are made in this paper: 1) Explore the main factors that trigger the load skewness problem in distributed stream join systems and carefully model the load balancing problem at the application level. 2) Develop a Gated Recurrent Unit Sequence to Sequence model to predict key frequency distribution of streams, and propose a dynamic grouping algorithm and a feedback-based resource elasticity scaling algorithm to solve the load imbalance problem caused by hot keys in real time. 3) Design and implement an adaptive stream join system Aj-Stream based on the prediction model and the proposed algorithm on Apache Storm. 4) Evaluate the system performance through extensive experiments on a large scale real-world dataset and multiple synthetic datasets. The experimental results demonstrate that the Aj-Stream proposed in this paper exhibits stable throughput and latency performance with both static data streams of varying skewnesses and dynamic data streams. In comparison to existing stream-connected systems, Aj-Stream demonstrated a 22.1% increase in system throughput and a 45.5% decrease in system latency when dealing with frequently fluctuating data streams.

1. Introduction

Stream computing is widely used in areas such as social networking, Internet of Things (IoT) [1] and real-time monitoring [2]. In the era of Big Data, the number of Internet users and IoT devices is increasing and the amount of data to be processed in real time are growing exponentially. In some enterprises, the volume of data can even peak at hundreds of millions messages per second. As the size of enterprise clusters grows and the volume of data increases, the challenges of stream computing become more acute. In this case, low-latency and high-throughput distributed stream processing systems (DSPSs) are required to handle dynamic and volatile data streams in real-time [3]. Popular DSPSs used in enterprises include Apache Storm [4], Apache Flink [5], Spark Streaming [6] and Apache Samza [7].

In DSPSs, the basic data unit in an input stream is called a tuple which consists of a key and a corresponding value. Join is a common operation originally used in traditional databases to join data from multiple static data tables. While in stream computing, the join operation faces unbounded data. In each stream join operation, the keys of multiple tuples are compared and the corresponding calculation result is produced according to the business logic set in code when certain conditions are satisfied. Join operations are designed to extract and match information across different streams, but the complex design of join operations in a stream processing environment often increases the latency of processing. Stream join models are primarily designed to optimize complex join operations and are widely used for processing data streams such as click streams, order streams and monitoring streams.

* Corresponding author.
E-mail addresses: sundaweicn@cugb.edu.cn (D. Sun), clzhang@cugb.edu.cn (C. Zhang), shang.gao@deakin.edu.au (S. Gao), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2023.11.002
Received 19 March 2023; Received in revised form 29 August 2023; Accepted 2 November 2023
Available online 4 November 2023

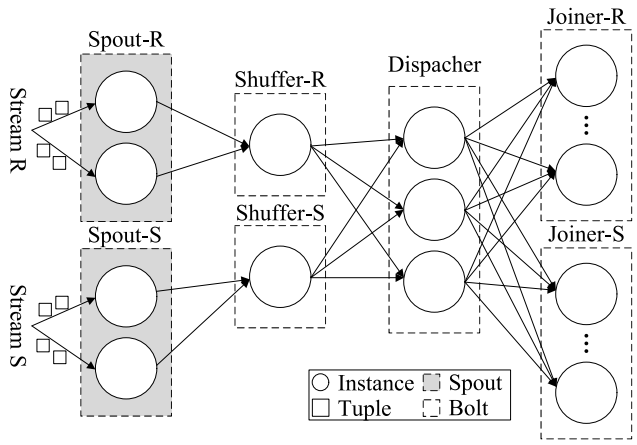

Fig. 1. The structure of topology based on the join-biclique model.

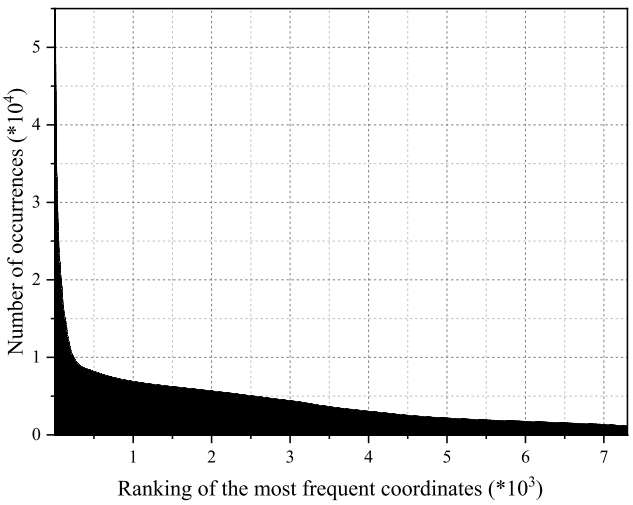

Fig. 2. Probability curve of keys in Didi Chuxing dataset.

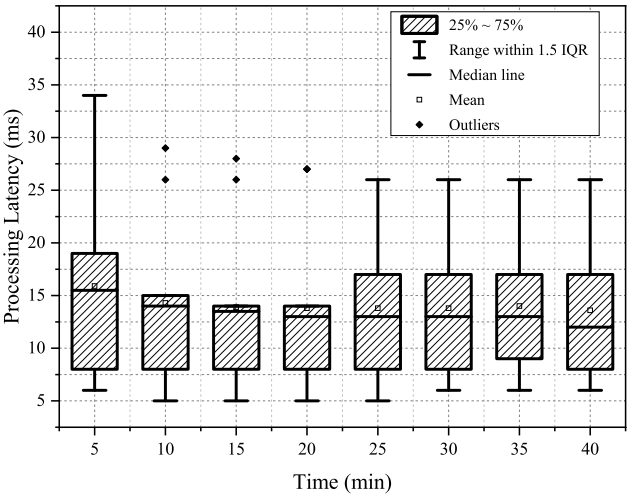

Fig. 3. Latency variation of instances in wordcount application.

Current research on stream join models can be divided into two main categories: join-matrix model [8–10] and join-biclique model [11–13]. The join-matrix model randomly divides the input streams of an application into multiple sub-streams without intersection.

Assume we have two input streams R and S; R and S are divided into m and n sub-streams, respectively. The stream joins of R and S form an $m * n$ two-dimensional matrix, with the elements of the matrix representing the sub-operations of the stream joins.

In DSPSs, join operators in the join-matrix model are usually applied to arbitrary join predicates, to guarantee the integrity of the join result. Each sub-operation takes a pair of sub-streams from R and S for join calculation. No matter how the partitioning schemes vary, the joining of R and S has to be computed $m * n$ times to complete the computation. As can be seen, it consumes considerable memory and computational overhead. Therefore, it is not conducive to performing join computation for large-scale data streams. Moreover, when dealing with skewed data, as each sub-stream is sent to at least one instance for computation, it is difficult to rebalance the load or provide elastic scaling with a lightweight solution when certain instances experience overload problems.

Similar to join-matrix, join-biclique model also divides the input streams into multiple sub-streams, but join-biclique organizes all the processing units (i.e. instances) of the stream application into a bipartite graph. We still assume that there are two input streams R and S, and they are divided into m and n sub-streams, respectively. For one side of the bipartite graph, the m sub-streams of stream R are stored on the corresponding instances of stream R. Simultaneously, they are sent to join the sub-streams with the same key stored on the instances of stream S. During the computation, the stream application does not store any copy of the data and the number of joins per sub-stream is lowered at the same time, significantly reducing memory requirements and computational costs.

In the join-biclique model, tuples with hot keys (i.e., the frequently appearing part of the key set) in stream S can be easily matched to those with the same key in stream R by comparison, while the cold keys (i.e., the seldom appearing part of the key set) needs more comparisons. Therefore, the processing of hot keys becomes a potential bottleneck for DSPSs. When the keys in data stream are evenly distributed, Key Grouping can perfectly solve this problem, because the same keys in stream R and stream S are sent to the same instance by the hash function and the workload of each instance is approximately the same. The topology based on the join-biclique model is used in our experiment. Its topology structure is shown in Fig. 1.

However, it is common that real-world data streams are skewed. We count 300 million taxi tracks in a large-scale dataset provided by Didi Chuxing [14] and generate a probability curve of the keys. As shown in Fig. 2, the data exhibit a clear long-tailed distribution. In this case, if keys are grouped by fields, as shown in Fig. 3, the workload of one instance may be much greater than those of other instances or the queue length may reach the upper limit. It will trigger the back pressure mechanism and cause the upstream instances to halt, leading to lower system throughput and higher processing latency. In this paper we argue that the key frequency distribution is one critical cause of overload in downstream instances, which are commonly referred to as hot keys or “heavy hitters”.

In order to address the problems caused by the skewed data, we can use key frequency to divide the data stream into sub-streams, but how to efficiently and dynamically calculate the frequency of keys is still a problem for online grouping strategies in data stream processing. To collect the statistics of key frequency online, real-time monitoring is required. Due to the limitation of memory capacity and storage time, it is impossible to store the frequency of all keys in the system. Therefore, it is necessary to estimate the key frequency. Real-time data stream has the characteristics of fast transmission speed and unlimited data, making the key frequency estimation challenging. There are two requirements to satisfy: firstly, the estimation needs to be accurate enough; Secondly, the memory usage for data processing must be sufficiently small. The most common method is using sketch [15,16] for stream frequency estimation.

With the development of artificial intelligence (AI) algorithms, more and more works [17–19] are beginning to use AI algorithms to make online adjustments to grouping strategies. After training the AI algorithms have a faster response time and are able to process and analyze changing data in the environment in real time, thus dynamically adjusting the grouping algorithm to adapt to changes in the environment. However, AI algorithms often require a large amount of historical data for training, otherwise their analysis results may be inaccurate, resulting in poor grouping strategy adjustment. In addition, due to the exponential growth in the calculation of all possible scenarios where a batch of tuples could be allocated to a compute node, resulting in a substantial decision space, the application of AI algorithms at the decision-making stage becomes challenging.

Based on the problem analysis, we summarize the following challenges to load balancing of DSPSs with dynamic stewed data: (1) As the stream content keeps changing, the previously generated distribution strategies might no longer be applicable for the next time window, and may even aggravate the overload of some compute nodes. (2) The training of a AI model takes time, so it is impossible to collect large-scale data for training at runtime. But in scenarios involving streams with changes in their underlying key distribution or concept drift, relying solely on a pre-trained model can result in outdated information and, consequently, lead to poor suboptimal balancing decisions. (3) As the application scenarios and objectives of AI algorithms vary at different stages of data processing, the use of AI algorithms requires careful consideration and selection of the appropriate stage in the data processing pipeline, taking into account specific business requirements and the nature of data processing processes.

To tackle these challenges, this paper proposes and implements an adaptive load balancing strategy for stream joining system. The strategy involves several key components: analyzing the relationship between key frequencies in data and processing instance loads, identifying load imbalances caused by differences in high and low-frequency key-generated loads; establishing a stream topology model and quantifying load models for parallel instances based on topology; introducing a prediction-based grouping strategy for dynamic load balancing adjustments; utilizing a GRU sequence-to-sequence model with attention mechanism to predict key frequency distribution in future data stream time windows; and suggesting a feedback-based resource scaling strategy to counter resource-induced load imbalance. The method, Aj-Stream, is developed and integrated into Apache Storm. Its effectiveness is demonstrated through experiments on the DiDi Chuxing dataset, showcasing its capability in mitigating load imbalance arising from high-frequency keys in stream joining.

The remainder of this paper is organized as follows. In Section 2, we present a model for the load balancing problem caused by dynamic skewed data. Section 3 describes the architecture and design of our Aj-Stream system. Section 4 evaluates the performance of Aj-Stream through extensive experiments. Section 5 reviews related work on stream grouping strategies and elastic scaling strategies. Finally, in Section 6 we summarize our work and discuss the future work.

2. Problem statement

In this section, we describe the research problem and formulate the stream topology model, load quantification model and key distribution prediction model for stream computing environments.

2.1. Problem description

Load variance between processing instances is one of the key factors influencing the system performance. Most stream applications have to face load imbalance problem due to data skewness. A small square in Fig. 4 represents a Tuple in the data stream, where different patterns represent different keys. The dashed box indicates the set of downstream instances that are logically associated with the upstream

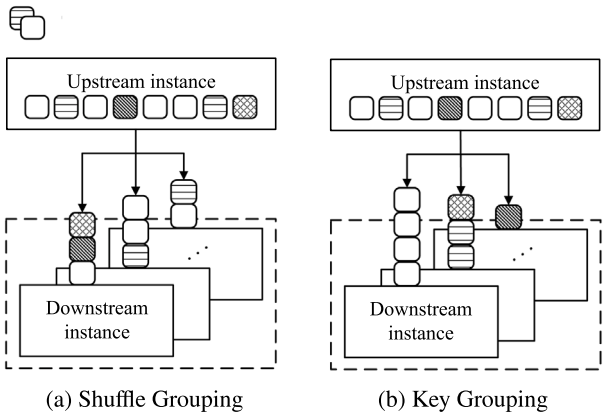

Fig. 4. Stream application with different grouping strategies.

instances in terms of code processing. Data flows through the upstream instance to the downstream instance. As shown in Fig. 4(a), for a stream application, if Shuffle Grouping strategy is used, the upstream instance will send tuples to downstream instances in a round-robin manner. Although load balancing between each instance can be guaranteed, we seldom use this strategy for tuple allocation in stateful operations because an aggregation instance has to be set up for result aggregation.

In Fig. 4(b), Key Grouping strategy groups tuples with the same key to the same downstream instance. It eliminates the need for aggregation, but if the data stream is severely skewed, some instances may have a long queue (i.e. overloaded). In this case, the underloaded instances will stop working until the overloaded instances' states are eased. This may cause back pressure or tuple missing problems.

2.2. Stream topology model

A stream topology in DSPS can be represented by a directed acyclic graph (DAG) [20], defined as $G = (V, E)$. A topology defines all the necessary components for running a stream application, mainly including sources, processing operators and the relationship between sources and processing operators. We define the set of n vertices in a DAG as $V = \{v_1, v_2, v_3, \ldots, v_n\}$. Each vertex of the DAG consists of several task instances, which can be represented as $I(v_i) = \{i_1, i_2, \ldots, i_m\}$. The code of the stream application is actually executed by these instances. E is a set of directed edges, each element of which corresponds to a pair of vertices in V and can be expressed as $E = \{e_{ij} | i, j \in n\}$. Edge e_{ij} denotes the direction of data stream from vertex v_i to vertex v_j in the DAG. Its value indicates the associated weight (e.g. computational or communication cost).

2.3. Load quantification model

In DSPSs, it is necessary to distinguish between overloaded and underloaded instances by quantifying the workload of each instance, so as to facilitate the subsequent load balancing procedure. Tasks are the executor of application code and each task carries an instance of sources or processing operators. The group of each upstream instance can only allocate tuples to different downstream instances. This tuple-instance allocation scheme has a direct impact on the system performance. Stream grouping algorithms can be used to globally load balance the downstream operators.

For a stream join system with join-biclique model, the process to consume tuples is mainly as follows: tuples of stream S are routed to I_{S_j} (i.e., the jth instance of stream S) for storage; meanwhile, they are sent to all the instances of R to compare with the tuples stored in I_{R_i} (i.e., the ith instance of stream R). Similarly, the process in another group of joining instances is the same, so in this paper we only

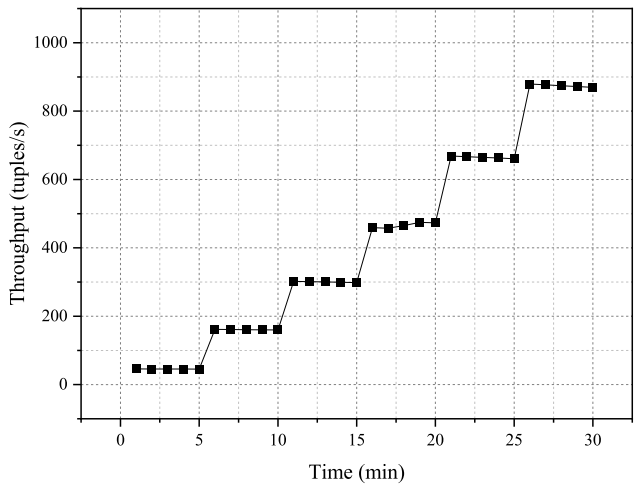

Fig. 5. The average throughput variation of a join instance as resources increase over time.

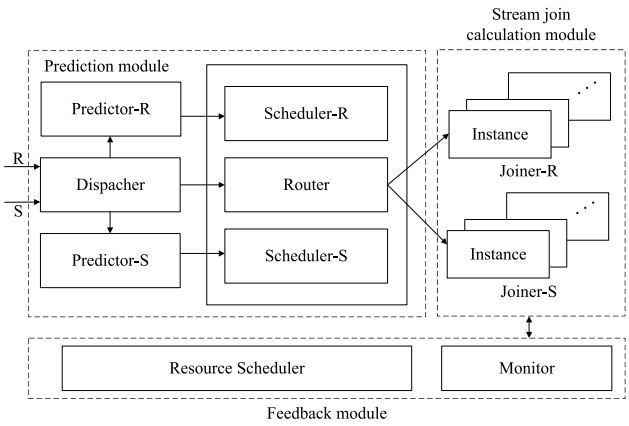

Fig. 6. Structure of Aj-Stream.

discuss one group of join instances. From the above process, it can be learned that the workload L_{R_i} on instance I_{R_i} can be referred to as the product of the total number of tuples (denoted as R_i) stored on I_{R_i} and the tuple queue length of stream S waiting for processing on I_{R_i} (denoted as ΔS_i). Note that since each comparison incurs a different overhead, for ease of modeling we assume that each tuple comparison takes the same amount of time, i.e., the average of the time required for all comparisons is used as the time for each comparison. Our revised explanation explicitly states that the Load quantification model assumes a representative average for the comparison overheads, acknowledging that in practical scenarios, variations in comparison times might exist due to the complexity of if statements and other factors.

$$L_{R_i} = R_i * \Delta S_i, i \in (1, 2, \dots, n) \tag{1}$$

The system performance often depends on the number of parallel instances with the heaviest load, thus we define the degree of load imbalance LI as follows,

$$LI = \frac{max\left(L_{R_i}\right) - avg\left(L_{R_i}\right)}{avg\left(L_{R_i}\right)}, i \in (1, 2, \dots, n) \tag{2}$$

For stream R, we use $max(L_{R_i})$ to denote the heaviest loaded instance and $avg(L_{R_i}) = \frac{1}{n}\sum_{i=1}^{n} L_{R_i}$ to denote the average workload of all instances. Since $max(L_{R_i})$ can be equal to or greater than $avg(L_{R_i})$, and LI can be equal to or greater than zero, it can be concluded that the larger LI is, the more severe the load imbalance is.

Let the instantaneous load generated by key k in instance R_i at time t_0 be $L_i^k(t_0)$, then $f(t) = \int L_i^k(t)dt$ is the total load over time as a function of time t. Assuming that k is all sent to a join instance, if we treat the number of tuples R_i^k stored in the instance with key k as a constant, we can obtain the following equation by Eq. (1),

$$\begin{aligned} \int L_i^k(t)dt &= R_i^k * \Delta S_i^k \\ &= R_i^k \int v_{S_i}^k(t)dt \end{aligned} \tag{3}$$

where we define the processing rate of a tuple in stream S as a function $v_{S_i}(t)$ with respect to t. Deriving both sides of Eq. (3) yields the following expression for the instantaneous load L_t^k,

$$L_i^k(t) = R_i^k * v_{S_i}^k(t) \tag{4}$$

It can be seen that the instantaneous workload generated by key k is related to the total number of its tuples and the processing rate of the instance. Similarly we can induce that the instantaneous workload of instance R_i is related to its memory capacity and CPU resources. Fig. 5 records the average throughput variation of a join instance per minute. One of the join instances was monitored for 30 min after the topology was stable, and the CPU and memory resources of this join instance were increased in equal amounts every 5 min. The data shows that the more memory and CPU resources an instance has, the higher its throughput will be.

In fact, there are many factors affecting the load limit of an instance. E.g., besides of memory and CPU, there are also bandwidth and transfer latency, etc. Moreover, the instance with the lowest load ceiling can easily become a performance bottleneck for the whole system. For memory and CPU resources, we can solve this by using elastic scaling and allocating more resources to the instance. Our elastic scaling strategy prioritizes vertical scaling, i.e., increasing the resources allocated to workers. For transfer latency, adjacent upstream and downstream instances can be placed on the same worker through the task scheduler at the time of application deployment, avoiding the situation where two instances that are far apart communicate. For bandwidth resources, we did not intend to scale them in our design, as bandwidth resources will not become a bottleneck for system throughput in most cases. In our experiments, we provide sufficient bandwidth and ensure that applications do not over consume the bandwidth of the cluster.

3. Aj-stream overview

In this section, we first give an overview of Aj-Stream, then present the algorithms for key frequency distribution prediction, key selection and migration. Finally, the feedback-based migration and scaling strategies are discussed.

3.1. System architecture

A fast and efficient stream join system is expected to support high throughput, low latency and memory cost, and scalability while ensuring the completeness of the results. We design an adaptive stream joining system, Aj-Stream, based on the join-biclique model. It has a key frequency distribution prediction model and a feedback module, and has been implemented on Apache Storm.

As shown in Fig. 6, Aj-Stream consists of a prediction module, a feedback module and a stream join calculation module. The prediction module is composed of a predictor and a task scheduler, where the predictor takes on the task of predicting the frequency distribution of keys and sends the prediction results to the task scheduler. The task scheduler divides each key k into N_k sub-streams according to the prediction-based grouping algorithm. As fluctuations of data stream can be seen as a moving average model with a persistent decline after a

peak is detected, the allocation of tuples alone cannot solve the load skewness problem.

Therefore, we design a feedback module to balance the system load by scaling the operators elastically based on the feedback information. In the feedback module, we set up two monitors monitoring the input stream rate and performance information of Joiner-R and Joiner-S operators respectively, and feed this information into the resource scheduler. The resource scheduler adjusts the resources of Joiner-R and Joiner-S based on the feedback information and the elastic scaling algorithm. Note that the scaling of the system can be divided into two scalability options: horizontal and vertical. Horizontal implies an increase in the number of workers, and vertical implies an increase in the resources allocated to workers [21]. Our elastic scaling strategy prioritizes vertical scaling, aiming to increase the resources allocated to workers. While both horizontal and vertical scaling approaches are supported, we emphasize the advantages of vertical scaling due to the significant migration and aggregation costs associated with frequent adjustments to operator parallelism.

Aj-Stream is built on top of the Apache Storm architecture. Storm consists of a Nimbus subsystem, a Zookeeper subsystem and a Supervisor subsystem. As the master node in the cluster, Nimbus creates stream applications based on the logic of code and deploys each stream application to an appropriate Supervisor subsystem with the help of scheduling algorithm. The configurations in storm.yaml (the configuration file of Storm platform) can be customized to specify the employed scheduling strategy. As the components in a Storm cluster do not store the state information, we store the state information in Zookeeper to ensure the stability of the cluster. Zookeeper is also responsible for the communication between Nimbus subsystems and Supervisor subsystems in the cluster for cluster coordination. The Supervisor node can start and stop worker processes as needed. In this way, the operators assigned to this node are managed.

3.2. Prediction model

To support a better load balance on dynamic data, predicting the data stream distribution for the next time window in advance can be an effective method. We can adjust the grouping strategy for the next time window based on the prediction to enable a faster response to data fluctuations.

In our study, we primarily utilize time-based windows to partition the incoming data stream for predictive purposes. This approach involves dividing the continuous data stream into discrete time intervals, allowing us to analyze and predict trends within specific temporal contexts. By opting for time-based windows, we aim to capture the temporal dependencies inherent in the data, which is crucial for accurate predictions in dynamic environments.

Regarding the specific behavior of our windows, we adopt a sliding window strategy. This strategy entails overlapping consecutive windows, enabling us to continuously process and predict based on the most recent data while retaining historical context. This sliding window mechanism offers a balance between capturing short-term patterns and considering longer-term trends, enhancing the robustness of our predictive model.

The determination of window size, advance and offset is a multifaceted decision-making process. Based on the characteristics of the Aj-Stream and Didi datasets, the size of the time window used in this paper is 10 min, and also, since the prediction is a real-time prediction, we set the offset to zero and the advance to 10 min.

The serial characteristic of data streams and the strong autocorrelation of the vast majority of real-world datasets allow us to analyze and model stream data using time series processing models such as Autoregressive Integrated Moving Average (ARIMA) [22]. In this paper, we use a Gated Recurrent Unit (GRU)-based seq2seq model [23] to model and predict data stream for the following reasons: (1) Recurrent Neural Networks (RNN) [24] is an algorithm that extends the ARIMA model and can easily identify relevant exogenous features. (2) GRU is an expansion of RNN. It uses gate structure regulation to overcome the effects of short-term memory. (3) seq2seq has no input/output constraints and is more flexible than simply RNN prediction. Note that in seq2seq, each decoding of the decoder is used as input for the next decoding, and errors at each step of the training process will accumulate, making the training result error increasingly large. To solve this problem, we have added an attention mechanism to the model. In this paper, the encoder and decoder use the same structure (i.e. GRU), and the update can be represented by Eq. (5). In our prediction model, the encoding vector is first computed by the encoder and then injected into the decoder for the prediction result.

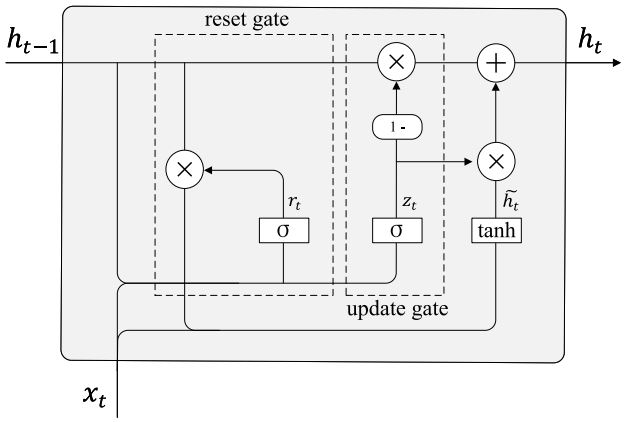

Fig. 7. Relationship between gates in GRU.

$$h_t = (1 - z_t) * h_{t-1} + z_t * \hat{h}_t \tag{5}$$

Given input streams $R = (r_1, r_2, r_3, \ldots, r_T)$ in T historical time windows, the data streams in each window are counted and generate a key frequency distribution. The set of key frequency distributions for the full time window can then be expressed as $\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \ldots, \mathcal{F}_T)$ for each time window. Let the set of keys be $K = (k_1, k_2, k_3, \ldots, k_n)$, then the key frequency distribution for the tth time window can be expressed as $\mathcal{F}_t = (f_{k_1}, f_{k_2}, f_{k_3}, \ldots, f_{k_n}), t \in [1, T]$. The f_{k_i} represents the frequency of occurrence of ith key. We use $\mathcal{F}_t$ as the input data to the prediction model, and the update process on the encoder side can be represented by the following Eqs. (6)–(8), where h_t is the hidden state transferred from the previous GRU node, r_t is the reset gate, z_t is the update gate, and $\hat{h}_t$ is the hidden state after the reset. The relationship between gates in GRU is shown in Fig. 7.

The prediction process of this prediction model is that the state vector is first computed by the encoder and then injected into the decoder to obtain the prediction result.

The update process at the encoder side is simply the use of reset gates and update gates to control the output hidden state or candidate hidden state, using the key frequency distribution of the tth time window as the input data in the update process.

$$r_t = \sigma\left(W_r\left[h_{t-1}, x_t\right]\right) \tag{6}$$

$$z_t = \sigma\left(W_z\left[h_{t-1}, x_t\right]\right) \tag{7}$$

$$\hat{h}_t = tanh\left(W_{\hat{h}}\left[r_t * h_{t-1}, x_t\right]\right) \tag{8}$$

The decoder utilizes the attention mechanism to compute a context vector a_t, representing a weighted average of the input sequence, based on the current state c_t and all the hidden states of the encoder. The computation of a_t is shown in Eqs. (9)–(11), where α_{ti} is the attention weight and e is the attention score.

$$a_t = \sum_{i=1}^{T} \alpha_{ti} h_i. \tag{9}$$

$$\alpha_{ti} = \frac{exp(e_{ti})}{\sum_{k=1}^{T} exp(e_{tk})}. \tag{10}$$

$$e_{ti} = v^T tanh(w_h h_i + w_c c_t). \tag{11}$$

The decoder stitches together the state vector c_t and the context vector a_t to get the prediction result for the current time window through the linear and softmax layers. Finally, the prediction result of the current time window is taken as the output and used as the input of the next time window y_{t+1}.

3.3. Candidate key selection

In streaming applications, a heavy hitter can cause a skewed workload for downstream instances, so effectively dividing out the headers of data is essential to a dynamic grouping strategy. In the solution explored in this paper, we first identify the head of data based on the prediction results and add them to the candidate key set $\mathcal{H}$, then we use a different scheme for them than for the tails when assigning them to downstream instances. An instance load threshold is set to control the instance load and determine the number of these heavy hitters (i.e. the size of H, $|\mathcal{H}|$). We divide the grouping strategy into two parts: candidate key selection and prediction-based grouping. The first part is described in Algorithm 1.

Algorithm 1 Candidate key selection algorithm

Input: candidate key set $\mathcal{H}$, prediction result $\mathcal{F}$, routing table A, workload threshold θ_L, memory size of the instance $M_{instance}$, size of each tuple m_{tuple}
Output: updated candidate key set $\mathcal{H}$

```
Get the candidate key set for the last time window H
Get the predicted results for next time window F < key, ratio >
Get the current routing table A
Sort F in descending order of ratio
Calculate the size of candidate key set |H| by Eq. (12)
if the routing table A is empty then
    for each key k in the predicted results F do
        Add the top |H| keys of F to H
    end for
else if the size of routing table A is less than |H| then
    for each key k in the predicted results F do
        if k does not exist in the routing table A then
            Add k to candidate key set H
        end if
    end for
else
    return candidate key set H
end if
return candidate key set H
```

(a) Based on the definition of workload in Section 2.3, we define the upper load limit for each instance as a threshold $\theta_L \in (0, 1)$. Its default value is 0.8 in this paper. User can modify this threshold. Based on the instance load threshold θ_L, the size of tuple head $|\mathcal{H}|$ is calculated according to Eq. (12), where $M_{instance}$ is the memory size of the instance being set in the configuration file, and m_{tuple} denotes the size of each tuple. The tuple size can be obtained from the pre-processor in this paper.

$$|\mathcal{H}| = \frac{\theta_L M_{instance}}{m_{tuple}} \tag{12}$$

(b) Obtain the predicted frequency distribution $\mathcal{F}_{t+1}$ for the next time window after the start of t-th time window. When the routing table is empty, the top $|\mathcal{H}|$ keys in the predicted results are selected and added to the candidate key set $\mathcal{H}$. When the routing table is not empty and the number of entries in the routing table is less than $|\mathcal{H}|$, the predicted results will be checked in turn, and the keys that do not exist in the routing table will be selected as candidate keys and added to the set of candidate keys. There is also a case where the number of routing table entries is greater than or equal to $|\mathcal{H}|$, in which case the algorithm will skip the candidate key update process for the current time window and wait for the next time window to receive new results of prediction.

3.4. Prediction based grouping

Inevitably, in the allocation process, some keys may have a predicted volume greater than the threshold we set. This case requires subsequent adjustment. Our proposal is to split these heavy hitters using the idea of Partial Key grouping (PKG) [25].

After the candidate keys have been selected, the grouping scheduler needs to split and add all the keys to the candidate key set in the routing table. In the prediction-based grouping algorithm as described in Algorithm 2, we split the number of candidate keys and use the first-fit method to assign each key to the appropriate destination. When the system is running, for the keys that are not in the routing table, the scheduler uses a hash function to assign their tuples. For each key k in the routing table, the scheduler distributes its tuples to the destinations using a round-robin method. The main steps of the prediction-based grouping algorithm are described as follows.

(a) Create a routing table. In Eq. (13), $|I_R|$ represents the number of joiner-R, $\mathcal{P}_{t+1}(k)$ is the predicted value of the frequency of key k in the prediction result. First, the number of splits N_k for each candidate key k is determined according to computing the integer part of the product of $|I_R|$ and $\mathcal{P}_{t+1}(k)$. The destination instance of each split is determined using the first-fit method and added to the routing table, while k is removed from the set of candidate keys. Note that in the routing table, each entry consists of a key and multiple destinations.

$$N_k = \lfloor |I_R| * \mathcal{F}_{t+1}(k) \rfloor \tag{13}$$

(b) Update the routing table. As the frequency prediction result of each key may change, we need to update the number of splits of heavy hitters. Use Eq. (13) to calculate the new splits number of the key k and record it as N'_k. The difference ΔN_k between the numbers of splits before and after the update is produced by Eq. (14). Thereafter, we can adjust the entries of heavy hitters in the routing table according to the difference ΔN_k. For example, if $\Delta N_k = -1$, we only need to delete one destination from the routing table for key k. If $\Delta N_k = 2$, we use the first-fit method to select two suitable destinations for the split of key k and add them to the routing table. If $\Delta N_k = 0$, no need to update the entries for key k in the routing table.

$$\Delta N_k = N'_k - N_k \tag{14}$$

When a decision is made to reassign the ownership of a particular key to a different processing unit due to load balancing or skewness mitigation, the system identifies this change. The state associated with the key is captured and serialized. This includes all relevant data, metadata, and any intermediate results associated with the key's processing. The serialized state is securely transmitted from the previous key owner to the new key owner. This communication ensures that the state is transferred intact and without corruption.

The transfer process is designed to be atomic and consistent. This means that either the entire state is successfully transferred and integrated into the new owner's processing context, or the transfer is not executed at all. This prevents data inconsistencies or partial states. Once the state is received by the new key owner, a verification and validation process is performed to ensure the integrity and accuracy of the transferred state. This involves checksums, validation checks, and reconciliation with any updates that might have occurred during the transfer. In case of any issues during the transfer or validation process, a rollback mechanism is in place to revert to the previous state and ensure that the system remains in a consistent state.

Algorithm 2 Prediction-based grouping algorithm

Input: candidate key set $\mathcal{H}$, prediction result $\mathcal{F}$, routing table A
Output: routing table A

1: Get I_R, the instance set in Stream R
2: Get $|I_R|$, the number of instances in I_R
3: **for** key k in routing table A **do**
4: Get N_k, the size of $A(k).Dest$
5: Update the number of splits for k to N'_k by Eq. (13)
6: $\Delta N_k \leftarrow N'_k - N_k$
7: **if** $\Delta N_k > 0$ **then**
8: **for** each instance d in instance set I_R and $\Delta N_k \neq 0$ **do**
9: **if** d does not exist in $A(k).Dest$ and a split of k is suitable for instance d **then**
10: Add d into $A(k).Dest$
11: $\Delta N_k \leftarrow \Delta N_k - 1$
12: **end if**
13: **end for**
14: **else if** $\Delta N_k < 0$ **then**
15: Delete $|\Delta N_k|$ destinations from $A(k).Dest$
16: **else**
17: Break
18: **end if**
19: **if** $A(k).Dest$ is empty **then**
20: Delete k and $Dest$ from routing table A
21: **end if**
22: **end for**
23: **for** key k in candidate key set $\mathcal{H}$ **do**
24: Initialize an empty instance set $Dest$
25: Add k and $Dest$ into routing table A
26: Delete k from candidate key set $\mathcal{H}$
27: Calculate the number of splits N_k by Eq. (13)
28: $n \leftarrow N_k$
29: **for** each instance d in instance set I_R and $n \neq 0$ **do**
30: **if** d does not exist in $A(k).Dest$ and a split of k is suitable for instance d **then**
31: Add instance d into $A(k).Dest$
32: $n \leftarrow n - 1$
33: **end if**
34: **end for**
35: **end for**
36: **return** the routing table A

3.5. Feedback based scaling

Considerable prior studies have investigated dynamic grouping strategies to improve the adaptation to skewed stream data. However, dynamic grouping strategies often incur heavy migration costs. A dynamic grouping strategy alone is not a good solution to load skewness under dynamic data. To reduce the migration costs and improve the system adaptability, we use an elastic scaling method in the feedback scheduler, in addition to the periodic adjustment of the stream grouping strategy approach. In the feedback scheduler, we set up two monitoring components to collect performance and load information of the streams and feed it into the elastic scaling component. A heuristic algorithm is proposed to compute a resource allocation scheme in conjunction with a stream grouping strategy to solve the dynamic load skewing problem. The basic idea is working out an approximate optimal resource allocation scheme based on the feedback received.

We identify the lag instances in the stream join application based on the processing latency information $Delay_R = \{d_{r1}, d_{r2}, d_{r3}, \ldots, d_{rN}\}$ in the feedback message. The processing latency represents the time taken by the instance from the time it receives a unique tuple to the time the tuple is processed, which gives a clear indication of the lag of the instance. When the average processing latency d_{ri} of I_{R_i} satisfies Eq. (15), I_{R_i} can be considered as a lagging instance, otherwise it is a normal instance. Then we get the resource allocation for each instance $RAt_R = \{ra_{r1}, ra_{r2}, ra_{r3}, \ldots, ra_{rN}\}$ and calculate the available resources by Eq. (16).

In this paper we use (c, m) to denote resources, where $c \in (0, 1)$ is defined as the percentage of allocated CPU resources to the total CPU, and m as the amount of memory resources in megabytes.

$$d_{ri} \geq \frac{\theta}{N} \sum_{i=1}^{N} d_{ri} \tag{15}$$

$$\mathcal{R}_{available} = \mathcal{R}_{total} - \sum_{i=1}^{N} ra_{ri} \tag{16}$$

The data input rate dir_{ri} for each instance can be obtained from the feedback information. The resource allocation plan $RAp_R = \{\Delta ra_{r1}, \Delta ra_{r2}, \Delta ra_{r3}, \ldots, \Delta ra_{rN}\}$ is determined by Eq. (17), where (c_0, m_0) denotes the preset resource allocation unit. To avoid a too-fast resource adjustment, we set a threshold λ that will be used as a coefficient in the calculation when the coefficient of (c_0, m_0) exceeds the threshold. We set an appropriate value for λ in our experiments. The elastic scaling algorithm will release a portion of instance's resources in each round of resource allocation to avoid resource waste. Specifically, when the data input rate of an instance is less than the average, the calculated resource allocation plan is negative and the corresponding instance will release the corresponding resources when it receives a resource scaling command. This feedback based scaling algorithm is described in Algorithm 3.

In this algorithm, the input data includes all the resources information of the cluster, allocation plan, feedback and DAG graph information. The output data is the resource allocation plan. In line 1, a collection of resource allocation plans is first initialized. In lines 2–4, the algorithm identifies the instances with high latency in the DAG graph, which are saved in the variable $\mathcal{G}$, and calculates the current available resources, which are saved in $\mathcal{R}_{available}$. In lines 5–11, the computation of the resource allocation plan is performed without $\mathcal{G}$ being empty, It first obtains the input stream rate, then calculates the resource allocation Δra_{ri} for each instance in $\mathcal{G}$ based on the input stream rate, and finally adds Δra_{ri} to the resource allocation plan.

$$\Delta ra_{ri} = \begin{cases} \lambda(c_0, m_0) & \lambda < \lambda^* \\ \lambda^*(c_0, m_0) & \lambda \geq \lambda^* \\ , \end{cases} \tag{17}$$

where $\lambda = \dfrac{N dir_{ri} - \sum_{i=1}^{N} dir_{ri}}{\sum_{i=1}^{N} dir_{ri}}$

4. Performance evaluation

We implement Aj-Stream on Storm platform using Kafka, a distributed messaging system [26], as the message queue for stream applications. Extensive experiments are conducted on a large-scale real-world dataset. In this section, we first discuss the experimental environment settings, and then analyze the prediction accuracy and system performance.

To implement the prediction module, we need a data stream preprocessing component and a prediction component external to the stream join system. Data stream from the source goes through Apache Kafka to the stream join system, then is sent to the data preprocessing part for analysis. The data stream preprocessing component examines data distribution within the current time window and forwards this information to the prediction component, which subsequently stores the predictions in the database. In the feedback module, a monitoring component tracks and stores system info in the database, oversees resource monitoring of joining instances, processes data for resource

Algorithm 3 Feedback Based Scaling Algorithm

Input: total resource $\mathcal{R}_{total}$, allocation scheme RAt_R, feedback message Fbm , DAG.
Output: allocation plan RAp_R.

```
RAp_R ← Initialize an empty set
if R_total is not empty then
    G ← Identify the lagging instance in DAG by Eq. (15)
    R_available ← Calculate the available resource by Eq. (16)
    if G is not empty then
        for i in 1 to N do
            dir_ri ← data input rate in I_Ri
            Δra_ri ← Calculate the resource allocation plan by Eq. (17)
            RAp_R.add(Δra_ri)
        end for
    else
        return RAp_R
    end if
    return RAp_R
end if
return RAp_R
```

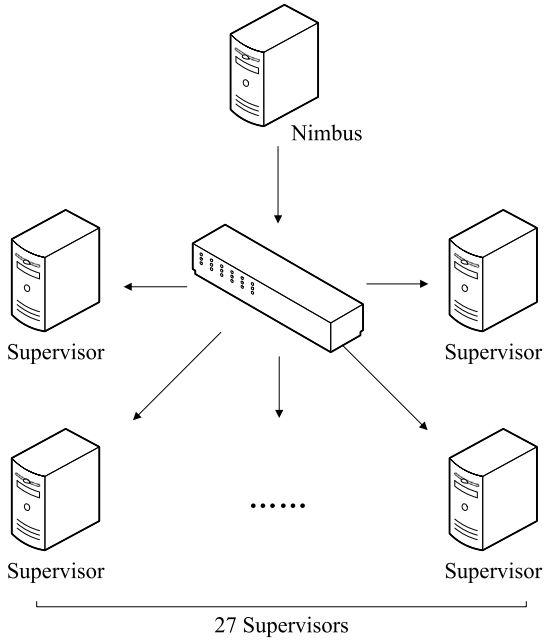

Fig. 8. Experimental topology.

scaling, and utilizes Storm's Metrics API to capture status information of the stream join system. It also stores ID-info pairs in the database.

Aj-Stream employs Redis, a high-performance in-memory database, as both the streaming application data source and intermediate storage. Redis boasts speed, supports various data types, offers robust data manipulation commands, and provides persistence. Its distributed functionality enhances data processing and concurrency, making it suitable for distributed streaming applications. To utilize Redis in streaming applications, Storm offers *RedisBolt* for writing Tuple data to Redis and *RedisLookupBolt* for querying and sending data downstream as Tuples.

4.1. Experimental environment

We deploy Aj-Stream and BiStream on a cluster of the Ali Cloud computing platform. As shown in Fig. 8, the cluster consists of 1 Nimbus node and 27 Supervisor nodes, each of which is equipped with a two-core 2.67 GHz Intel Xeon CPU, 2 GB RAM and a 100 Mbps Ethernet interface card. The same software settings are configured on each node, including Java JDK 1.8 64 bit on top of Ubuntu 20.04.1 64 bit and Storm 2.1.0. In addition, Zookeeper 3.4.14 runs on 3 machines for cluster management and Kafka 2.3.0 on 2 machines as the message queue for the applications. Details are shown in Table 1 .

Table 1
Software configuration of Aj-Stream.

Software	Version
OS	Ubuntu 20.04.1 64 bit
Storm	Apache-Storm-2.1.0
JDK	JDK 1.8 64 bit
Python	Python 2.7.2
Zookeeper	Zookeeper-3.4.14
Kafka	Kafka-2.3.0
Redis	Redis-6.0.5

In our experiments, we use both the real-world data and synthetic data to validate the performance and use Kafka to control the data rates for real input fluctuation simulation. The real-world dataset includes all the Didi online taxi tracks (3 billion) and order data (7 million) in city Chengdu, China during November 2016. The track data is a combination of track records of each taxi, recorded periodically and independently by each taxi, and driver ID, order ID, location information and a timestamp. The order data is a collection of user order information, including order ID, timestamp and location information. We use the location information as the join key in our experiments, and the join operation is to output all the taxis near the passenger's location and all the passengers near the taxi's location. Moreover, we add a data generation source to our Aj-Stream topology to generate synthetic data using real-world data for arbitrary domain sizes with varying skewness. In the synthetic data, the tuples follow *Zipf* distributions controlled by the skewness parameter $-z$, and the scale is controlled by the parameter $size$. We employ the $-fluc$ parameter to control the rate fluctuation of data stream that is input into the data source component. This fluctuation adheres to the range specified by $-fluc$. Since the data is not uniformly distributed in the data stream, the data sub-streams of two neighboring time windows in the data stream may have different skewness. Therefore, the two parameters $-z$ and $-fluc$ can be used to simulate the skewness and stream rate fluctuation of real-world data.

In addition, the experiments are built upon a sliding window strategy that is the core of the evaluation methodology. This strategy involves overlapping consecutive windows, allowing for a continuous processing flow and facilitating forecasts based on the most current data, while retaining the historical context. The sliding window mechanism effectively manages the capture of short-term patterns and considers longer-term trends. This balance considerably enhances the robustness and reliability of our forecasting model, making it highly suitable for handling dynamic data streams and evolving patterns.

4.2. Throughput and latency

Throughput and latency are two critical performance metrics for DSPSs. Average throughput is a metric of the system's processing power (i.e. the speed of processing), defined as the rate at which all join instances in the system generate results. The higher the average throughput, the better the processing performance. Latency represents the length of time it takes for a tuple to be processed in the join instance, defined as the processing time for the tuple to arrive at a join instance until it completes its computation. The lower the latency, the better the processing capability. For real-time stream applications, it is essential to provide millisecond level latency within acceptable limits, while providing the highest possible throughput at the same time. A monitor is designed to monitor the system throughput and latency. When the system is running, the metric values are captured every 60 s.

We first characterize the overall system throughput and latency performance of Aj-Stream, then compare the gain achieved with BiStream

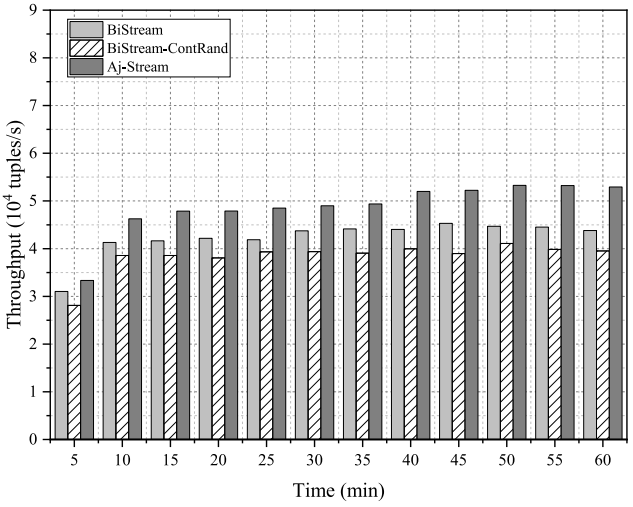

Fig. 9. Throughput variation of BiStream, BiStream-ContRand and Aj-Stream with 24 join instances.

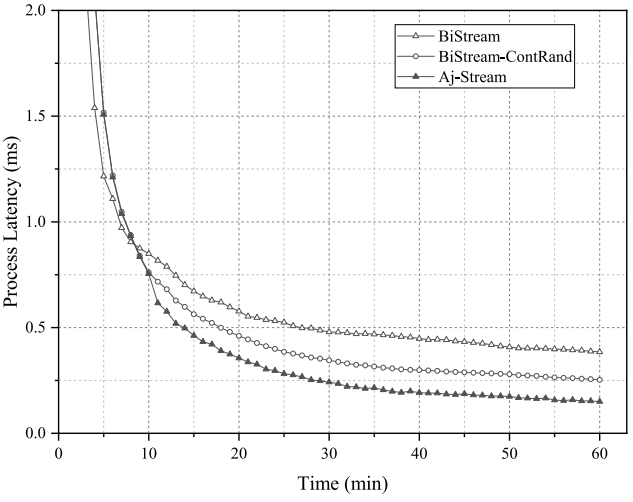

Fig. 10. Latency variation of BiStream, BiStream-ContRand and Aj-Stream with 24 join instances.

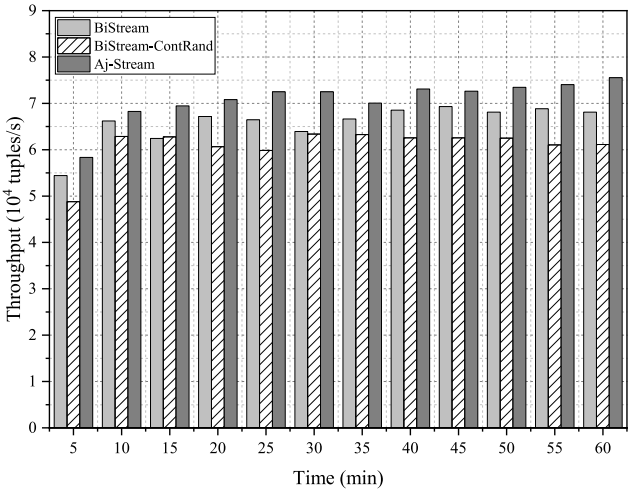

Fig. 11. Throughput variation of BiStream, BiStream-ContRand and Aj-Stream with 48 join instances.

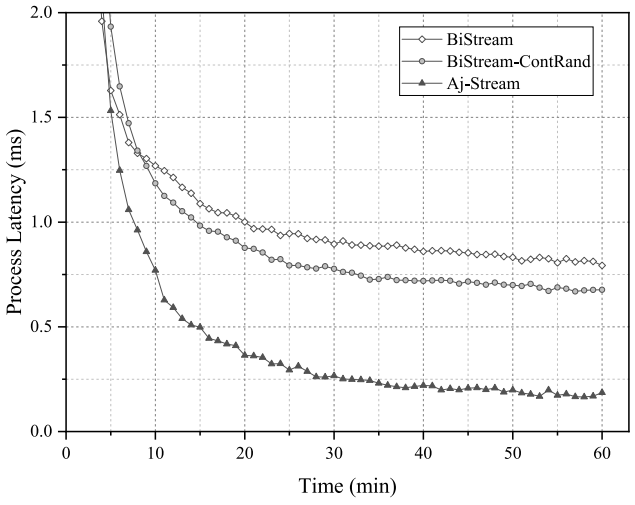

Fig. 12. Latency variation of BiStream, BiStream-ContRand and Aj-Stream with 48 join instances.

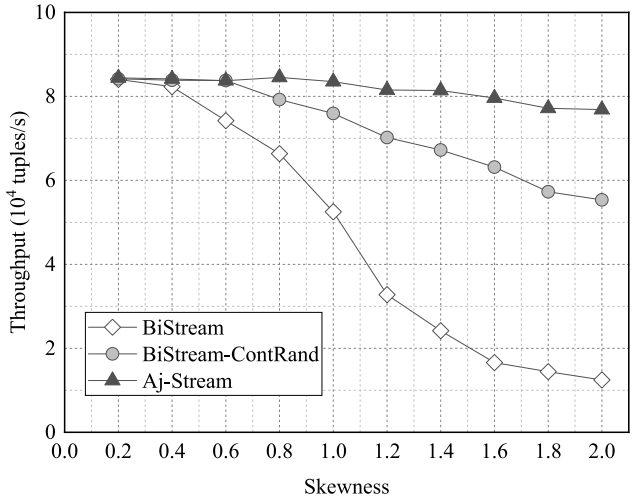

Fig. 13. Average throughput of BiStream, BiStream-ContRand and Aj-Stream with different data skewnesses.

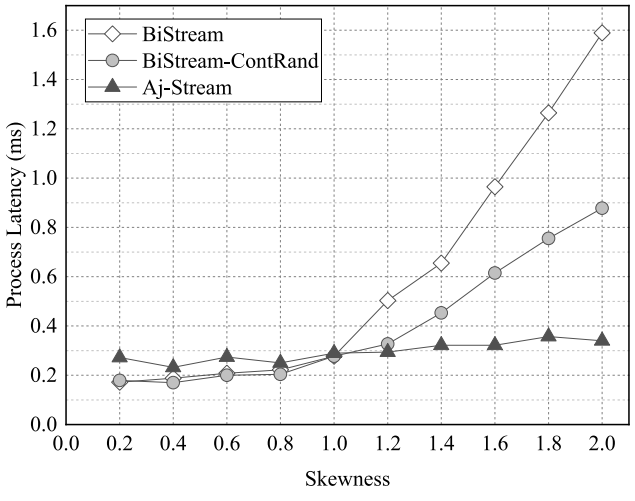

Fig. 14. Average latency of BiStream, BiStream-ContRand and Aj-Stream with different data skewnesses.

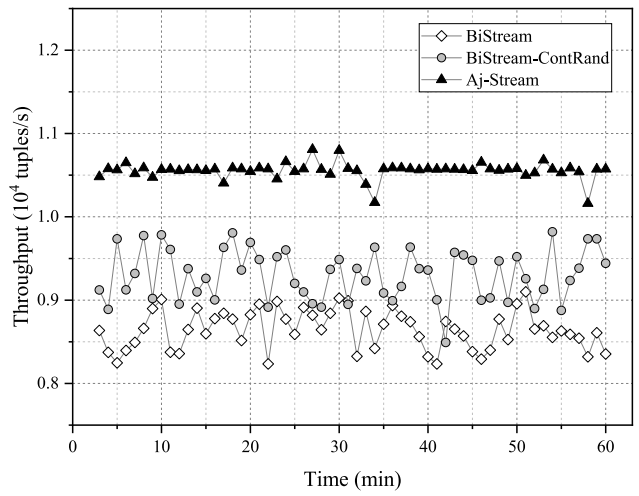

Fig. 15. Throughput variation of BiStream, BiStream-ContRand and Aj-Stream with dynamic data.

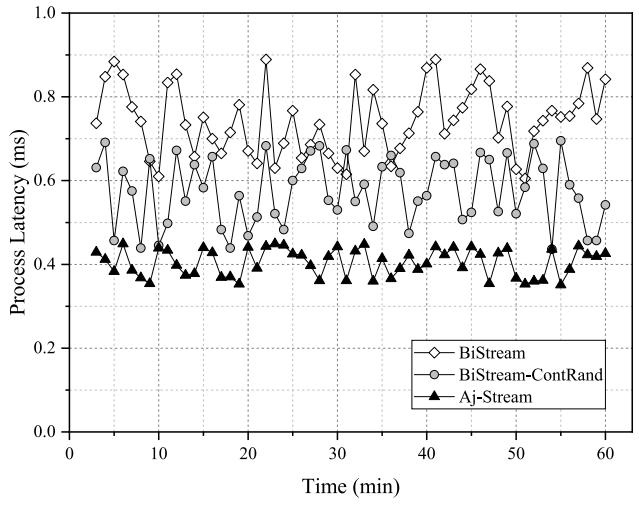

Fig. 16. Latency variation of BiStream, BiStream-ContRand and Aj-Stream with dynamic data.

and BiStream-ContRand. For the parameter settings, data stream rate is 1000 tuples/s, number of join instances is 24 and 48 (equal numbers of joiner-R and joiner-S), data size is 30G, application runs for 60 min when metric data is recorded. Figs. 9–12 show the variation of throughput and latency over time by BiStream, BiStream-ContRand and Aj-Stream running with a join instance number of 24 (Figs. 9 and 10) and with a join instance number of 48 (Figs. 11 and 12), respectively. In Figs. 9 and 11, the throughput of all methods improve when computational resources are increased. Besides, it can be seen that the throughput of Aj-Stream is higher than those of BiStream and ContRand under both resource conditions. It is due to the fact that under the prediction based grouping algorithm and feedback based scaling algorithm, the workloads of the individual parallel join instances of Aj-Stream are more balanced, resulting in better throughput performance. At the same time, as shown in Figs. 10 and 12, the increase in computational resources leads to an increase in latency. This is because increasing the number of instances increases the risk of imbalance, and the greater the skewness of data, the more significant the impact of this risk. In summary, our Aj-Stream produces excellent performance, compared with the other two schemes.

To verify the performance of Aj-Stream under different data skewnesses, we compare the throughput and latency of BiStream, BiStream-ContRand and Aj-Stream on synthetic datasets. For the parameter settings, data stream rate is 1000 tuples/s, number of join instances is 24, and data size is 30 G. The application runs for 60 min with the same size and different skewness of data. To obtain stable throughput and latency, we use data during 31 to 60 min and calculate the average value. Figs. 13 and 14 show the average throughput and latency of the stream joining application under different skewness and BiStream without the ContRand scheme are significantly affected when the data skewness becomes higher. The BiStream-ContRand shows some degradation in both the throughput and latency under higher data skewness. Our scheme performs well under all the data skewness.

In addition, we use the $-fluc$ parameter to control the rate of the data stream to vary dynamically within a range every 10 min to simulate real-world data streams and to verify the throughput and latency performance when processing dynamic data. For the parameter settings, initial rate is 1000 tuples/s, number of join instances is 24 and Didi dataset is used as the input. The application runs for 60 min on the same scaled data, and the metric data is recorded every 60 s. As shown in Figs. 15 and 16, our scheme produces good performance under various skewness and is impacted less on both throughput and latency with the scaling algorithm.

4.3. Load-balancing capability

Since only random routing strategies are available in BiStream without the ContRand solution, in our experiments to verify load balancing performance we only compare with the BiStream-ContRand with a mixed routing strategy built in. We use the degree of load imbalance defined in Eq. (2) to measure the load balancing performance. To verify the load balancing capability under dynamic data, we first set the number of join instances to 24, the data stream rate to 1000 tuples/s and use the Didi dataset, then compare the workload imbalance between Aj-Stream and BiStream-ContRand under different dataset sizes. The data stream rate is set to 1000 tuples/s and a subset of Didi dataset with size of 70G is used as input to compare the workload imbalance between Aj-Stream and BiStream-ContRand with different numbers of instances. The variation of load imbalance for BiStream-ContRand in Fig. 17 indicates that the smaller the dataset size is, the more imbalanced the system tends to be. This is because the system is more prone to producing imbalance allocation with a smaller scale of key set. Whereas Aj-Stream significantly improves the balancing performance at different data sizes. Fig. 18 shows that the workload imbalance is also related to the number of join instances. With additional instances, there is a significant increase in load imbalance for BiStream-ContRand, while Aj-Stream has the better load balancing performance with both instance numbers.

4.4. Prediction model accuracy

To evaluate the accuracy of the prediction model in predicting key frequency distributions, we used the Didi dataset as a data stream with a prediction interval of 1 s in our experiments. First, a monitor is implemented to record the frequency variations of key k in the input stream. Then, the prediction module is used to generate the predictions and fit them to the actual values. The predicted and true values for 20 min of input are collected consecutively and subjected to error analysis. Figs. 19 and 20 show the fit and error analysis of the predicted and true values, respectively. As can be seen in Fig. 19, our prediction method can accurately predict the trend of key frequency changes most of the time. Meanwhile, in Fig. 20, by using the mean absolute error(MAE) to evaluate the predicted values, our prediction method gives good prediction accuracy.

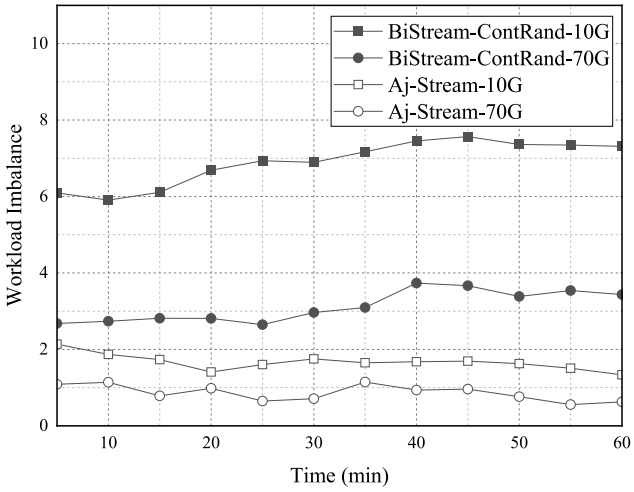

Fig. 17. Real-time degree of load imbalance on different sized datasets.

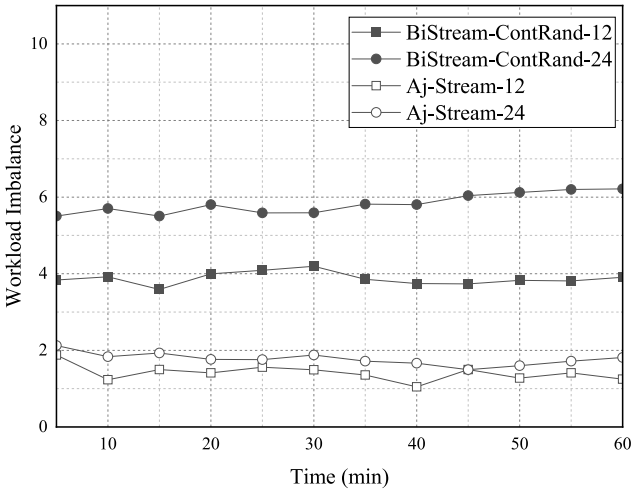

Fig. 18. Real-time degree of load imbalance under different numbers of join instance.

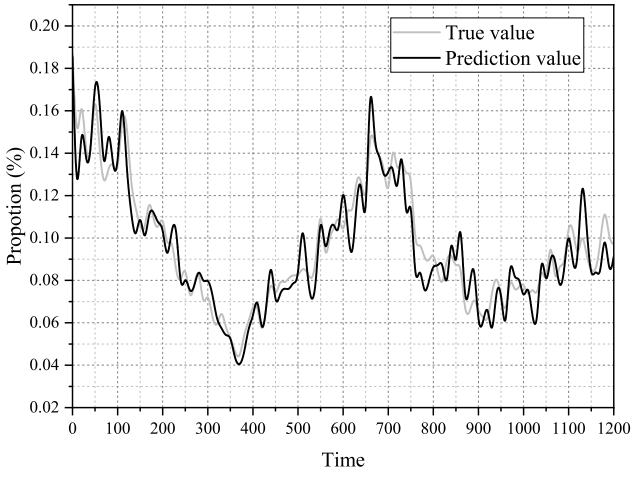

Fig. 19. True value and predicted value of data stream rate.

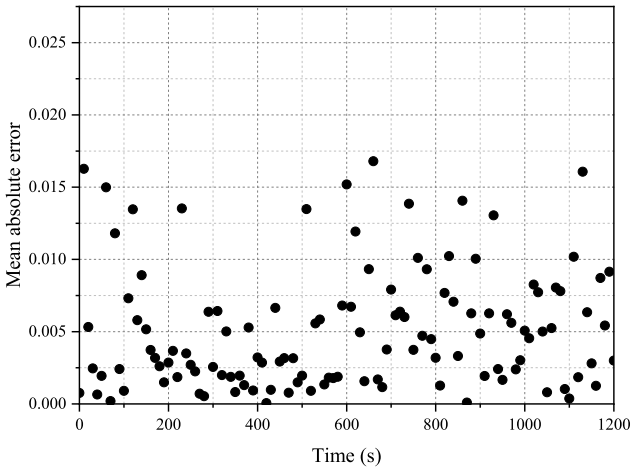

Fig. 20. Mean absolute error of predicted value.

Table 2
Comparison of Aj-Stream and related work.

Parameter	Related work					Aj-Stream
	[12]	[27]	[11]	[22]	[28]	
Performance modeling	✓	✓	✓	✓	✓	✓
Optimization of stream join	✓	✓	✓	×	×	✓
Skewed data	✓	✓	×	×	×	✓
Prediction	×	×	×	×	×	✓
Elasticity	×	×	×	×	×	✓

5. Related work

In this section, we review two broad categories of related works: stream grouping strategies and elastic scaling strategies, as well as comparing our work with the closely related papers as shown in Table 2.

5.1. Stream grouping strategy

Among the variety of partitioning strategies used in existing DSPSs, shuffle grouping and key grouping are the two most commonly used. Shuffle grouping strategy distributes data to all downstream instances in a polling manner. As the data are divided randomly, this strategy does not retain the data states and can only be used to process stateless data. Key grouping uses hash function to route data to a specific downstream instance, so it can ensure the data with the same key be distributed to the same downstream instance. This strategy works best when the frequency of each key is not of much difference. However, the real-world data are often skewed. There are always some keys that are more popular than the others. When processing this type of data, this key grouping strategy will cause some downstream instances overloaded, leading to back pressure and performance bottleneck.

To solve the above problems, many works proposed grouping strategies based on key partitioning. The difficulty of this type of strategies is how to divide the stream into sub-streams. Nasir et al. proposed PKG [25] and a "Two Choices" strategy [29] to allow hot key selection for multiple instances on the basis of PKG. However, these two methods only consider load balancing. When the application topology is large (e.g., the number of workers exceeds 50), there still exists a load imbalance problem. Chen et al. proposed a Popularity-aware heterogeneous DSPS [30]. It uses an ingenious "coin experiment" method to determine potential hot keys. This work defines a key as a hot key if the number of coin flips that result in a 1 is greater than a threshold. This method can quickly adapt to the key frequency changes of highly dynamic data streams. Aslam et al. implemented a pre-filtering-based stream division method in Pre-filtering [31]. It uses pre-filtering to identify hot keys

and to assign multiple instances to them. However, since the filtering is performed in real time, it may increase the data processing delay.

Because a DSPS analyzes and calculates the data that enters the system at any time (that is, operate on the data in units of tuples), it makes the system very sensitive to the real-time distribution of data streams. When a large amount of data streams enter DSPS, even small rate changes can impact the system performance. The number of tuples of a key in a real-time data stream is likely to increase or decrease rapidly in a short period of time. In this case, the static stream grouping strategy often does not work well. If the data partitioning strategy is not adjusted dynamically, once the key frequency is changed in the next time period, there may be a sudden load increase in a downstream instance, making it a performance bottleneck. At the same time, a single key-based partitioning method cannot handle the load balancing problem well. Meanwhile, this method increases the cost of aggregation by destroying the operational semantics of key-based grouping.

More work focused on the mixed strategy of stream grouping. This method uses the key-based partitioning strategy while considering other factors that cause load imbalance. It is possible to compensate for the shortcomings of key segmentation methods to varying degrees while taking full advantage of the flexible character of key partitioning.

Katsipoulakis et al. [32] considered both the load and the aggregation cost, and minimized the load tilt with a lower aggregation cost. Pacaci et al. [33] proposed a mixed stream partition strategy. The head of the data stream is controlled by a routing table. Window semantics are introduced to keep the size of routing table small, and a hash function is used to divide the tail.

Fang et al. [34] designed a framework that combines load balancing and fault tolerance mechanisms at the same time to achieve load balancing and fault tolerance under the premise of low computing costs. They also proposed a hybrid strategy [35], which considers the migration cost on the basis of key partitioning. Even if a workload variance becomes greater, the application can be rebalanced at a smaller migration cost.

Although mixed strategies can dynamically adjust the routing of keys, most hybrid strategies require high migration costs for dynamic adjustment, and a certain amount of memory for the routing table used for migration. Therefore, how to solve the dynamic stream grouping problem at lower memory and migration costs is a major challenge.

Some work used machine learning or reinforcement learning methods to optimize data stream grouping. A common approach is to rely on deep learning to distinguish high-frequency and low-frequency keys, and then use the idea of PKG for stream division.

The DKG [36] proposed by Rivetti et al. learns stream distribution and uses the learned knowledge to construct a global map for key grouping. However, the learning algorithm used in that method is offline and not suitable for real-time scenarios.

Liu et al. in SP-Partitioner [37] rely on learning a batch of data to predict the data distribution of the next batch, and generate a reference table based on the prediction results to guide the data distribution of the next batch. Abdelhamid et al. proposed a data grouping method PartLy based on reinforcement learning [28]. This method can provide effective load balancing data partitions and reduce counting overhead.

5.2. Elastic scaling strategy

In a data stream computing system, elastic scaling strategy is also a key to lowering the system latency and improving throughput [38–44]. With the demand for real-time data processing, distributed real-time computing platforms such as Samza have been proposed. They are highly scalable and fault-tolerant, and meet the need for low latency and high throughput. However, due to the lack of online elastic resource management mechanisms, they all rely on manual changes to configuration files for resource scheduling. In order to maintain high availability and performance at runtime, many papers proposed different optimization schemes in terms of computation or memory, etc.

Jyoti et al. [45] proposed a heterogeneity-aware efficient elastic scaling strategy based on cloud platform. It uses a horizontal hybrid auto-scaling approach to perform scaling operations in both the local and global dimensions. Both the overall resource utilization and the performance of each operator are monitored. It takes into account the structure of the streaming platform and the characteristics of the cloud environment when making scaling decisions, allowing better allocation of resources across dynamic data streams and effectively improving system performance compared to a single dimensional scaling strategy.

In [39], this paper introduces a method for elastic stateful stream processing in Storm, which can dynamically adjust the topology structure and resource allocation according to the changes of data streams. The method utilizes Storm's distributed cache mechanism and state checkpoint mechanism to achieve lossless state migration and recovery. The method also proposes an elastic control strategy based on load prediction and cost–benefit analysis, which can reduce resource consumption while meeting the quality of service requirements.

In [46], This paper proposes an adaptive topology decomposition method, which can optimize the topology structure and task allocation according to the characteristics and load conditions of Storm applications. The method first divides the operators in Storm applications into different groups by cluster analysis, and deploys each group as a sub-topology on a cluster node. Then, the method dynamically adjusts the connection relationship between sub-topologies by monitoring the data transmission volume and delay between them, and performs load balancing according to the network bandwidth between nodes. Finally, the method evaluates the parallelism and processing capacity of operators within sub-topologies, and reduces resource consumption while ensuring quality of service requirements.

All the above elastic scaling strategy methods have certain advantages in DSPSs, but the elastic scaling strategy and strategies for grouping and scheduling is rarely considered. Moreover, these articles are lacking in their consideration of the impact of heterogeneous cloud resources.

6. Conclusions and future work

The problem of load balancing is a key issue in achieving low system latency and high system throughput for stream computing systems. In the face of data streams with rapidly fluctuating frequency distributions and rates, most solutions in current work cannot quickly and adaptively adjust the system configuration to a well-balanced state. This paper proposes an adaptive stream join system, Aj-Stream, to address this problem. It focuses on how to keep the load balancing across parallel instances in a hash stream join systems under dynamic data streams.

We develop a prediction model to predict the key frequency distribution of data streams. Two algorithms are proposed: the prediction-based dynamic grouping algorithm and the feedback-based resource elasticity scaling algorithm. The former is used for grouping strategy adjustment before data stream fluctuation. It ensures that the system maintains low workload imbalance under dynamic data streams; the latter tracks system operator performance and available resources to adjust system resource allocation and to improve resource utilization. Experimental results on Apache Storm platform show that Aj-Stream improves the system throughput and latency performance and significantly reduces system imbalance compared to existing solutions.

Our future work will be focusing on:

(1) Deploying Aj-Stream in large heterogeneous clusters to test the system latency and throughput by processing real dynamic data streams and comparing with existing stream join systems in a heterogeneous cluster environment.

(2) Using reinforcement learning methods to support decision making and to investigate more effective ways of solving the load balancing problem in big data streaming computing systems.

CRediT authorship contribution statement

Dawei Sun: Conceptualization, Methodology, Validation, Writing – original draft, Funding acquisition. **Chunlin Zhang:** Formal analysis, Methodology, Investigation, Writing, Data curation. **Shang Gao:** Formal analysis, Investigation, Writing – review & editing. **Rajkumar Buyya:** Methodology, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 62372419; the Fundamental Research Funds for the Central Universities,China under Grant No. 265QZ2021001; Melbourne-Chindia Cloud Computing (MC3) Research Network, Australia.

References

[1] P.M. Kumar, C.S. Hong, F. Afghah, G. Manogaran, K. Yu, Q. Hua, J. Gao, Clouds proportionate medical data stream analytics for internet of things-based healthcare systems, IEEE J. Biomed. Health Inf. 26 (3) (2022) 973–982.
[2] F. Zhang, C. Zhang, L. Yang, S. Zhang, B. He, W. Lu, X. Du, Fine-grained multi-query stream processing on integrated architectures, IEEE Trans. Parallel Distrib. Syst. 32 (9) (2021) 2303–2320.
[3] D. Sun, S. Gao, X. Liu, X. You, R. Buyya, Dynamic redirection of real-time data streams for elastic stream computing, Future Gener. Comput. Syst. 112 (2020) 193–208.
[4] Apache Storm, 2022, http://storm.apache.org/.
[5] Apache Flink, 2022, https://flink.apache.org/.
[6] Apache Spark, 2022, https://spark.apache.org/streaming/.
[7] Apache Samza, 2022, http://samza.apache.org/.
[8] M. Elseidy, A. Elguindy, A. Vitorovic, C. Koch, Scalable and adaptive online joins, Proc. VLDB Endow. 7 (6) (2014) 441–452.
[9] J. Fang, R. Zhang, X. Wang, T.Z. Fu, Z. Zhang, A. Zhou, Cost-effective stream join algorithm on cloud system, in: International Conference on Information and Knowledge Management, Proceedings, Vol. 24-28-Octo, 2016, pp. 1773–1782.
[10] A. Vitorovic, M. Elseidy, C. Koch, Load balancing and skew resilience for parallel joins, in: 2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016, 2016, pp. 313–324.
[11] Q. Lin, B.C. Ooi, Z. Wang, C. Yu, Scalable distributed stream join processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 2015-May, 2015, pp. 811–825.
[12] S. Zhou, F. Zhang, H. Chen, H. Jin, B.B. Zhou, FastJoin: A skewness-aware distributed stream join system, in: Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019, IEEE, 2019, pp. 1042–1052.
[13] Q. Wang, D. Zuo, Z. Zhang, S. Chen, T. Liu, SepJoin: A distributed stream join system with low latency and high throughput, in: 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2023, pp. 633–640.
[14] Didi Chuxing GAIA Initiative, 2022, https://outreach.didichuxing.com/.
[15] P. Roy, A. Khan, G. Alonso, Augmented sketch: Faster and more accurate stream processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 26-June-20, 2016, pp. 1449–1463.
[16] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, S. Uhlig, Cold filter: A meta-framework for faster and more accurate stream processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2018, pp. 741–756.
[17] H. Herodotou, Y. Chen, J. Lu, A survey on automatic parameter tuning for big data processing systems, ACM Comput. Surv. 53 (2) (2020).
[18] H. Zhang, D. Sun, A. Sajjanhar, R. Buyya, A data stream prediction strategy for elastic stream computing systems, in: Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST, vol. 413 LNICST, 2022, pp. 148–162.
[19] H. Zhang, X. Geng, H. Ma, Learning-driven interference-aware workload parallelization for streaming applications in heterogeneous cluster, IEEE Trans. Parallel Distrib. Syst. 32 (1) (2021) 1–15.
[20] D. Sun, H. He, H. Yan, S. Gao, X. Liu, X. Zheng, Lr-Stream: Using latency and resource aware scheduling to improve latency and throughput for streaming applications, Future Gener. Comput. Syst. 114 (2021) 243–258.
[21] G. Van Dongen, D. Van Den Poel, Influencing factors in the scalability of distributed stream processing jobs, IEEE Access 9 (2021) 109413–109431.
[22] J. Xu, J. Tang, Z. Xu, C. Yin, K. Kwiat, C. Kamhoua, A deep recurrent neural network based predictive control framework for reliable distributed stream data processing, in: Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019, IEEE, 2019, pp. 262–272.
[23] A. Warstadt, A. Singh, S.R. Bowman, Neural network acceptability judgments, Trans. Assoc. Comput. Linguist. 7 (2019) 625–641.
[24] J.T. Connor, R.D. Martin, L.E. Atlas, Recurrent neural networks and robust time series prediction, IEEE Trans. Neural Netw. 5 (2) (1994) 240–254.
[25] M.A.U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, M. Serafini, The power of both choices: Practical load balancing for distributed stream processing engines, in: Proceedings - International Conference on Data Engineering, Vol. 2015-May, 2015, pp. 137–148.
[26] Apache Kafka, 2022, https://kafka.apache.org/.
[27] F. Zhang, H. Chen, H. Jin, Simois: A scalable distributed stream join system with skewed workloads, in: Proceedings - International Conference on Distributed Computing Systems, Vol. 2019-July, IEEE, 2019, pp. 176–185.
[28] A.S. Abdelhamid, W.G. Aref, PartLy: Learning data partitioning for distributed data stream processing, in: Proceedings of the 3rd International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM 2020, 2020, pp. 2–5.
[29] M.A.U. Nasir, G.D.F. Morales, N. Kourtellis, M. Serafini, When two choices are not enough: Balancing at scale in distributed stream processing, in: ICDE 2015, 2015.
[30] H. Chen, F. Zhang, H. Jin, Pstream: a popularity-aware differentiated distributed stream processing system, IEEE Trans. Comput. 70 (10) (2020) 1582–1597.
[31] A. Aslam, H. Chen, H. Jin, Pre-filtering based summarization for data partitioning in distributed stream processing, Concurr. Comput. (March) (2021) 1–25.
[32] N.R. Katsipoulakis, A. Labrinidis, P.K. Chrysanthis, A holistic view of stream partitioning costs, Proc. VLDB Endow. 10 (11) (2017) 1286–1297.
[33] A. Pacaci, M. Tamerzsu, Distribution-aware stream partitioning for distributed stream processing systems, in: Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and beyond, BeyondMR 2018, 2018.
[34] J. Fang, P. Chao, R. Zhang, X. Zhou, Integrating workload balancing and fault tolerance in distributed stream processing system, World Wide Web 22 (6) (2019) 2471–2496.
[35] J. Fang, R. Zhang, T.Z. Fu, Z. Zhang, A. Zhou, X. Zhou, Distributed stream rebalance for stateful operator under workload variance, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2223–2240.
[36] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, B. Sericola, Efficient key grouping for near-optimal load balancing in stream processing systems, in: DEBS 2015 - Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, 2015, pp. 80–91.
[37] G. Liu, X. Zhu, J. Wang, D. Guo, W. Bao, H. Guo, SP-Partitioner: A novel partition method to handle intermediate data skew in spark streaming, Future Gener. Comput. Syst. 86 (2018) 1054–1063.
[38] A. Agrawal, A. Floratou, Dhalion in action: Automatic management of streaming applications, Proc. VLDB Endow. 11 (12) (2018) 2050–2053.
[39] V. Cardellini, M. Nardelli, D. Luzi, Elastic stateful stream processing in storm, in: International Workshop on Parallel Optimization using/for Multi and Many-Core High Performance Computing (POMCO) in Conjunction with 14th International Conference on High Performance Computing and Simulation (HPCS), 2016.
[40] T. De Matteis, G. Mencagli, Elastic scaling for distributed latency-sensitive data stream operators, in: Proceedings - 2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2017, IEEE, 2017, pp. 61–68.
[41] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy, Dhalion: Self-regulating stream processing in Heron, Proc. VLDB Endow. 10 (12) (2017) 1825–1836.
[42] J. He, Y. Chen, T.Z. Fu, X. Long, M. Winslett, L. You, Z. Zhang, HaaS: Cloud-based real-time data analytics with heterogeneity-aware scheduling, in: Proceedings - International Conference on Distributed Computing Systems, Vol. 2018-July, IEEE, 2018, pp. 1017–1028.
[43] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, T. Roscoe, Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows, in: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, 2018, pp. 783–798.
[44] L. Wang, T.Z. Fu, R.T. Ma, M. Winslett, Z. Zhang, Elasticutor: Rapid elasticity for realtime stateful stream processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, 2019, pp. 573–588.

[45] J. Sahni, D.P. Vidyarthi, Heterogeneity-aware elastic scaling of streaming applications on cloud platforms, J. Supercomput. 77 (9) (2021) 10512–10539.
[46] X. Cheng, Q. Liang, L. Ding, Y. Feng, Adaptive topology decomposition for storm, in: 2017 International Conference on Electrical Engineering and Informatics (ICELTICs), 2017.

Dawei Sun is an Associate Professor in the School of Information Engineering, China University of Geosciences, Beijing, P.R. China. He received his Ph.D. degree in computer science from Northeastern University, China in 2012, and conducted the Postdoctoral research in the department of computer science and technology at Tsinghua University, China in 2015. His current research interests include big data computing, cloud computing, and distributed systems. In these areas, he has authored or co-authored over 70 journal and conference papers.

Chunlin Zhang received the Bachelor Degree in computer science and technology from the Henan University of Urban Construction, China in 2019. He is currently studying for his master's degree in Computer Technology from the China University of Geosciences, Beijing, China. His research interests include big data stream computing and distributed systems.

Shang Gao received her Ph.D. degree in computer science from Northeastern University, China in 2000. She is currently a Senior Lecturer in the School of Information Technology, Deakin University, Geelong, Australia. Her current research interests include distributed system, cloud computing, and cyber security.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored over 750 publications and four text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 160 with 139,600+ citations). He served as the founding Editor-in-Chief (EiC) of IEEE Transactions on Cloud Computing and now serving as EiC of Journal of Software: Practice and Experience.