



A skewed stream rebalance strategy for elastic stateful stream processing systems

Dawei Sun^{a,*}, Xinyi Chen^a, Si Lei^a, Jie Wen^a, Atul Sajjanhar^b, Rajkumar Buyya^c

^a School of Artificial Intelligence, China University of Geosciences, Beijing, 100083, China

^b School of Information Technology, Deakin University, Geelong, Victoria 3216, Australia

^c Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

ARTICLE INFO

Keywords:

Data rebalance
Stateful operators
Data skew
Stream computing
Distributed stream processing

ABSTRACT

In distributed stream processing systems, data skew and cluster heterogeneity lead to a serious imbalance in load distribution among parallel processing instances of stateful nodes. However, the existing stream grouping strategies mainly focus on the balance of data division, ignoring the difference of computing instance processing capacities and the impact of network latency on system performance when stateful node data is grouped. In this paper, a skewed stream rebalance strategy for elastic stateful stream computing systems is proposed to optimize the data rebalance of stateful nodes. (1) A data stream graph model, data stream grouping model, application latency model and load skew model are constructed, and the data stream rebalance problem is formalized. (2) A key splitting method based on frequency statistics and classification is proposed, in which frequency statistics combines counter and sketch, and time attenuation predicts the key frequency distribution of data streams. The frequency classification algorithm determines the number of candidate instances, for the key value based on the frequency and threshold, which are scaled at run time according to the downstream instance load condition. (3) A skewed stream rebalance strategy based on latency awareness and instance assignment is proposed, in which monitoring periodically obtains the network latency between upstream and downstream instances, and the data input rate and processing rate of downstream instances. The weights of the instances can be changed adaptively by estimating the latency of the downstream instances according to the queuing model and monitoring information. Finally, the target instances of tuples are randomly selected based on the weights. The experimental results demonstrate that, compared with the baselines, Sr-Stream achieves up to an 84.43% reduction in system latency, improves throughput by up to 3.66×, and significantly reduces load imbalance.

1. Introduction

As the demands for real-time data processing and analysis become more complex, distributed stream processing systems, such as Storm, Flink, Spark and Heron [1], are widely applied across various domains, such as social network analysis [2], real-time event detection [3], internet of things [4]. The rebalancing strategy in the distributed stream computing system can determine which instance receives data and how much data it receives, which can effectively solve the load balance problem in the system and ensure a reasonable stream and fast processing of data. Because of their complex operations, stateful nodes lead to a large amount of load accumulation and become bottlenecks of application processing.

Stateful nodes have data storage capabilities and need to store intermediate results as internal states. When new data is passed in, the nodes compute the output and the internal state based on the new data. For stateful nodes, state migration is required when parallel tasks change [5]. Initially, Flink and Spark paused the entire system during state transitions, which caused latency spikes. Flux takes action to pause only nodes that require state transitions, which reduces latency, but also suspends the entire system if the node is a critical component in the topology. Therefore, when changes occur, stateful nodes should reduce both the volume of state migration and the quantity of states that require migration, which is closely related to the data rebalance between parallel instances of stateful nodes.

* Corresponding author.

E-mail addresses: sundaweicn@cugb.edu.cn (D. Sun), chenxinyi@email.cugb.edu.cn (X. Chen), leisi@email.cugb.edu.cn (S. Lei), wenjie2573@email.cugb.edu.cn (J. Wen), atul.sajjanhar@deakin.edu.au (A. Sajjanhar), rbuyya@unimelb.edu.au (R. Buyya).

<https://doi.org/10.1016/j.jpdc.2026.105362>

Received 29 June 2024; Received in revised form 25 March 2026; Accepted 22 September 2026

Available online 23 September 2026

0743-7315/© 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

Shuffle Grouping (SG) and Key Grouping (KG) [6] are two of the most widely used grouping methods in distributed stream processing systems. SG randomly distributes every tuple to parallel instances according to the polling rule, ensuring that each instance processes a roughly equivalent number of tuples. KG groups streams by using a specified field in a tuple as a key value and assigning the same component of the tuple with the same key value to the same instance. For stateful nodes, SG can effectively achieve data-level load balance, but the state of all key values are required to be saved in each downstream instance. A data stream contains hundreds of thousands of key values, resulting in high memory cost and difficulty to scale. KG can store the state according to the key value, and when the system changes, the number of states which need to be processed is small, and only a part of the state needs to be processed in the old instance. However, when the data skew is large, key grouping distributes all the data of high-frequency key values to the same instance for processing. As a result, the load of a single downstream instance increases compared with other instances, which greatly affects the system performance. In addition, storage, update, clearing, migration, and sharing of stateless nodes require certain resources and time, and are more complex than load balance of stateless nodes.

Elastic stream processing systems should achieve high throughput and low latency in the long run. It is necessary to be capable of dynamically reallocating the workload among instances assigned to nodes when changes occur. Traditional load rebalance strategies aim to implement dynamic rescaling through resource scheduling [7,8], parallelism scaling [9,10] and load migration [11,12]. However, when dealing with a data stream which is constantly changing, these computationally expensive online rebalance strategies may have suboptimal performance.

Skewed data stream often leads to imbalanced load distribution among compute instances. When the data is skewed, over time, instances of stateful nodes that process high-frequency key values will receive too much data, and the processing rate will not keep up with the data input rate; consequently, the processing latency will continue to increase. On the other hand, the upstream node may be compelled to decelerate its processing rate due to the system backpressure mechanism, and the downstream node of the stateful node may need to wait for the processing results of all instances of the stateful node, which will lead to a degradation in system performance, increase the system latency and reduce the throughput.

In this paper, we present Sr-Stream, a skewed stream rebalance strategy for elastic stateful stream processing systems, to optimize the data rebalance of stateful operators. This strategy uses the lightweight frequency statistics algorithm combined with counter and sketch to improve the accuracy of high-frequency key frequency prediction. At the same time, the benefits and costs of key splitting are considered to dynamically adjust the classification threshold, and the appropriate number of candidate instances is calculated. The weight of the downstream instance is determined from the aspects of network latency, the input rate of data stream and processing rate of the instance, and the load balance between the instances is achieved and the traffic between the nodes with high network latency is reduced. Our contributions are listed below:

- We present a generic data stream graph model, a data stream grouping model, application latency model and load skew model, and formalize the load rebalance problem.
- We use key splitting to deal with the bottlenecks resulting from keys with high frequency, and utilize weight adjustment method so that the tuple assignment probability of the instances can be adaptively changed based on the network latency, data rate and processing rate.
- We evaluate Sr-Stream for latency, throughput, and load imbalance. Experiments demonstrate that Sr-Stream reduces latency by up to 84.43%, achieves a 3.66× improvement for throughput, and significantly reduces load imbalance compared with the state-of-the-art methods.

The rest of the paper is structured as follows: Section 2 describes the system model and formalizes problems. Section 3 focuses on the system

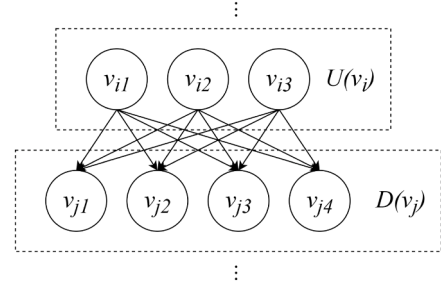


Fig. 1. Instance sets of vertices.

architecture and algorithms of rebalance. Section 4 shows the details of Sr-Stream implementation. Section 5 introduces the experimental environment and analyzes performance evaluation results of Sr-Stream. Section 6 reviews the related work. Finally, conclusions and future work are given in Section 7.

2. Problem statement

In this section, the rebalance problem in distributed stream processing systems is formalized with the system model, which is composed of data stream graph model, data stream grouping model, application latency model and load skew model. The models are mainly used as system abstractions and decision-guidance models for online stream grouping and routing, rather than standalone predictive models requiring separate fitting-accuracy validation. Their effectiveness is evaluated indirectly through the end-to-end experiments in Section 5, where the proposed Sr-Stream, built upon these models, is assessed by latency, throughput, and load-imbalance metrics under different workloads and deployment settings.

2.1. Data stream graph model

The logical topology of a streaming application can be mapped to a directed acyclic logical graph $DAG = (V(G), E(G))$, where $V(G) = \{v_1, v_2, \dots, v_n\}$ is defined as a finite set of n nodes. Every node $v_i \in V(G)$ is an operator. It can create multiple instances $v_i = \{v_{i1}, v_{i2}, \dots, v_{ik}\}$, $k \in \{1, 2, \dots\}$, that can be assigned to nodes and complete work in parallel. $E(G) = \{e_{v_{ik}, v_{jm}} | v_{ik}, v_{jm} \in V(G)\}$ is defined as a finite set including directed edges, and each edge $e_{v_{ik}, v_{jm}}$ indicates communication between instances v_{ik} and v_{jm} , where v_{ik} and v_{jm} are the upstream and downstream instances of $e_{v_{ik}, v_{jm}}$, respectively.

In graph G , data from upstream nodes v_i are sent to downstream nodes v_j . $U(v_i)$ and $D(v_j)$ represent the set of upstream nodes v_i and the set of downstream nodes v_j , respectively. $n_{v_i} = |U(v_i)|$ and $n_{v_j} = |D(v_j)|$ denote the number of instances in $U(v_i)$ and $D(v_j)$. $U(v_i) = \{v_{i1}, v_{i2}, v_{i3}\}$ is the upstream node set and $n_{v_i} = 3$ as illustrated in Fig. 1. Accordingly, $D(v_j) = \{v_{j1}, v_{j2}, v_{j3}, v_{j4}\}$ is the downstream nodes set and $n_{v_j} = 4$. A data stream ds is characterized as an infinite sequence which is composed of data tuples with infinite time distribution and quantity, denoted as $ds = (dt_1, dt_2, \dots, dt_i, \dots)$.

Each data tuple dt_i is a record of attributes $(ts_i, key_i, value_i)$, where $ts_i, key_i, value_i$ is representing the timestamp, key and value of the i^{th} data tuple dt_i . Keys are not unique and multiple tuples may have the same key, which can be used to group tuples to maintain state.

Once the nodes have completed processing, new tuples will be produced. The output tuples produced by the same nodes constitute a new data stream.

2.2. Data stream grouping model

When the number of instances of the downstream nodes v_j for an upstream node v_i is greater than 1, v_i will partition the output data stream

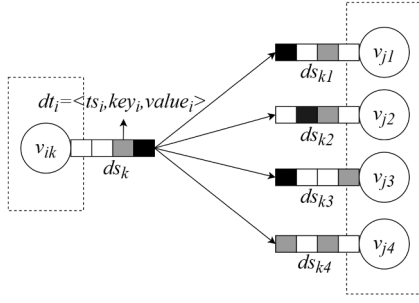


Fig. 2. Partitioning data stream.

among all instances of the downstream nodes v_j through a partition method.

We can map tuples in the data stream output from upstream nodes to downstream nodes by using the partition function $P(ds)$ which achieves different outcomes according to data characteristics and nodes operations. Therefore, the instance v_{ik} of the upstream nodes v_i converts the input data stream ds_{ik}^{in} into ds_{ik}^{out} through function logic and divides it into n_{v_i} independent substreams sent to the downstream nodes n_{v_i} downstream instances. Accordingly, the input stream of the downstream instances v_{jm} is the sum of the substreams sent from all upstream instances v_i as shown in Eq. (1) and Eq. (2).

$$ds_{v_{ik}}^{out} = \bigcup_{m=1}^{n_{v_j}} ds_{km}, m \in \{1, 2, \dots, n_{v_j}\}, \quad (1)$$

$$ds_{v_{jm}} = \bigcup_{k=1}^{n_{v_i}} ds_{km}, k \in \{1, 2, \dots, n_{v_i}\}. \quad (2)$$

In addition to the global grouping that sends each tuple to all downstream instances, in other grouping strategies, a tuple is divided into only one instance sent downstream in a substream, and the same tuple does not exist in each substream, as shown in Eq. (3).

$$\bigcap_{m=1}^{n_{v_j}} ds_{km}(dt_i) = \emptyset, m \in \{1, 2, \dots, n_{v_j}\}. \quad (3)$$

Using the partition function $P(ds)$, a tuple dt_i is sent to the instance $v_{jm} \in D(v_j)$. As illustrated in Fig. 2, the output data stream $ds_k = \{ds_{k1}, ds_{k2}, ds_{k3}, ds_{k4}\}$ is mapped to 4 downstream instances $v_j = \{v_{j1}, v_{j2}, v_{j3}, v_{j4}\}$.

The input rate of downstream instance v_{jm} is the sum of the substreams rates assigned to v_{jm} by all upstream instance of $U(v_i)$ as shown in Eq. (4).

$$\lambda_{v_{jm}} = \sum_{k=1}^{n_{v_i}} \lambda(ds_{km}), k = 1, 2, \dots, n_{v_i}. \quad (4)$$

Stateful nodes often use a hash function to implement the grouping function $P(ds)$. The hash function can divide tuples with the same key value into the same instance, there is no tuple with the same key value in the substream that the upstream nodes sent to instances, and the sum of the key range of each substream is equal to the key range of the upstream output stream, as shown in Eqs. (5) and (6).

$$\bigcap_{m=1}^{n_{v_j}} ds_{km}(key) = \emptyset, m = 1, 2, \dots, n_{v_j}. \quad (5)$$

$$|K| = \sum_{m=1}^{n_{v_j}} |ds_{km}(key)| = \sum_{m=1}^{n_{v_j}} |K_{v_{jm}}|, ds_{km}(key) \subseteq K. \quad (6)$$

Each parallel instance needs to maintain the corresponding state, and the memory space overhead of each instance maintenance status is proportional to the size of the key domains it handled. Assuming that each key value memory overhead indicates $Mem(key)$.

The total memory consumed by all downstream instances is proportional to the size of the key range in the upstream nodes data stream, which is also the minimum memory required by the downstream stateful nodes to maintain their state, as shown in Eq. (7).

$$M(key) = |K| * mem(key), key \in K. \quad (7)$$

2.3. Application latency model

The network latency $LA(e_{v_{im}, v_{ik}})$ of the edge $e_{v_{im}, v_{ik}}$ represents the time required to transfer a tuple between an instance v_{ik} of nodes v_i and instance v_{jm} of nodes v_j , and is related to the size $b(v_{ik})$ of output data re-balance of instance v_{ik} , the bandwidth $band(e_{v_{ik}, v_{jm}})$, the communication distance $dis(e_{v_{ik}, v_{jm}})$, and Propagation rate $p(e_{v_{ik}, v_{jm}})$. With the increase of the system size, the system's network latency increases with the various communication distances. When the instance is transmitted in the same process, the network latency is close to 0. When the instances are within the same node, the network latency is related to the physical speed of the node. When instances are at different nodes, the network latency reaches the highest level, affected by the network bandwidth and propagation rate, and at the same time, as the physical distance increases, as shown in Eq. (8).

$$LA(e_{v_{ik}, v_{jm}}) = \begin{cases} c_0 = 0, & \text{if } v_{ik} \text{ and } v_{jm} \text{ on the same node,} \\ \frac{b(v_{ik})}{band(e_{v_{ik}, v_{jm}})} + \frac{dis(e_{v_{ik}, v_{jm}})}{p(e_{v_{ik}, v_{jm}})}, & \text{otherwise.} \end{cases} \quad (8)$$

A directed path from v_{ik} to v_{jm} can be described as $p_{v_{ik}, v_{jm}}$. The latency of $p_{v_{ik}, v_{jm}}$ is calculated by Eq. (9), where $LA(e_{v_{ik}, v_{jm}})$ denotes the network latency, and $LA(v_{jm})$ denotes the processing latency, including queuing time and computing time.

$$LA(v_{ik}, v_{jm}) = LA(v_{jm}) + LA(e_{v_{ik}, v_{jm}}). \quad (9)$$

As shown in Eq. (10), v_{ik} denotes the average latency between instance v_{ik} and the downstream instances, where $weight_{km}$ denotes the probability that v_{ik} sends tuples to v_{jm} . Furthermore $weight_{k1} + weight_{k2} + \dots + weight_{kn_{v_j}} = 1$, and $LA(v_{ik}, v_{jm})$ will increase with the increase of $weight_{km}$. Therefore, you need to choose the appropriate $weight_{km}$ so that $LA(v_{ik}, v_{jm})$ reaches the smallest level.

$$LA(v_{ik}, v_j) = \sum_{m=1}^{n_{v_j}} weight_{km} \times LA(v_{ik}, v_{jm}). \quad (10)$$

2.4. Load skew model

In the distributed stream computing system, the indicators that affect the load of the calculation node are mainly composed of CPU, memory and bandwidth, and disk resources. In the node clusters, the node i load calculation is shown in Eq. (11), where cpu_i , mem_i , $band_i$ and $disk_i$ represent CPU utilization, memory utilization, bandwidth utilization and disk utilization respectively. The coefficients w_1 to w_4 represent the weights of the corresponding resources, and $w_1 + w_2 + w_3 + w_4 = 1$. The load imbalance is defined in Eq. (12).

$$load_i = w_1 \cdot cpu_i + w_2 \cdot mem_i + w_3 \cdot band_i + w_4 \cdot disk_i, \quad (11)$$

$$LIB = \frac{1}{N} \cdot \sum_{i=1}^N \frac{|load_i - load_{avg}|}{load_{avg}^2}. \quad (12)$$

For $v_j \in V(G)$, the load imbalance is shown in Eq. (13).

$$LIB_{v_j} = \frac{1}{n_{v_j}} \sum_{m=1}^{n_{v_j}} \frac{L(v_{ji}) - L_{avg}(v_j)}{(L_{avg}(v_j))^2}, \quad (13)$$

where $L_{avg}(v_j)$ denotes the average load of all n_{v_j} instances. Normally, LIB_{v_j} needs to meet the constraint $LIB_{v_j} \in [LIB_{min}, LIB_{max}]$. However, it is difficult and unnecessary to achieve optimal balance in the

system, because it has the capacity to tolerate some load imbalance. In our experimental environment, as the system load increases, CPU tends to become the bottleneck earlier than memory, bandwidth, and disk. Accordingly, it is assigned a relatively higher weight of 0.4 in the composite node load metric, whereas each of the other resource types is assigned a weight of 0.2. In general, these weights are intended to reflect the bottleneck sensitivity of different resources and can be adjusted according to resource availability: the weight of a resource can be reduced when it is relatively abundant, and increased when it becomes more constrained.

The definition of instance load is that with the increase of data, the consumption of CPU, memory, and bandwidth increases [13]. Therefore, for each instance $v_{jm} \in v_j$, the load condition is calculated from Eq. (14).

$$L(v_{jm}, \tau) = \text{num}(v_{jm}, \tau) \times \mu_{jm}, \quad (14)$$

where $\text{num}(v_{jm}, \tau)$ is the number of accepted tuples, and μ_{jm} is the processing rate of v_{jm} .

At time τ , the average load $L_{avg}(\tau)$ between instances of nodes v_i is shown in the Eq. (15), where n_{v_j} is the number of instances of nodes v_j .

$$L_{avg}(\tau) = \frac{1}{n} \sum_{m=1}^{n_{v_j}} L(v_{jm}, \tau). \quad (15)$$

At time τ , the load imbalance of instance v_{jm} is shown in the Eq. (16).

$$LIB_{v_{jm}}(\tau) = \frac{|L(v_{jm}, \tau) - L_{avg}(\tau)|}{L_{avg}(\tau)}. \quad (16)$$

The load imbalance between a set of parallel instances of nodes v_i is defined as the average load imbalance of all instances of downstream nodes v_j , as shown in Eq. (17).

$$LIB_{v_j}(\tau) = \frac{1}{n_{v_j}} \sum_{m=1}^{n_{v_j}} LIB_{v_{jm}}(\tau). \quad (17)$$

An effective stream rebalance mitigates load imbalance among instances, which is helpful to reduce the latency and enhance the throughput. The data stream rebalance optimization problem can be formalized by Eq. (18):

$$\begin{cases} \min (LIB_{v_j}(\tau)) \text{ and } \min (LA(v_{ik}, v_j)) \\ \text{subject to } LIB_{min} \leq LIB_{v_j}(\tau) \leq LIB_{max}. \end{cases} \quad (18)$$

The current analysis focuses on modeling the latency and load factors used in online routing decisions. Specifically, Eq. (9)-(10) characterize the latency components considered by Sr-Stream, and Eq. (14)-(17) define the load and load-imbalance metrics used for adaptive key classification and routing. Compared with PKG [6], which mainly addresses skew through partial key grouping, Sr-Stream further incorporates network latency and heterogeneous processing capacity into the routing decision.

3. Sr-Stream: architecture and algorithms

In this section, we introduce the architecture of Sr-stream and focus on four core algorithms of rebalance strategy, which include key frequency statistics, key frequency classification, latency awareness and instance assignment.

3.1. Sr-Stream's architecture

As shown in Fig. 3, Sr-Stream is implemented based on Apache Storm. After an upstream instance generates an output tuple, the key frequency statistics module first updates the frequency summary of the tuple key. The key frequency classification module subsequently uses the frequency summary to determine the key splitting degree, i.e., the

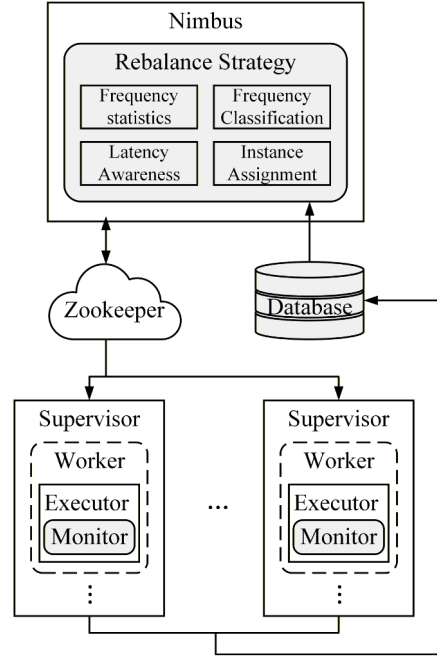


Fig. 3. Sr-Stream's architecture.

number of candidate downstream instances that can receive tuples associated with this key. After the candidate instance set is determined, the latency awareness module further evaluates downstream instances based on runtime metrics collected by the monitor, including network latency, tuple arrival rate and processing rate, and generates an instance weight table. Based on the candidate set and the corresponding weights, the instance assignment module finally selects the target downstream instance for the tuple.

In Fig. 3, the rebalancing strategy is a logical decision-making framework rather than a centralized data processing node. Runtime statistics are collected locally by the Monitor on each worker/executor, whereas routing decisions are performed along the tuple emission path at the upstream executor. Database is responsible for aggregating monitoring summaries and maintaining the parameters required for rebalancing.

3.2. Key frequency statistics

Sr-Stream employs a key frequency statistics module to maintain online estimates of key frequencies at runtime. When an output tuple is generated by an upstream instance, the module first updates the frequency summary of the corresponding key. In data stream S , the total number of current input tuples is defined as m . When the number of occurrences of key value k_i in data stream S is i , the empirical probability of key value k_i occurring in data stream S is defined as $f_{k_i} = i/m$. Then the empirical probability distribution D of key values in the data stream is shown in Eq. (19).

$$D = \langle f_{k_1}, f_{k_2}, \dots, f_{k_L} \rangle, \sum_{i=1}^L f_{k_i} = 1, L = |K|. \quad (19)$$

As shown in Fig. 4, the key frequency statistics module adopts a two-level summary structure composed of a filter and a sketch. When a tuple arrives, the module first checks whether its key is already stored in the filter. If so, the frequency estimate of that key is updated directly; otherwise, the sketch is used to update and estimate its frequency. The filter is designed to retain high-frequency keys for more accurate tracking, while the sketch compactly summarizes the remaining keys with bounded overhead. As shown in Algorithm 1, a time-decay operation is applied to the maintained statistics so that the module can adapt to changes in the stream distribution.

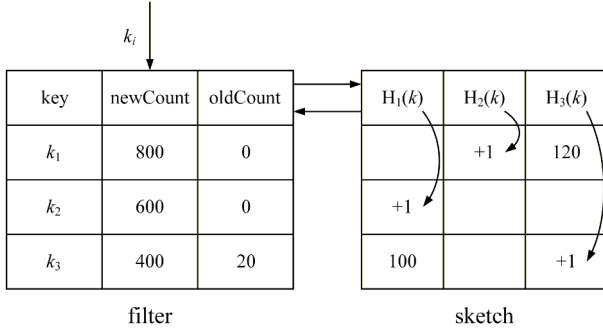


Fig. 4. Key frequency statistics.

When the key value frequency in sketch is higher than the minimum key value frequency in filter, data exchange occurs between filter and sketch. The sketch type we used is the Count-Min Sketch (CMS) [14], because its minimum-counter frequency estimate with bounded one-sided error is well suited for online hot-key detection and for comparing candidate keys with the minimum-frequency item in the filter during the filter-sketch exchange. CMS occasionally identify false hot keys. However, the impact of such cases is limited: they may conservatively assign non-hot keys to a larger candidate instance set, introducing a small amount of additional routing overhead, but they do not affect the system's overall decision-making. Errors in filter only occur when data is exchanged with sketch. So when the error in sketch is smaller, the frequency statistics is more accurate. sketch uses d different hash functions to build a key to matrix mapping. The frequency calculation in sketch is shown in Eq. (20).

$$\forall k_i \in K, \tilde{f}_{k_i} = \min_{0 < t < d} S[t][h_t(k_i)], \tilde{f}_{k_i} \geq f_{k_i}. \quad (20)$$

To ensure the accuracy of sketch, the accuracy parameter is defined as ϵ and the error probability parameter is defined as δ , then $w = 2/\epsilon$ and $d = \log(1/\delta)$. With a probability of at least $1 - \delta$, the error between sketch's estimated occurrence frequency and its true frequency for any one element is less than ϵ times the sum of the true frequencies for all elements. That is, assuming the key error of filter $\tilde{f}_i - f_i = 0$, the total frequency of filter key value is F_c . Therefore, the error expectation of CAS in the two-tier statistical model is shown in Eq. (21).

$$E(\tilde{f}_{k_i}) = \sum_{i=|K|-C}^{|K|} (\tilde{f}_{k_i} - f_{k_i}) \cdot f_{k_i} < \epsilon(1 - F_c)^2. \quad (21)$$

Frequency statistics supports frequency search of key values. When the key value is stored in filter, the average positioning search time is t_c . When the key value is stored in sketch, the first step is to iterate through filter to determine whether the key value is in filter. If not, locate in sketch and the search time is t_s . Therefore, the search time expectation of the statistical model is shown in Eq. (22).

$$E(t_{k_i}) = F_c \cdot t_c + (1 - F_c) \cdot t_s, t_s > t_c. \quad (22)$$

We utilize a time-decay method to adapt to changes in the data stream. The time decay is achieved by multiplying all counts by α at the end of time period T . It is calculated from Eq. (23).

$$C_k(T) = C_k(T-1) \times \alpha, 0 \leq \alpha \leq 1, \quad (23)$$

where $C_k(T-1)$ denotes the count in $(T-1)^{th}$ period, and α denotes the decay factor. The key frequency is defined as the ratio of the count of the key to the total count. α and T jointly control the trade-off between stability and responsiveness of the frequency summary. A larger α or longer T preserves more historical information and yields smoother estimates, while a smaller α or shorter T enables faster adaptation to workload shifts at the cost of higher sensitivity to short-term fluctuations.

The key frequency statistics is defined in Algorithm 1. The input is the output data stream ds_k of upstream instances v_{ik} , period time

Algorithm 1 Key frequency statistics.

Input: data stream ds_k of upstream v_{ik} ; period time T .

Output: frequency $f(k_i)$ of key_i

```

1: Get the current time  $t_{cur}$ .
2: if  $t_{cur} - t_{last} \geq T$  then
3:   Update  $C_k$  and  $C_{total}$  according to (23)
4:   Update  $t_{last} = t_{cur}$ .
5: end if
6: while  $key_i$  in  $ds_k$  do
7:   if  $key_i$  in the filter then
8:      $C_k = C_k + 1$ 
9:   else
10:    if filter not full then
11:      Add  $key_i$  to filter, and update  $C_k = 1$  and  $O_k = 0$ .
12:    else
13:      while line  $i$  in the sketch do
14:        Use the hash function  $h_i(k)$  to get the column  $j$  of the  $key_i$ .
15:         $C_k = C_k + 1$  of column  $j$ 
16:      end while
17:       $C_k$  of  $key_i$  is the minimum count of all line in the sketch.
18:    end if
19:    Get the minimum count  $C'_k$  in filter.
20:    if  $C'_k \leq C_k$  then
21:      Update sketch with  $(k', C_k - O_k)$ .
22:      Add  $key_i$  to filter, and update  $C_k$  and  $O_k$  in the filter to  $C_k$  in the sketch.
23:    end if
24:  end if
25:   $C_{total} = C_{total} + 1$ 
26: end while
27: return  $C_k / C_{total}$ 

```

T . The output is frequency $f(k_i)$ of key_i . When the tuple in the input stream arrives, it will first check for the presence of the key value k within the filter, and if so, C_k is increased by 1; If it is not found and filter does not reach the maximum capacity C , add k to filter and set C_k to the initial values 1 and O_k to 0 (lines 6–11); If the filter is full, the algorithm uses hash functions to map k to the corresponding position in row d of sketch and increases C_k by 1, and C_k is the smallest in row d (lines 12–18). When the number of occurrences of the k key value in filter, use the replacement policy to exchange the two items of filter and sketch (lines 19–24). Finally, add 1 to the total number of tuples. According to the time decay model, when the time period ends, the time decay calculation is performed on the number of keys in the key statistics algorithm, and the time is updated (lines 1–5). The time complexity of Algorithm 1 is $O(n)$, where n is the size of the filter.

This design is scalable because the module does not maintain exact counts for all keys. Instead, it uses a fixed-size filter and a bounded sketch to keep frequency information online, so both memory usage and update overhead remain manageable under high input rates and large key spaces. Moreover, time decay is applied once per statistics period rather than for every tuple.

3.3. Key frequency classification

The size of the tuple key range for all instances of upstream nodes sent to downstream nodes is $L = |K|$. The key value $k_i \in K$ is distributed to multiple instances, i.e., the set of candidate instances of k_i is $D_{k_i} = v_1, v_2, \dots, v_{d_{k_i}}$. The number of waiting instances assigned by key value k_i is $d_{k_i} = |D_{k_i}|$. Both the memory overhead M_i which downstream nodes needs to maintain state and the aggregation overhead AGG_i of key splitting are positively correlated with the key splitting size, and the key

splitting overhead is shown in Eq. (24).

$$O_i = \beta \cdot M_i + \gamma \cdot AGG_i. \quad (24)$$

We set a classification threshold θ to classify key according to frequency. When the low-frequency key is assigned to two candidate instances, a suitable threshold θ should be between $2/n$ and $1/5n$ [15], where n denotes the number of downstream instances. Therefore, the ideal system average load is $1/n$. When the key frequency is higher than the ideal load, it will result in load imbalance. Therefore, we initialize the classification threshold θ to $1/n$ and dynamically change the classification threshold θ according to the load of downstream instances.

Algorithm 2 Key frequency classification.

Input: frequency $f(k_i)$ of key_i ; Threshold θ ; number of downstream instances n ; the maximum and minimum load imbalance $LIB_{\max}$, $LIB_{\min}$.

Output: the number d of candidate instances for key_i .

```

1: if  $f(k_i) \geq \theta$  then
2:    $Key_i$  can be distributed to all downstream instances.
3:    $d = n$ 
4: else
5:    $Key_i$  can be distributed to two downstream instances.
6:    $d = 2$ 
7: end if
8: Get the system load imbalance  $LIB_{v_j}$ .
9: if  $LIB_{v_j} > LIB_{\max}$  then
10:   $\theta = \theta/2$ 
11: end if
12: if  $LIB_{v_j} < LIB_{\min}$  then
13:   $\theta = \theta + \delta$ 
14: end if
15: return  $d$ 

```

The key frequency classification is defined in Algorithm 2. The input is frequency $f(k_i)$ of key_i , Threshold θ , the number of downstream instances n , the maximum and minimum load imbalance $LIB_{\max}$ and $LIB_{\min}$. The output is the number d of candidate instances for key_i . When the key frequency is greater than the threshold θ , d is the total number of downstream instances n . When the key frequency is less than the threshold θ , the candidate instance of the key value is 2 (lines 1–7). LIB_{v_j} is derived from periodic runtime statistics collected by the monitor and computed according to Eqs. (14)–(17). We initialize θ as $1/n$ to align the splitting threshold with the average partition scale. The bounded update band around this scale is used to avoid under-splitting or over-splitting, while the asymmetric update rule (halving under high imbalance and linear increase under low imbalance) is designed for fast response and conservative recovery. When the system load imbalance of a downstream instance is greater than the maximum load imbalance defined by the application, the algorithm will halve the threshold θ to restore the system load balance. When the downstream instance load imbalance is less than the minimum load imbalance defined by the application, the algorithm linearly increases the threshold θ . The time complexity of Algorithm 2 is $O(1)$.

3.4. Latency awareness

Suppose the arrival rate of instance v_{jm} is λ_{jm} and the processing rate is μ_{jm} , where $\rho_{v_{jm}} = \lambda_{v_{jm}}/\mu_{jm}$. When $\rho_{v_{jm}} \geq 1$, the instance processing speed is lower than the new tuple speed, and the instance waiting queue accumulates more tuples as time progresses, resulting in infinite queuing latencies. Conversely, when $\rho_{v_{jm}} < 1$, tuples are processed faster than tuples arrive. The average number of tuples $E[L]_{v_{jm}}$ and the average queue length $E[L_q]_{v_{jm}}$ can be calculated as shown in equations Eq. (25)

and Eq. (26).

$$E[L]_{v_{jm}} = \frac{\lambda_{v_{jm}}}{\mu_{v_{jm}} - \lambda_{v_{jm}}}, \mu_{v_{jm}} > \lambda_{v_{jm}}, \quad (25)$$

$$E[L_q]_{v_{jm}} = \frac{\lambda_{v_{jm}}^2}{\mu_{v_{jm}}(\mu_{v_{jm}} - \lambda_{v_{jm}})}, \mu_{v_{jm}} > \lambda_{v_{jm}}. \quad (26)$$

We adopt an $M/M/1$ -style approximation for latency estimation, where the tuple arrival process is approximated as Poisson, the service time is approximated as exponentially distributed, and each downstream instance is modeled as a single-server queue under the stability condition $\lambda < \mu$. This model is used for lightweight online latency estimation and relative instance ranking in routing decisions. When the actual workload deviates from these assumptions, e.g., under bursty arrivals or non-exponential service times, the absolute accuracy of the estimated latency may be affected, but the model still provides a practical approximation for online routing. The model captures heterogeneity in network latency and effective processing capacity. CPU-side differences are reflected through the monitored processing rate, while memory effects are indirectly reflected through processing-rate and queueing variations.

Based on the Erlang formula, the total time T of the tuple follows the exponential distribution of parameter $\mu - \lambda$, and the overall processing latency of the tuple includes the queuing time and the calculation time, so the average processing latency $LA(v_{jm})$ and the average queuing latency $LA(v_{jm})_q$, as shown in Eq. (27) and (28).

$$LA(v_{jm}) = E[LA(v_{jm})] = \frac{1}{\mu_{v_{jm}} - \lambda_{v_{jm}}}, \mu_{v_{jm}} > \lambda_{v_{jm}}, \quad (27)$$

$$LA(v_{jm})_q = LA(v_{jm}) - \frac{1}{\mu_{v_{jm}}} = \frac{\lambda_{v_{jm}}}{\mu_{v_{jm}}(\mu_{v_{jm}} - \lambda_{v_{jm}})}, \mu_{v_{jm}} > \lambda_{v_{jm}}. \quad (28)$$

Therefore, the completion time of tuples is related to the communication latency between upstream and downstream instances and the processing latency of downstream instances. Monitors are placed on each instance to detect the arrival rate and processing rate of the instance. Assume that the downstream instance v_{jm} and upstream instance v_{ik} , network latency is $LA(e_{v_{ik}, v_{jm}})$. The estimated latency of the tuple from downstream instances to the end of instance processing is shown in Eq. (29).

$$LA(v_{ik}, v_{jm}) = LA(e_{v_{ik}, v_{jm}}) + \frac{1}{\mu_{v_{ik}} - \lambda_{v_{jm}}}, \mu_{v_{ik}} > \lambda_{v_{jm}}. \quad (29)$$

In Sr-Stream, each upstream instance has its own grouping and runs the rebalance algorithm. Therefore, for each upstream instance, there is a local imbalance due to network latency. However, for all upstream instances, a relative global balance will eventually be reached. In order to reduce the influence of excessive local imbalance on global balance, $weight_1 LA(v_{ik}, v_{j1}) = weight_2 LA(v_{ik}, v_{j2}) = \dots = weight_n LA(v_{ik}, v_{jn})$ is made to reach the lower limit of local imbalance on the basis of minimizing the communication volume between instances with high communication latency. Therefore, the weights of each downstream instance are calculated by Eq. (30).

$$weight_i = \frac{(LA(v_{ik}, v_{jm}))^{-1}}{\sum_{l=1}^n (LA(v_{ik}, v_{jl}))^{-1}}. \quad (30)$$

The latency awareness is defined in Algorithm 3. The input is the upstream instance v_{ik} and the downstream instance set D_{v_j} , and the output is the downstream instance weight table of the instance. When the system first starts running, the upstream instance obtains the communication time with all downstream instances and initializes the instance weight to the reciprocal of the network latency (lines 2–5); then the upstream instance periodically obtains the input of the downstream instance rate and output rate, estimating instance processing latency based on queuing model and network latency, and calculate the weight using the inverse of the latency (lines 6–12). During initialization and runtime

Algorithm 3 Latency awareness.**Input:** upstream instances v_{ik} ; downstream instances set D_{v_j} .**Output:** weight table $weight[1, \dots, n_{v_j}]$.

```

1: while instance  $v_{jm}$  in  $D_{v_j}$  do
2:   Get the network latency  $LA(e_{v_{ik}, v_{jm}})$ .
3:   Initialize  $weight[m]$  by reciprocal of  $LA(e_{v_{ik}, v_{jm}})$ .
4:   Normalizes  $weight[1, \dots, n_{v_j}]$  according to (30).
5:   Sort  $weight[1, \dots, n_{v_j}]$  by weight in a descending sequence.
6: end while
7: if monitoring information arrives then
8:   while instance  $v_{jm}$  in  $D_{v_j}$  do
9:     Get the network latency  $LA(e_{v_{ik}, v_{jm}})$ .
10:    Get data input rate  $\lambda_{v_{jm}}$  and processing rate  $\mu_{v_{jm}}$ .
11:    Update  $weight[m]$  from  $v_{ik}$  to  $v_{jm}$  by reciprocal of (29).
12:    Normalizes  $weight[1, \dots, n_{v_j}]$  according to (30).
13:    Sort  $weight[1, \dots, n_{v_j}]$  by weight in a descending sequence.
14:   end while
15: end if
16: return  $weight[1, \dots, n_{v_j}]$ 

```

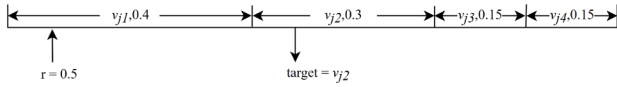


Fig. 5. Weighted randomized instance selection.

adjustment, the algorithm normalizes the weights and sorts the weight table from large to small. The time complexity of Algorithm 3 is $O(n)$, where n is the total number of downstream instances.

3.5. Instance assignment

An example of random distribution of weights is shown in Fig 5. First of all, a random number r is 0.5, the weight of v_{j1} is 0.4, at this time $r - 0.4 = 0.1 > 0$, v_{j1} is not selected. When the weight of v_{j1} is 0.3, at this time $0.1 - 0.3 < 0$, the random number r falls in the weight interval of v_{j2} , then the target instance is v_{j2} .

The instance assignment is defined in Algorithm 4. The input is the output data stream ds_k of the upstream instances v_{ik} and the downstream instance set D_{v_j} , and the output is the target instance v_{jm} of each tuple. When the tuple arrives, the instance weight table of the current time period is first obtained according to Algorithm 3, and then the number of candidate instances of the tuple is calculated according to Algorithm 1 and 2. When the number of instances, d , equals the total number of downstream instances, n , the candidate instance set comprises all downstream instances. A random number between 0 and 1 is selected and the instance weight table is traversed. When the random number falls within the weight interval of an instance, that instance will be the target instance (lines 1–13). When the number of instances d is 2, two independent random hash functions are used to calculate the candidate instances $weight[v_{jH_1(k)}]$ and $weight[v_{jH_2(k)}]$ of the key value in the downstream instance. A random number is selected between 0 and the sum of weights of the two candidate instances, and when the random number falls within the weight interval of an instance, that instance will be the target instance (lines 14–25). Finally, the upstream instance distributes the tuple to the target instance target.

4. Sr-Stream implementation

We implement Sr-Stream system building upon the analysis and algorithms described in the above section. In this section, we present the implementation details of Sr-Stream.

The monitor mainly performs statistical operations on the runtime state of the system in real-time, and provides the status information

Algorithm 4 Instance assignment.**Input:** data stream ds_k of upstream v_{ik} ; downstream instances set D_{v_j} .**Output:** target instance v_{jm} of each tuple.

```

1: Get  $weight[1, \dots, n_{v_j}]$  by algorithms 3.
2: while  $key_i$  of tuple  $dt_i$  in  $ds_k$  do
3:   Get the number  $d$  of candidate instances by algorithms 1 and 2.
4:   if  $d == n$  then
5:     Candidate instances are all downstream instances.
6:     Select a random number  $r$  from 0 to 1.
7:     while instance  $v_{jm}$  in  $D_{v_j}$  do
8:       Update  $r = r - weight[v_{jm}]$ .
9:       if  $r \leq 0$  then
10:        Assign tuple  $dt_i$  to instance  $v_{jm}$ .
11:        Break.
12:       end if
13:     end while
14:   else
15:     while  $d = 1, 2$  do
16:       Use hash function  $H_d(k)$  to determine candidate instances  $v_{jH_d(k)}$ .
17:     end while
18:     Select a random number  $r$  from 0 to the sum of  $weight[v_{jH_1(k)}]$  and  $weight[v_{jH_2(k)}]$ .
19:     if  $r \leq weight[v_{jH_1(k)}]$  then
20:       Assign tuple  $dt_i$  to instance  $weight[v_{jH_1(k)}]$ .
21:     else
22:       Assign tuple  $dt_i$  to instance  $weight[v_{jH_2(k)}]$ .
23:     end if
24:   end if
25: end while
26: return target instance  $v_{jm}$  of each tuple

```

required by the data stream grouping to the grouping. Runtime monitoring metrics are periodically stored in the database. After recovery, Sr-Stream re-initializes the metrics based on the latest monitored information, while the frequency summary is rebuilt from subsequent stream observations. System run statistics in Storm are updated using a sliding window, sampling and measuring the number of tuples received and processed by each executor in the time window, as well as the latency. Worker nodes report heartbeats to Zookeeper, and Nimbus displays them on the Storm UI. In addition, Storm provides built-in statistics on which the Sr-Stream monitors operate. The monitor needs to obtain the network latency between the upper and lower instances, the input rate and the processing rate of the downstream instances. The input stream rates and output rates returned to the downstream instance are calculated by calling the associated classes of Storm's IMetric interface.

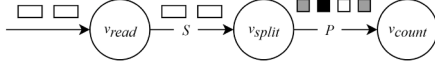
The hostnames and port assignments of upstream and downstream instances are obtained by invoking the relevant methods of context, and the network latency of upstream and downstream instances is obtained by using the measurement tool. When the system is running, the system first creates and registers the context object of the task with the statistics. Then a timer is set in the executor that sends the timing information to the task on time. When the task receives the scheduled information, it sends the statistical information. The upstream instance receives runtime information from the downstream instance every 30 seconds. This period is chosen as a trade-off between monitoring overhead and adaptation responsiveness. A shorter period enables faster updates of LIB , θ , and routing parameters, but increases control overhead and sensitivity to short-term fluctuations; a longer period reduces overhead and smooths the statistics, but may delay adaptation to rapid workload changes.

The rebalance strategy, which is the core of Sr-Stream, mainly implements custom grouping through the CustomStreamGrouping

Table 1

Hardware configuration of computing nodes in the cluster.

Type	CPU cores	Memory	Bandwidth	Disk
1	2 vCPUs	2 GB	100 Mbps	20 GB
2	1 vCPU	1 GB	100 Mbps	20 GB

**Fig. 6.** The logical topology of WC.

interface and has two main parameters. The first parameter is componentId, which represents the id of the upstream component. The second parameter is the object 'SrGrouping' of a custom grouping class. The preparation and chooseTasks methods of 'SrGrouping' are mainly implemented. The prepare method parameters include topology information, data stream id, and downstream instance set. The prepare method is used to initialize group information. chooseTasks returns the target instance of the data grouping. In chooseTasks, a custom calculateWeight() is first used to calculate the weight of all downstream instances from runtime information. The key frequency is then calculated using the CounterData() method of the object of the CounterAndSketch class based on the key of the current tuple, and the number of candidate instances of the tuple is determined by the threshold and frequency. Then, the candidate instance set of all downstream instances is calculated according to the number of candidate instances, and targetTaskId is randomly selected according to the weight in the candidate instances.

The additional overhead introduced by Sr-Stream comes from online frequency-summary maintenance, periodic collection/reporting of monitoring information, and periodic updates of routing parameters. These modules do not process the actual tuple stream in a centralized manner; instead, the actual tuples are still transmitted directly between executors along the normal data path. Therefore, Sr-Stream does not introduce a new single-point data-plane bottleneck. Moreover, since the frequency summary is maintained using a fixed-size filter and a bounded sketch, and the monitoring/parameter updates are triggered periodically rather than through per-tuple global synchronization, the extra runtime overhead remains manageable in practice.

5. Performance evaluation

This section focuses on the comparison experiments of Sr-Stream, key grouping (KG), partial key grouping (PKG), FlexSP [16] and FISH [17]. The experimental part includes the comparison of system performance at input rate of 4000 tuples/s and parallelism level of 16, and the comparison of system latency, throughput, and load imbalance at different parallelism levels and input rates [18].

5.1. Experimental environment

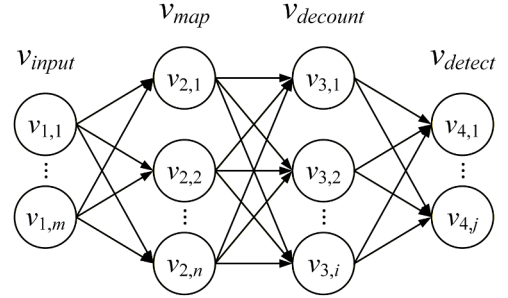
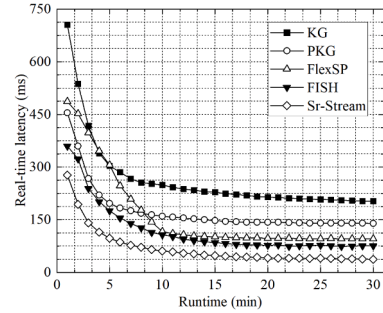
The computing cluster is composed of 9 computing nodes, which are divided into two types. Among them, one node is the master node to support Nimbus, and other nodes process service logic as slave nodes. Zookeeper is built on three nodes of the primary and secondary nodes and is responsible for the coordination between the primary and secondary nodes. The hardware configuration is shown in Table 1, and Table 2 lists the software configuration. The heterogeneous configurations in Table 1 provide the basis for runtime performance differences across cluster nodes. In the evaluation, we focus on the end-to-end behavior of Sr-Stream under such heterogeneous deployment. Sr-Stream exploits this heterogeneity through routing decisions based on monitored processing rates and network latency.

The experiments used Amazon reviews dataset [19] for WordCount (WC) application, and on-demand ride-hailing dataset [20] for Road

Table 2

Software configuration of the cluster.

Software	Version
Storm	apache-storm-2.1.0
OS	Ubuntu 20.04.1 64bit
Python	python 2.7.2
Zookeeper	zookeeper-3.4.14
MySQL	MySQL-5.1.73
JDK	jdk1.8 64bit

**Fig. 7.** The instance topology of RCD.**Fig. 8.** Real-time latency of WC under stable input rate and parallelism level.

Condition Detection (RCD) application [21]. The data records from the datasets are emitted as input tuples to emulate realistic skewed key distributions. As illustrated in Fig. 6, the topology of WordCount consists of three nodes, v_{read} , v_{split} and v_{count} . v_{read} reads the data records and sent them to the downstream instances, v_{split} splits each data record into individual words, and v_{count} counts the occurrences of each word and outputs the results in real time. The instance topology of RCD includes 4 nodes, as shown in Fig. 7, where v_{input} inputs records containing taxi ID, GPS location, and timestamp. v_{map} maps the GPS coordinates of each record onto a map grid. $v_{decount}$ counts and deduplicates the number of distinct taxi IDs in each region. v_{detect} computes the taxi count for each region and compares it against a predefined threshold; when the threshold is exceeded, a road congestion detection result is generated.

5.2. Latency performance

The real-time latency of Sr-Stream and the baselines is shown in Fig. 8 (WC) and Fig. 9 (RCD). At the beginning of the system operation, there was a high latency due to loading of the configuration file, task allocation and instance startup. However, Sr-Stream was significantly better than the baselines. As time goes on, the latency gradually decreases and tends to stabilize. Within 20 to 30 minutes, for WC, the latency of Sr-Stream is approximately 37.50 ms, the latency of KG, PKG, FlexSP and FISH are approximately 205.80 ms, 140.30 ms, 96.00 ms and 75.40 ms, respectively.

The same trend is exhibited for RCD. The final latency of Sr-Stream and the baselines are 112.01 ms, 350.76 ms (KG), 269.98 ms (PKG),

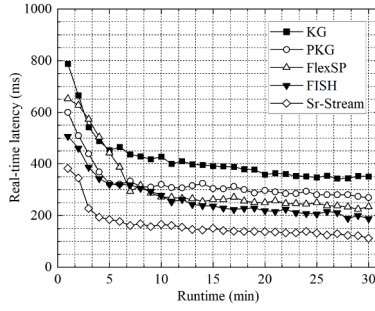


Fig. 9. Real-time latency of RCD under stable input rate and parallelism level.

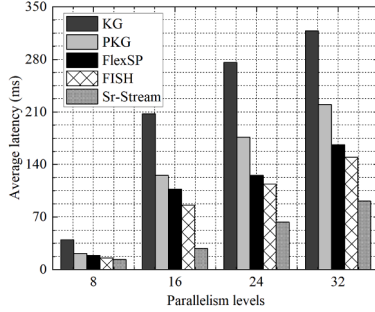


Fig. 10. Average latency of WC under different parallelism levels.

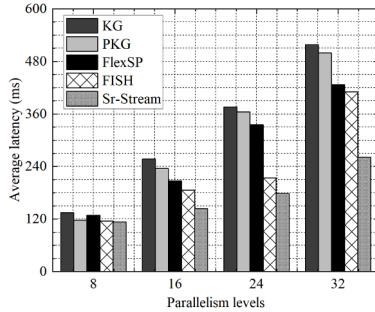


Fig. 11. Average latency of RCD under different parallelism levels.

234.70 ms (FlexSP) and 188.02 ms (FISH), respectively. Although PKG mitigates hot-key skew through its partitioning mechanism, it does not consider network latency or instances' processing capabilities. In contrast, Sr-Stream adaptively adjusts the candidate instance set based on key frequency and determines routing decisions according to runtime metrics, thereby delivering better latency performance. Compared with the baselines, Sr-Stream decreases the real-time latency by approximately 50.92–81% for WC, and 40.43–68% for RCD.

The average latency under different parallelism levels is shown in Fig. 10 (WC) and Fig. 11 (RCD). When the parallelism level is 8, compared to baselines, Sr-Stream reduces the average latency by approximately 11.07–65.52% for WC, and reduces the average latency by up to 15.53% for RCD. As parallelism increases, Sr-Stream achieves better latency performance than the baselines. When the parallelism level is 32, Sr-Stream reduces the average latency by approximately 39.03–71.29% for WC, and reduces the average latency by approximately 36.38–49.56% for RCD.

The average latency at different input rates is shown in Fig. 12 (WC) and Fig. 13 (RCD). The average latency of Sr-Stream at different rates is significantly reduced compared with the baselines. At an input rate of 5000 tuples/s, the average latency of Sr-Stream is 22.31 ms for WC, which is 18.93–71.26% lower than those of baselines. When the input rate is 10,000 tuples/s, the average latency of Sr-Stream is 32.11 ms, which reduces average latency by 53.31–84.43% compared to baselines.

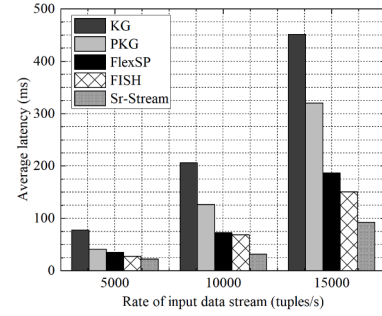


Fig. 12. Average latency of WC under different input rates.

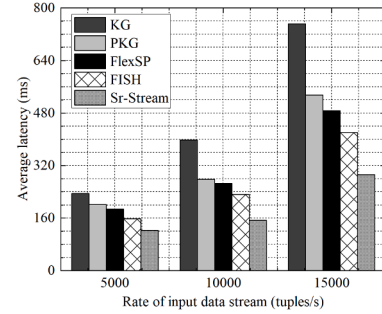


Fig. 13. Average latency of RCD under different input rates.

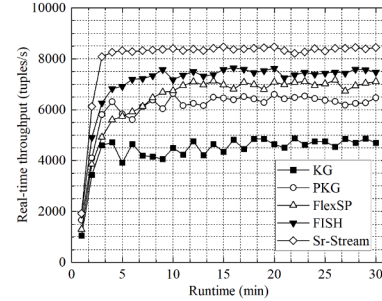


Fig. 14. Real-time throughput of WC under stable input rate and parallelism level.

When the input rate is 15,000 tuples/s, Sr-Stream reduces the average latency by 38.65–79.5%. The same trend is exhibited for RCD. Compared to baselines, Sr-Stream reduces the average latency by 22.35–47.87%, 34.22–61.47% and 30.49–61.08% under the input rates of 5000, 10000, and 15,000 tuples/s, respectively. It can be concluded from the experiment that when the system reaches stability, the latency of Sr-Stream is remarkably lower than those of baselines. Due to its elasticity, Sr-Stream can adapt to different parallelism levels and different input rates.

5.3. Throughput performance

The real-time throughput is shown in Fig. 14 (WC) and Fig. 15 (RCD). Sr-Stream is less affected by system startup, and the throughput is remarkably higher than those of baselines. After 5 minutes of operating WC, the real-time throughput of Sr-Stream, KG, PKG, FlexSP and FISH are 8323.51, 3915.85, 5829.61, 5740.21 and 6922.79 tuples/s, respectively. In addition, Sr-Stream is more stable at runtime, with roughly the same throughput between 5–30 minutes, while the baselines fluctuate more. For RCD, Sr-Stream's final throughput is 7289.56 tuples/s, compared to those of baselines, Sr-Stream increases the final throughput by 23.61–117.32%.

Fig. 16 (WC) and Fig. 17 (RCD) show the average throughput comparison when the parallelism level is adjusted. When the parallelism

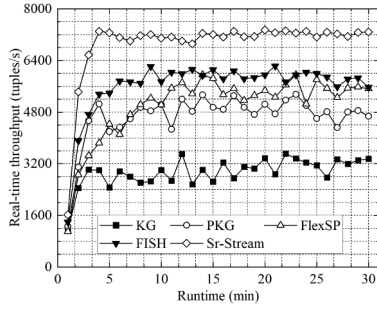


Fig. 15. Real-time throughput of RCD under stable input rate and parallelism level.

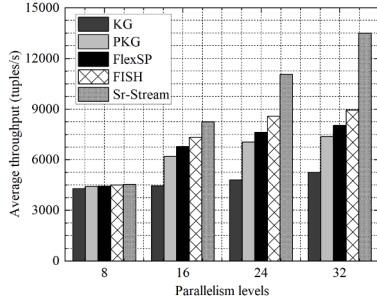


Fig. 16. Average throughput of WC at different parallelism levels.

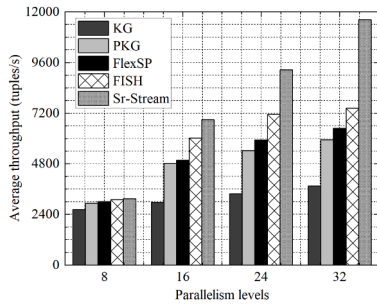


Fig. 17. Average throughput of RCD under different parallelism levels.

level is 8, both of running WC and RCD, the average throughputs across all five methods is comparable. When parallelism is 16, the average throughputs of Sr-Stream is 8253.2 tuples/s for WC, and 6889.54 tuples/s for RCD. Compared to baselines, Sr-Stream increases the average throughput by 12.73–85.50% and 14.35–132.38% for WC and RCD, respectively. As parallelism level further increases, the throughput improvement achieved by Sr-Stream becomes more pronounced. At 32 parallelism level, the average throughputs of Sr-Stream are 13492.9 tuples/s for WC, 11634.25 tuples/s for RCD, which achieve improvements of $1.34\text{--}2.57\times$ for WC and $1.36\text{--}2.1\times$ for RCD compared to the baselines.

Fig. 18 (WC) and Fig. 19 (RCD) show the average system throughput comparison when the input stream rate is adjusted, which exhibit that the average throughput increases with the increasing of input rate. When the input rate is 5000 tuples/s, the average throughput of Sr-Stream and the baselines at the same level due to the low system load. As the input rate increases to 10,000 tuples/s, the average throughput of Sr-Stream is 8243.51 tuples/s for WC and 6359.42 tuples/s for RCD. Compared to the baselines, Sr-Stream increases the average throughput by 12.49–86.81% for WC, and 8.31–122.82% for RCD. At a higher input rate of 15,000 tuples/s, the average throughput of Sr-Stream is 12461.08 tuples/s for WC, and 10785.65 tuples/s for RCD, which achieve improvement of $1.63\text{--}2.75\times$ for WC and $1.76\text{--}3.66\times$ for RCD. This is because at high input rates, the baselines are limited by slow instances, limiting the input rate, while Sr-Stream makes better use of resources,

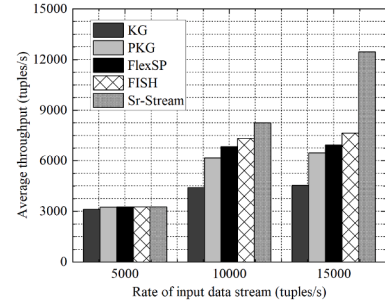


Fig. 18. Average throughput of WC under different input rates.

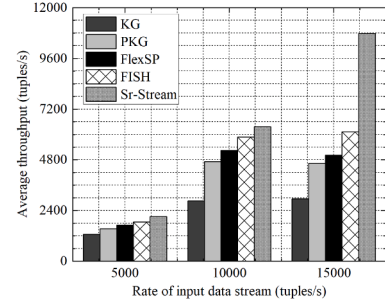


Fig. 19. Average throughput of RCD at different input rates.

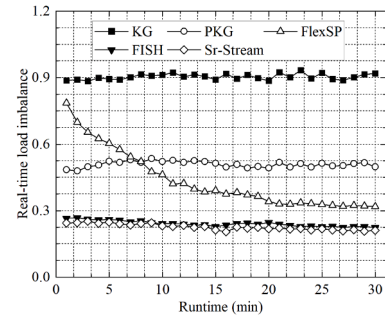


Fig. 20. Real-time load imbalance of WC under stable input rate and parallelism level.

has not yet reached the processing limit, and can continue to process the input tuples.

The experiment proves that Sr-Stream improves system throughput. Especially when the system processing rate is low or the input rate is high, the baselines are stuck in a bottleneck, and Sr-Stream can still work normally.

5.4. Load imbalance performance

At the parallelism level is 16, the real-time load imbalance is shown in Fig. 20 (WC) and Fig. 21 (RCD). Sr-Stream's real-time load imbalances are remarkably lower than KG, PKG and FlexSP, and slightly lower than FISH. Initially, the load imbalance of KG and PKG increased due to data skew. Due to key statistics do not accurately reflect the true distribution, FlexSP exhibited relatively high load imbalance. As the system gradually collects more accurate workload and tuples are routed to the least-loaded instance, the load imbalance reduced. As the system operates, Sr-Stream's real-time system imbalance eventually stabilizes at around 0.21 for WC, and converges to around 0.27 for RCD. Compared to baselines, Sr-Stream reduces the real-time imbalance by 6.2–77.04% for WC, and 11.50–71.26% for RCD.

Fig. 22 (WC) and Fig. 23 (RCD) show the average load imbalance with adjusted parallelism levels. When parallelism level is 8, the aver-

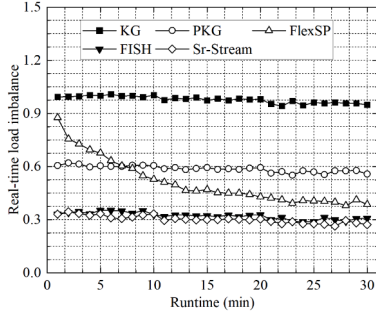


Fig. 21. Real-time load imbalance of RCD under stable input rate and parallelism level.

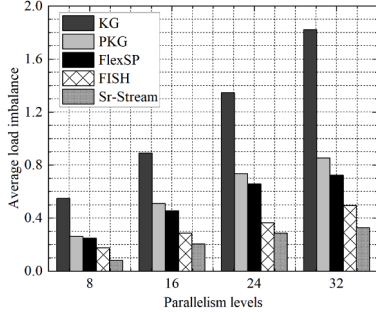


Fig. 22. Average load imbalance of WC under different parallelism levels.

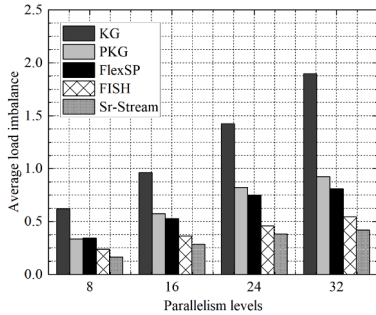
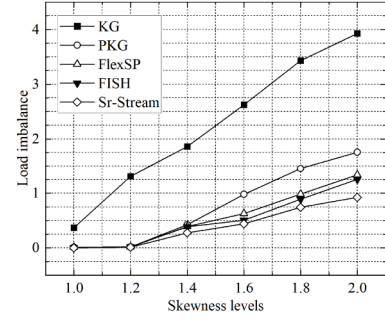


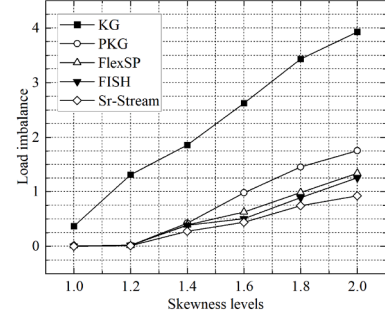
Fig. 23. Average load imbalance of RCD under different parallelism levels.

age load imbalance of Sr-Stream is 0.081 for WC and 0.164 for RCD. Compared to baselines, Sr-Stream reduces the average load imbalance by 53.98–85.21% for WC and 31.61–72.85% for RCD. As parallelism increases, the load imbalance of all methods rises, with KG exhibiting the largest increase. In contrast, Sr-Stream achieves the smallest increase. When the parallelism level reaches to 32, the average load imbalance of Sr-Stream is 0.327 for WC and 0.421 for RCD. Among the baselines, the lowest load imbalance (FISH) is 0.495 for WC and 0.546 for RCD. Compared to baselines, Sr-Stream reduces the average load imbalance at least 33.94% for WC, and at least 22.89% for RCD.

Fig. 24 (WC) and Fig. 25 (RCD) show the load imbalance comparison for changed parallelism level and data stream skewness. Both the parallelism level is 8 and 32, all methods exhibit increasing load imbalance as the skewness level grows. Among them, KG exhibits the largest increase, reaching 3.927 and 3.988 for WC and RCD, respectively (parallelism level is 8). As the parallelism level changed to 32, the load imbalance of all methods is exacerbated. However, Sr-Stream maintains a relatively low level of load imbalance compared to the baselines. Even at a parallelism level of 32, Sr-Stream maintains load imbalance values of 2.248 for WC and 2.536 for RCD at a skewness of 2.0. Compared to baselines, Sr-Stream reduces the load imbalance by up to 99.46% (compared to KG, at parallelism level is 8 and skewness level is 1.0 for WC), and at



(a) Parallelism of 8



(b) Parallelism of 32

Fig. 24. Load imbalance of WC under different skewness levels: (a) at parallelism of 8; (b) at parallelism of 32.

least 8.34% (compared to FISH, at parallelism level is 8 and skewness level is 1.2 for RCD).

The experiment proves that the system load imbalance is closely related to the degree of parallelism of the system. When parallelism level increases, the load imbalance resulting from data skew becomes more pronounced. Within a certain range of load balance, Sr-Stream can effectively mitigate the load imbalance problem resulting from data skew, and is noticeably better than baselines.

6. Related work

In this section, we review three categories of related works: state management, data stream grouping and load balance.

6.1. State management

In the stream computing topology, nodes are divided into stateful nodes and stateless nodes according to whether there is a dependency between the current processing result and the previous processing result [22,23].

Many distributed stream processing systems do not support state management, such as Heron and early versions of Storm. The later versions of Storm use Trident which leverages hash tables to store state, but with certain limitations. In some systems, the state is separated from the application logic and centralized in a remote database [24], periodically checking state to achieve fault tolerance. Other systems combine “soft state” stored in memory data structures with “hard state” stored in disk data for state management, such as Flink and Kafka. Therefore, state management is a very important and non-trivial work in distributed stream calculation.

Distributed stream processing systems are often dynamically scaled or imbalanced due to data fluctuations, system failures, etc. For stateful nodes, when parallel tasks change, state transition is required [5]. Initially, distributed stream processing systems paused entire system operations during state migration to maintain state consistency. Although

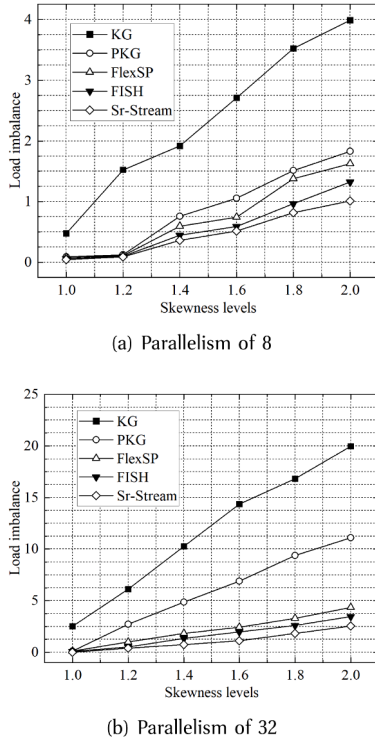


Fig. 25. Load imbalance of RCD under different skewness levels:(a) at parallelism of 8;(b) at parallelism of 32.

this scheme is very simple, it creates a latency peak during the migration. Therefore, Flux only suspends node instances that require state migration to reduce latency peaks. However, if the node is a critical component in the topology, this operation is also equivalent to pausing the entire topology.

There are two most commonly used grouping strategies in distributed stream processing systems, namely random grouping (SG) and key grouping (KG). For stateful nodes, SG can effectively achieve data-level load balance, but the state of all key values needs to be saved in each downstream instance. The number of key values in the data stream is enormous, resulting in high memory cost and limited scalability. KG can store the state according to the key value, and when the system changes, the number of states that needs to be processed is small, and only a part of the state needs to be processed on the old instance. However, when the data skew is large, key grouping distributes all the data of high-frequency key values to the same instance for processing. As a result, the load of a single downstream instance is much higher than that of other instances, which greatly affects the system performance.

6.2. Data stream grouping

In distributed stream computing, data grouping usually occurs at any stage of topology execution. When the previous stage instance has processed a tuple, it needs to send the output to the next stage instance through a grouping strategy. Currently, there is a wide range of research in distributed stream processing systems to address the load balance problem in parallel instances.

In order to solve the workload skew caused by data stream distribution, grouping function (GF) [25] proposes that using stateful data parallelism to improve throughput. However, when the content of the high-frequency key value changes rapidly, the grouping function will also need to be updated. Frequent migration can incur significant overhead if updates are made. If not updated, the system performance will be affected.

DKG [31] proposed a distribution-aware Key Grouping that is calculated for input streams with skewed Distribution characteristics. However, the learning algorithm is offline and cannot adapt to the changing data stream.

The hybrid grouping strategy(HGS) [26] based on key load division adopts the data distribution method combining hash function and routing table. Two grouping algorithms, ELB and CLB [27], based on the two optimization objectives of load balance of stateful nodes, are proposed to minimize load imbalance and minimize state migration overhead respectively.

The load imbalance problem arising from data skew is mitigated by Partial Key Grouping (PKG) [6] which employs two techniques: key grouping and local load estimation. This allows for adaptation to data skew using the classic “power of two choices”. Recognizing that only two candidate instances for some high-frequency key values are not enough to solve the skew problem, D-Choices and W-Choices [15] introduced a grouping scheme based on key frequency.

To solve the unreliability of local load estimation in PKG, BPG [29] proposed a Back Propagation Grouping strategy. To address the problem of hotkeys changing over time, FISH [17] proposed a new load grouping mechanism.

Subsequently, PStream [21] proposed a pop-aware differentiated distributed stream processing system, which uses random grouping to assign hot keys and hash functions to assign uncommon key values.

PFG [30] proposed a sketch-based pre-filtered grouping algorithm that utilizes a heavy-hit algorithm to monitor items in the data stream. Although PFG splits the high-frequency key values, it only considers the local load when the low-frequency key is distributed, and does not consider the load change when the high-frequency key value is distributed. In addition, pre-filtering algorithm of PFG cannot accurately predict the change of key frequency in real-time, and the identification of high-frequency key value is biased.

The local sensing grouping algorithm mainly improves the system performance according to the network communication time between upstream and downstream instances in the state stream. Squirrel [28] proposed a network-aware grouping framework to enhance system performance at different network distances. While Squirrel effectively reduces latency and improves system throughput compared with the basic grouping algorithm, for key grouping, the key values of high-load instances are randomly distributed to other instances, which does not meet the requirements of state storage in stateful data stream. A comparative summary of our work with other closely related studies is presented in Table 3.

6.3. Load balance

To optimize the data processing capacity and resource utilization, the distributed system tries its best to divide the workload into multiple parallel processing units for processing, so as to complete the work task cooperatively. However, due to various complex scenarios and data processing, load balance has become a major challenge for distributed systems.

The existing load balance algorithms are mainly divided into static and dynamic types. Static load balance algorithms assign tasks in a fixed way, regardless of the state of different nodes in the system. This method is easy to implement and has low time complexity and high task efficiency. Common algorithms include polling method [32], random method [33], hash method [34] and weight method [35], etc. Although polling method and random method equally allocate the number of tasks, they are only applicable to nodes with the same performance. If the performance of a node is poor, the bucket effect may occur, causing resource waste on other nodes and system faults. The weighting method assigns the weight based on the initial state of the nodes in the system, and then select the assignment task with the greatest weight, such as weighted polling and weighted random. However, these methods can not adapt well to the load changes caused by the data fluctuation and

Table 3
Comparison of Sr-Stream and related work.

Method	Capability				
	State Management	Dynamic Adaptation	Load Balance	Network Latency	Cost Saving
GF [25]	×	×	✓	×	×
HGS [26]	×	✓	✓	×	×
ELB [27]	×	✓	✓	×	×
PKG [6]	×	✓	✓	×	×
D-Choices [15]	×	✓	✓	×	×
Squirrel [28]	×	✓	×	✓	✓
BPG [29]	×	×	✓	×	×
FISH [17]	×	✓	✓	×	×
PStream [21]	×	✓	✓	×	×
PFG [30]	×	✓	✓	×	×
Sr-Stream (ours)	✓	✓	✓	✓	✓

the difference of node processing capacity in the distributed stream computing system, and can not meet the requirements of load balance.

The dynamic load balance algorithms distribute tasks according to the real-time state of the system runtime. Common dynamic load balance algorithms include the least connection method [36], the shortest response time [37], and the dynamic weight method [38]. The least connected method sends new tasks to the least connected node, it considers the load of the current node, and monitors the total number of concurrent task connections. The shortest response time only takes into account the response time of the processing node. When the system is running, the task is preferentially sent to the node that provides the fastest response time, but the response time must be measured continuously with the time. The dynamic weight method mainly assigns a weight to each node according to the real-time data of the node's current load and capacity in real time, and the weight is constantly changing.

Therefore, in contrast with the static load balance algorithm, the dynamic load balance algorithm needs to monitor the running state of the system in real-time and determine task allocation according to the load changes of each node. Although the algorithm is more complex, it is more suitable for distributed stream computing system to deal with dynamic data stream.

In addition, in a distributed stream computing system, the workload of instances is dynamically adjusted from multiple perspectives, such as resource scheduling [39–41], parallelism scaling [42–45] and load migration. Resource tuning is used to allocate appropriate resources for computing tasks while the system is running. When the computing task load is high, more resources are allocated. Parallelism scaling means that when the system is running, the appropriate degree of parallelism is set for each stage of the computing task. If the performance of a computing task in a certain phase does not meet the requirements, increase the parallelism of the computing task in this phase. Load migration is a migration policy. When the load of a computing task is too heavy, the load of the computing task is transferred to other nodes with lighter load in the system according to a certain policy.

7. Conclusions and future work

In this paper, we proposed Sr-Stream, a skewed stream rebalance strategy for elastic stateful stream processing systems. Sr-Stream integrates online key frequency statistics, dynamic key splitting, and latency-aware instance assignment to mitigate data skew and improve routing decisions under distributed deployment. Specifically, it uses a two-level frequency summary structure to track key frequency dynamics, adjusts the splitting degree of keys according to their observed frequencies and load imbalance, and further incorporates network latency and instance processing capability into the routing process. It has been confirmed that Sr-Stream performs well in latency, throughput, and system load imbalance compared with other rebalance algorithms, and has

excellent elasticity and stability, it significantly improves the overall system performance.

In future work, we will explore how to ensure a minimum load imbalance without splitting key values while deploying the entire topology on a cluster. The state management overhead caused by the reduction or increase of candidate instances of a key value is also a consideration. Additionally, we will further strengthen the theoretical analysis of Sr-Stream by deriving formal bounds for key performance metrics such as load imbalance and tail latency.

CRediT authorship contribution statement

Dawei Sun: Validation, Methodology, Funding acquisition, Conceptualization; **Xinyi Chen:** Writing – review & editing, Validation, Methodology, Data curation; **Si Lei:** Writing – original draft, Methodology, Investigation; **Jie Wen:** Investigation, Formal analysis, Data curation; **Atul Sajjanhar:** Writing – review & editing, Investigation, Formal analysis; **Rajkumar Buyya:** Writing – review & editing, Methodology, 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.

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](#) under Grant No. 265QZ2021001.

This paper is a substantial extension of a short paper [46] presented at ADMA 2023.

Data availability

Data will be made available on request.

References

- [1] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, V. Markl, Benchmarking distributed stream data processing systems, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, 2018, pp. 1507–1518.
- [2] Ecker, Raphael and Karagiannis, Vasileios and Sober, Michael and Schulte, Stefan, Latency-aware placement of stream processing operators in modern-day stream processing frameworks, *J. Parallel Distrib. Comput.* 199 (2025) 105041.
- [3] J. Tan, Z. Tang, W. Cai, W. Tan, X. Xiao, J. Zhang, Y. Gao, K. Li, A cost-aware operator migration approach for distributed stream processing system, *IEEE Trans. Cloud Comput.* 13 (1) (2025) 441–454.
- [4] W. Ullah, A. Ullah, T. Hussain, K. Muhammad, A.A. Heidari, J. Del Ser, S.W. Baik, V.H.C. De Albuquerque, Artificial Intelligence of Things-assisted two-stream neural network for anomaly detection in surveillance big video data, *Futur. Gener. Comput. Syst.* 129 (2022) 286–297.

- [5] R. Gu, H. Yin, W. Zhong, C. Yuan, Y. Huang, Mecas: latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems, in: 2022 USENIX Annual Technical Conference (USENIX ATC 22), 2022, pp. 539–556.
- [6] M.A.U. Nasir, M.G. De Francisci, D. Garcia-Soriano, N. Kourtellis, M. Serafini, The power of both choices: practical load balancing for distributed stream processing engines, in: 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 137–148.
- [7] H. Jin, F. Chen, S. Wu, Y. Yao, Z. Liu, L. Gu, Y. Zhou, Towards low-latency batched stream processing by pre-scheduling, IEEE Trans. Parallel Distrib. Syst. 30 (3) (2018) 710–722.
- [8] J. Huang, R. Li, X. Jiao, Y. Jiang, W. Chang, Dynamic dag scheduling on multiprocessor systems: reliability, energy, and makespan, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (11) (2020) 3336–3347.
- [9] T.Z. Fu, J. Ding, R.T. Ma, M. Winslett, Y. Yang, Z. Zhang, DRS: dynamic resource scheduling for real-time analytics over fast streams, in: 2015 IEEE 35th International Conference on Distributed Computing Systems, 2015, pp. 411–420.
- [10] 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: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 783–798.
- [11] J. Fang, P. Chao, R. Zhang, X. Zhou, Integrating workload balancing and fault tolerance in distributed stream processing system, World Wide Web 22 (2019) 2471–2496.
- [12] S.L. Mirtaheeri, L. Grandinetti, Dynamic load balancing in distributed exascale computing systems, Cluster Comput. 20 (2017) 3677–3689.
- [13] T. Toliopoulos, A. Gounaris, Adaptive distributed partitioning in apache flink, in: 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW), 2020, pp. 127–132.
- [14] Ristache, Dragos-Florian and Onak, Krzysztof, The adaptive use of count-min sketch: what is safe and what is not?, in: Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2, 2025, pp. 2432–2443.
- [15] M.A.U. Nasir, M.G. De Francisci, N. Kourtellis, M. Serafini, When two choices are not enough: balancing at scale in distributed stream processing, in: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), 2016, pp. 589–600.
- [16] S. Chen, D. Zuo, Z. Zhang, FlexSP: $(1 + \beta)$ -choice based flexible stream partitioning for stateful operators, in: Proceedings of the 53rd International Conference on Parallel Processing, 2024, pp. 732–741.
- [17] X. Liao, Y. Huang, L. Zheng, H. Jin, Efficient time-evolving stream processing at scale, IEEE Trans. Parallel Distrib. Syst. 30 (10) (2019) 2165–2178.
- [18] D. Sun, H. Yan, S. Gao, Z. Zhou, Performance evaluation and analysis of multiple scenarios of big data stream computing on storm platform 12 (7) (2018) 2977–2997.
- [19] Amazon review dataset, 2026, <https://github.com/topics/amazon-review-dataset>.
- [20] Didi Chuxing GAIA initiative, 2023, <https://gaia.didichuxing.com>.
- [21] H. Chen, F. Zhang, H. Jin, Pstream: a popularity-aware differentiated distributed stream processing system, IEEE Trans. Comput. 70 (10) (2020) 1582–1597.
- [22] Q. To, J. Soto, V. Markl, A survey of state management in big data processing systems, VLDB J. 27 (6) (2018) 847–872.
- [23] M. Mudassar, Y. Zhai, L. Liao, Efficient state management for scaling out stateful operators in stream processing systems, Big Data 7 (3) (2019) 192–206.
- [24] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom, Stream: the stanford data stream management system, Data Stream Manag. (2016) 317–336.
- [25] B. Gedik, Partitioning functions for stateful data parallelism in stream processing, VLDB J. 23 (4) (2014) 517–539.
- [26] J. Fang, R. Zhang, T.Z. Fu, Z. Zhang, A. Zhou, J. Zhu, Parallel stream processing against workload skewness and variance, in: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, 2017, pp. 15–26.
- [27] Q. Guo, Y. Zhou, Stateful load balancing for parallel stream processing, in: Euro-Par 2017: parallel Processing Workshops: Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28–29, 2017, Revised Selected Papers 23, 2018, pp. 80–93.
- [28] F. Chen, S. Wu, H. Jin, Network-aware grouping in distributed stream processing systems, in: Algorithms and Architectures for Parallel Processing: 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15–17, 2018, Proceedings, Part I 18, 2018, pp. 3–18.
- [29] X. Zhang, H. Chen, F. Hu, Back propagation grouping: load balancing at global scale when sources are skewed, in: 2017 IEEE International Conference on Services Computing (SCC), 2017, pp. 426–433.
- [30] A. Aslam, H. Chen, H. Jin, Pre-filtering based summarization for data partitioning in distributed stream processing, Concurrency Comput. Pract. Exper. 33 (20) (2021) e6338.
- [31] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, B. Sericola, Efficient key grouping for near-optimal load balancing in stream processing systems, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, 2015, pp. 80–91.
- [32] G. Liu, X. Wang, A modified round-robin load balancing algorithm based on content of request, in: 2018 5th International Conference on Information Science and Control Engineering (ICISCE), 2018, pp. 66–72.
- [33] V. Ariharan, S.S. Manakattu, Neighbour aware random sampling (NARS) algorithm for load balancing in cloud computing, in: 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), 2015, pp. 1–5.
- [34] J. Li, Y. Nie, S. Zhou, A dynamic load balancing algorithm based on consistent hash, in: 2018 2nd IEEE Advanced Information Management, Communications, Electronic and Automation Control Conference (IMCEC), 2018, pp. 2387–2391.
- [35] E.J. Ghomi, A.M. Rahmani, N.N. Qader, Load-balancing algorithms in cloud computing: a survey, J. Netw. Comput. Appl. 88 (2017) 50–71.
- [36] L. Zhu, J. Cui, G. Xiong, Improved dynamic load balancing algorithm based on least-connection scheduling, in: 2018 IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC), 2018, pp. 1858–1862.
- [37] O.H. Jader, S. Zeebaree, R.R. Zebari, A state of art survey for web server performance measurement and load balancing mechanisms, Int. J. Sci. Technol. Res. 8 (12) (2019) 535–543.
- [38] W. Wang, G. Casale, Evaluating weighted round robin load balancing for cloud web services, in: 2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2014, pp. 393–400.
- [39] X. Wei, L. Li, X. Li, X. Wang, S. Gao, H. Li, Pec: proactive elastic collaborative resource scheduling in data stream processing, IEEE Trans. Parallel Distrib. Syst. 30 (7) (2019) 1628–1642.
- [40] W. Daniel, A. Luciana, S. Pierre, H. Nicolás, PA-SPS: a predictive adaptive approach for an elastic stream processing system, J. Parallel Distrib. Comput. 192 (2024) 104940.
- [41] T. Chang, L. Li, M. Wu, W. Yu, X. Wang, C. Xu, PAGroup: privacy-aware grouping framework for high-performance federated learning, J. Parallel Distrib. Comput. 175 (2023) 37–50.
- [42] T.Z. Fu, J. Ding, R.T. Ma, M. Winslett, Y. Yang, Z. Zhang, DRS: auto-scaling for real-time stream analytics, IEEE/ACM Trans. Network. 25 (6) (2017) 3338–3352.
- [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: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 783–798.
- [44] L. Cheng, S. Kotoulas, T.E. Ward, G. Theodoropoulos, Improving the robustness and performance of parallel joins over distributed systems, J. Parallel Distrib. Comput. 109 (2017) 310–323.
- [45] J. Li, C. Zhang, J. Zhang, X. Qin, L. Hu, MiCS-P: parallel mutual-information computation of big categorical data on spark, J. Parallel Distrib. Comput. 161 (2022) 118–129.
- [46] S. Lei, D. Sun, A. Sajjanhar, An elastic scalable grouping for stateful operators in stream computing systems, in: Proc. the 19th Anniversary of the International Conference on Advanced Data Mining and Applications, ADMA 2023, Shenyang, China, Springer Press, Aug. 21–23, 2023, pp. 463–478.



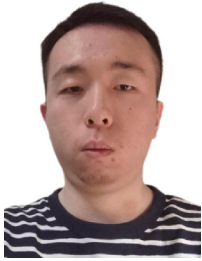
Dawei Sun is a Professor in the School of Artificial Intelligence, China University of Geosciences, Beijing, P.R. China. He received his Ph.D. degree in computer science from North-eastern University, China in 2012, then conducted the Post-doctoral research in the department of computer science and technology 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 100 journal and conference papers.



Xinyi Chen is a postgraduate student at the School of Artificial Intelligence, China University of Geosciences, Beijing, China. She received her Bachelor Degree in Zhejiang University Of Finance Economics in 2023. Her research interests include big data stream computing and distributed systems.



Si Lei received the B.E. and M.E degree in Computer Science and Technology from the China University of Geosciences, Beijing, China in 2020 and 2023, respectively. Her research interests include big data stream computing and distributed systems.



Jie Wen is a postgraduate student at the School of Artificial Intelligence, China University of Geosciences, Beijing, China. He received his Bachelor Degree in Electrical Engineering and Automation from Anhui Polytechnic University in 2021. His research interests include big data stream computing, data analytics and distributed systems.



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 textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h index 165 with 145300+ 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.



Atul Sajjanhar is a Senior Lecturer in the School of Information Technology, Deakin University, Australia. He obtained Master of Computing and Ph.D. degrees from Monash University, Australia. He has held positions in the Institute of Infocomm Research, Singapore and The University of Southern Queensland, Australia. His research areas of interest are in machine learning and pattern recognition. He has participated in several industry funded projects in Australia.