The Journal of Supercomputing (2025) 81:722
https://doi.org/10.1007/s11227-025-07216-8

An elastic reconfiguration strategy for operators in distributed stream computing systems

Dawei Sun[1] · Yinuo Fan[1] · Chengjun Guan[1] · Jia Rong[2] · Shang Gao[3] · Rajkumar Buyya[4]

Accepted: 17 March 2025

Abstract

Low latency and high throughput are crucial for distributed stream computing systems. Existing operator reconfiguration strategies often have poor performance under resource-limited and latency-constraint scenarios. The challenge lies in the elasticity of operator parallelism and reconfiguration of operators that balances performance constraints and performance improvement. To address these issues, we propose Er-Stream, an elastic reconfiguration strategy for various application scenarios. This paper discusses the Er-Stream from the following aspects: (1) We model task topology as a queuing network to evaluate system latency, and construct a communication cost model to formalize the reconfiguration problem; (2) we proposed an elastic strategy for operator parallelism to rationally utilize the available resources and reduce the processing latency of topology; (3) we proposed a reconfiguration strategy for operators to reduce the communication cost, and set thresholds added to control its trigger frequency; (4) we design and implement Er-Stream and integrated it into Apache Storm. We evaluate key metrics such as latency, throughput, resource usage, and CPU utilization in a real-world distributed stream computing environment. Results demonstrate significant improvements achieved by Er-Stream. In comparison with Storm's existing strategies, it reduces average system latency by up to 30%, increases average system throughput by 1.89 times, lowers average resource usage by 26.6%, and increases CPU utilization by 19.8%.

Keywords Operator parallelism · Elastic strategy · Operator reconfiguration · Distributed stream computing · Storm

Extended author information available on the last page of the article

Published online: 10 April 2025

1 Introduction

1.1 Background and motivation

Distributed stream computing (DSC) systems play an important role in the era of big data and are applied in a wide range of domains such as the Internet of Things (IoT) [1], environmental inspection [2], and fraud detection [3]. As the booming development of DSC systems, various stream computing frameworks have emerged, such as Apache Storm [4], Apache Samza [5] Apache Flink [6], Twitter Heron [7], and Apache Spark streaming [8]. Different application scenarios generate varied data processing requirements. Some application scenarios prioritize resource utilization during transaction processing, aiming to process transactions with minimal resources, such as internal log processing systems within companies. Conversely, other scenarios demand stringent system latency to ensure real-time performance even at the cost of higher resource consumption. These scenarios include urban traffic detection, real-time recommendation, and business monitoring [9]. Apache Storm, as a mainstream computing framework, has been widely used in various domains in recent years [10]. Although existing DSC frameworks perform well in many application scenarios, there is ongoing room for improvement, particularly in optimizing operator parallelism and configuration.

High throughput and low latency are two crucial metrics to evaluate DSC systems [11]. Achieving low average resource usage and high CPU utilization are primary objectives in designing and developing a DSC framework. Before being processed by a DSC system, a stream application is modeled as a directed acyclic graph (DAGs) [12]. The DAG outlines the sub-tasks with data dependencies, forming what is known as the task topology. Vertices in a DAG denote computations (operators) [13], while edges denote communication (data dependencies) between vertices. Once the modeling phase is complete, each DAG is submitted to the DSC platform and scheduled to run on one or more computing nodes. In DSC systems, existing scheduling strategies usually rely on users configuring the parallelism of operators for the DAGs. However, user-defined parallelism may not be optimal for the current operators, potentially resulting in prolonged tuple processing latency and reduced throughput. In addition, improper configuration may lead to higher average resource usage and lower CPU utilization during data processing.

Within the task topology, an operator can run one or more instances, with the number of instances representing the operator's parallelism. Dynamically determining this parallelism for each operator is crucial for enhancing performance [14, 15]. In recent years, researchers have attempted to adjust operator parallelism based on various parameters such as input and output rates of tuples [16]. Processing latency is a crucial parameter when evaluating DSC systems or frameworks. Lengthy processing latency degrades the system's timeliness, significantly hampering real-time response. Since each operator's parallelism may vary, their impact on processing latency also differs. Therefore, identifying operators requiring parallelism adjustment becomes vital. A practical approach involves prioritizing operators for parallelism adjustment by assessing each operator's impact on average processing latency within the task topology.

As data transmission occurs between upstream and downstream operators, termed communication, communication cost is inevitable within the system. Usually, the topology tasks submitted to stream computing system are evenly distributed across nodes based solely on the total number of instances these tasks have. This approach overlooks communication, performance, and load disparities between nodes, making it difficult to fully utilize computing resources and compromising overall system performance during operation. In the default configuration, instances run continuously unless explicitly terminated by the user. If the user wants to change task configurations, they must first terminate the execution and then reconfigure it, leading to prolonged latency due to the termination and restart of execution. Studies have shown that the operator placement and scheduling problem in DSC systems is NP-hard [17]. Online scheduling strategies [18–21] serve as a mainstream method for addressing these issues. Among them, Aniello et al. [20] demonstrate awesome performance as a popular online scheduling strategy. However, online scheduling strategies still exhibit certain drawbacks, notably the potential for resource overutilization or underutilization, which can degrade the performance and response time of stream computing systems.

The scheduling of instances is closely related to the configuration of operators. In our previous work [35], we focused on optimizing the scheduling strategy of DSC systems, which significantly improved latency. Building on this foundation, we expand our research to address a critical issue in DSC systems: operator reconfiguration. Our primary focus is on how to flexibly reconfigure operators to further improve system performance.

In summary, existing reconfiguration strategies can be improved in the following aspects: (1) employing an elastic strategy for operator parallelism, (2) adapting the system to function optimally in various application scenarios like resource-limited or latency-constraint scenarios, and (3) implementing a proper reconfiguration strategy for operators. To address the aforementioned issues, we investigate an elastic reconfiguration strategy aimed at improving the performance of DSC systems within resource-limited and latency-constraint application scenarios. Our experiments demonstrate that employing this strategy leads to improvements across various metrics including processing latency, CPU average utilization, throughput, and overall resource usage of the system.

1.2 Contributions

Our contributions can be summarized as follows:

1. Modeling the task topology as a queuing network to evaluate system latency, and constructing a communication cost model to formalize the scheduling problem.

2. Introducing an elastic strategy for operator parallelism aimed at reducing processing latency and increasing resource utilization in resource-limited and latency-constraint scenarios.
3. Proposing a configuration strategy for operators to minimize communication cost and implementing two thresholds to prevent node overload, determining the execution of reconfiguration strategies for cost reduction.
4. Designing and implementing the system model of Er-Stream and integrating it into Apache Storm, a mainstream DSC system.

1.3 Paper organization

The rest of this paper is organized as follows. In Sect. 2, we describe the models of streaming application, resource limitation, latency constraint, and communication cost. In Sect. 3, we focus on Er-Stream and explain its system architecture and implementation, the elastic strategy for parallelism of operators, and the algorithm for operator reconfiguration based on the communication cost model. In Sect. 4, we detail the performance evaluation of Er-Stream. In Sect. 5, we introduce the related work on operator parallelism and operator reconfiguration. Finally, in Sect. 6, we conclude the paper and discuss our future work.

2 Problem statement

Before formalizing the problem of reconfiguration and introducing our solution, we first model the streaming application, resource limitation, latency constraint, and communication cost within stream computing environments. For clarity, we summarize the primary notations used throughout the paper in Table 1.

2.1 Streaming application

In data stream processing systems, the logic of a real-time application can be modeled as a direct acyclic graph (DAG), $DAG = (V(G), E(G))$, $V(G) = \{v_1, v_2, \ldots\ldots, v_n\}$ is the set of n vertices in the DAG graph, and each vertex represents an operator called Spout or Bolt. The Spout is responsible for reading data from a data source and sends the tuple to a Bolt. The Bolt encapsulates processing logic and conducts specific data processing. Each operator can be parallelized into multiple instances such as $Vi = \{v_{i1}, v_{i2}, \ldots\ldots, v_{ik}\}$, V_{i1} represents the 1st instance of the ith operator. $E(G) = \{e_{1,2}\ e_{1,3}, \ldots\ldots, e_{n-i,n}\}$ represents the set of directed edges. $e_{1,2}$ represents the data flow from upstream vertex v_1 to downstream vertex v_2. A sample DAG graph is shown in Fig. 1.

After DAGs are submitted, the instances in an operator will be distributed to the slots of nodes. The nodes in a cluster are defined as $N = \{n_1, n_2, \ldots, n_i, \ldots, n_N\}$, where n_i represents the ith node in the cluster. The slots of each node are defined as $Slot_j = \{slot_{j,1}, slot_{j,2}, \ldots\ldots, Slot_{j,n}\}$, where $slot_{j,n}$ represents the nth slot in the jth node. In DSC systems, a node must contain at least one slot, and each slot can

Table 1 Description of primary symbols used in Er-Stream

Symbol	Description
v_n	nth vertex in a DAG
v_{ik}	kth instance of the ith vertex
n_i	ith node in the cluster
$minc_i$	Minimum resource requirement of ith operator
λ_0	Average arrival rate of tuples
L_i	Impact of ith operator on the average processing latency
$n_{\max}$	Maximum resources set by user
$T_{\max}$	Latency-constraint set by user
λ_i	Average arrival rate of ith operator input tuples
μ_i	Average processing rate of each instance in ith operator
$c_{\max}$	Maximum CPU resources
u	Maximum CPU utilization of node
E_j	jth instance running on node
$E[T_i](S)$	Average processing latency of input tuples in entire queuing network
$E_{i,j}$	Tuple transfer rate between ith and jth instances
$W_{i,j}$	Tuple transfer rate between ith and jth workers
U_{wi}	CPU usage of ith worker
U_{ni}	CPU usage of all nodes in the cluster

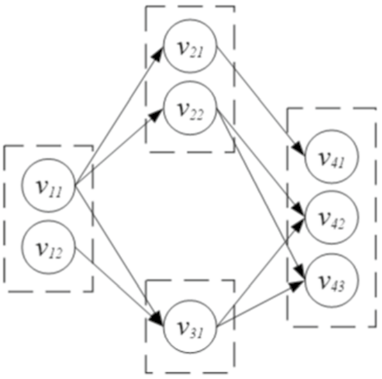


Fig. 1 A sample data flow topology

operate only one worker; however, a worker is capable of running multiple instances. As is shown in Fig. 2, the arrows represent the direction of data streams. Node n_1 contains a slot $Slot_{1,1}$, which has a worker $worker_{1,1}$, that in turn runs three instances: v_{11}, v_{12} and v_{42}.

2.2 Resource limitation

In a DSC system, there are various dimensions such as memory, I/O, and CPU [32] to measure resources on a processing node. In our test experiments, we monitored the

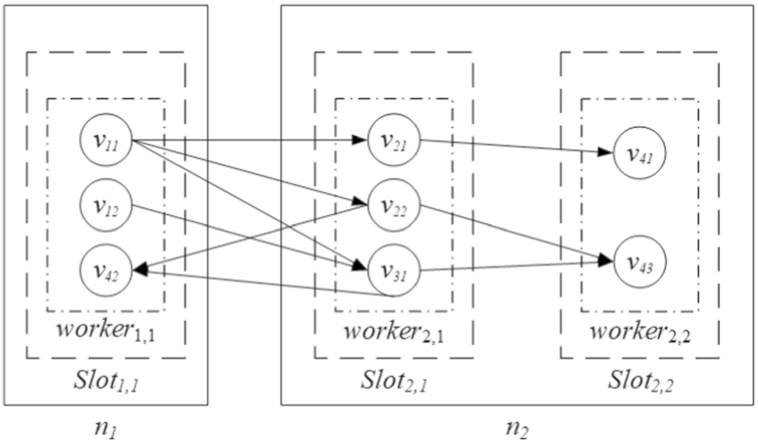


Fig. 2 Instance allocation of topology

utilization of resources such as CPU, memory, and I/O. We found that memory usage remained within a reasonable range, and I/O operations, due to features like data locality and cache optimization, did not become performance bottlenecks. Consequently, this paper focuses specifically on CPU resources.

In resource-limited scenarios, since the number of CPU resources is restricted, maximizing CPU utilization becomes crucial to enhance system performance. Before adjusting the parallelism of operators, the initial instances configuration needs completion. It is essential to ensure that each operator has the minimum required CPU resources at runtime, which is calculated by Eq. (1).

$$minci = \frac{\lambda_i}{\mu_i}, \tag{1}$$

where $minc_i$ is the minimum resource requirement of ith operator, λ_i is average arrival rate of ith operator input tuples, and μ_i is the average processing rate of each instance in ith operator. Queue-related information can be measured using the open-source code provided in [16], and information such as data stream transmission and CPU usage can be collected through Storm's built-in interfaces, IMetric, and IMetricConsumer.

Next, we calculate the cumulative $minc_i$ across all operators. The system can proceed with the subsequent adjustment of operator parallelism only if the total $minc_i$ is less than or equal to the maximum available resources. Conversely, if the total exceeds the available resources, it indicates that the existing resources fail to meet the minimum requirement for each operator, and thus, the system refrains from adjusting the parallelism of operators. This condition is represented by Eq. (2).

$$\sum_{i=1}^{n} minc_i \leq c_{max}, \tag{2}$$

where the c_{max} represents the maximum available resources. At runtime, the CPU resources available to each node cannot be ignored. The available CPU resources for each node can be denoted by $C = \{c_{n1}, c_{n2}, \ldots, c_{nN}\}$, where c_{ns} is the available CPU resource of *s*th node. For any node in the cluster, the executors on it need to satisfy Eq. (3).

$$\sum_{E_j \in ns} c_{E_j} \leq u * c_{ns}, \tag{3}$$

where u is the maximum CPU utilization of node, E_j is the *j*th instance running on node, c_{E_j} is the CPU resources consumed by E_j, ns is *s*th node in the cluster.

2.3 Latency constraint

Assume the total tuple processing time in a DAG is T_{total}, it is calculated by Eq. (4).

$$T_{\text{total}} = T_{\text{receive}} + T_{\text{queue}} + T_{\text{process}} + T_{\text{generate}}, \tag{4}$$

where T_{receive} is the time taken for the tuple to transmit from the upstream operator to the downstream operator, T_{queue} is the time that the tuple spends in a queue awaiting processing, T_{process} is the processing time of tuples by operator, and T_{generate} is the time used to generate a new tuple after the current tuple is processed.

Each operator in the topology can be approximated as a queuing system, with multiple such operators forming a queuing network. A topology containing one or more Spouts and one or more Bolts conforms to the feature of an open-loop system. Based on the above assumptions, we model the topology using open-loop Jackson queuing networks and evaluate system performance based on the average processing latency of the queuing network.

We model a topology as a Jackson network, in which each operator is regarded as a queuing system [17]. For a single operator S_i, the average processing latency of input tuples $E[T_i](S_i)$ is calculated by Eq. (5).

$$E[T_i](S_i) = E[T_i](Q_i) + \frac{1}{\mu_i}, \tag{5}$$

where μ_i is the average processing rate of all instances in *i*th operator, $E[T_i](Q_i)$ is the time for the input tuple to queue for processing in S_i (T_{queue}), while $1/\mu_i$ is the time required to process the input tuple by the executor of S_i (T_{process}). The value of $E[T_i](Q_i)$ is calculated by Eq. (6).

$$E[T_i](Q_i)\begin{cases} \frac{k(c_i m_i)^{c_i}}{ci!(1-m_i)^2 \mu_i c_i}, & m_i < 1 \\ +\infty, & m_i \geq 1 \end{cases}, \tag{6}$$

where c_i is the number of instances in S_i, m_i is the resource utilization of S_i, and k is the probability that there are no tuples in the system in the stationary state. It is obvious that when $m_i \geq 1$, the maximum processing speed of *i*th operator cannot meet the tuple arrival rate. In such cases, T_{queue} will increase infinitely over time, and each

$E[T_i](Q_i)$ is bounded when $\lambda_i < c_{i*}\mu_i$. Values for m_i and k are calculated by Eqs. (7) and (8), respectively.

$$mi = \frac{\lambda_i}{c_i \mu_i}, \tag{7}$$

$$k = \left(\sum_{l=0}^{c_i - 1} \frac{(c_i m_i)^l}{l!} + \frac{(c_i m_i)^{c_i}}{c_i!(1 - m_i)} \right)^{-1}, \tag{8}$$

The average processing latency of the entire queuing network $E[T_i](S)$ can be obtained by the weighted average of the average processing latency $E[T_i](S_i)$ of each operator, calculated by Eq. (9).

$$E[T_i](S) = \frac{1}{\lambda_0} \sum_{i=1}^{N} \lambda_i E[T_i](S_i), \tag{9}$$

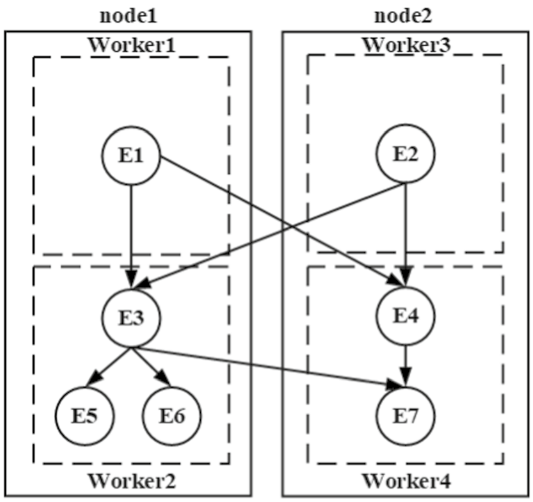
where λ_0 is the average arrival rate of tuples entering the queuing network.

After that, the latency of the system is monitored by (9). Whenever the latency exceeds the target latency-constraint set by the user, the system triggers its optimization mechanism to reconfigure the parallelism of all operators until the user-defined performance constraint is satisfied once more.

2.4 Communication cost

There are three distinct types of communication in the cluster: (1) between instances, (2) between workers, (3) and between nodes, as depicted in Fig. 3. The communication costs are significantly higher between workers and between nodes, compared to those

Fig. 3 Communication cost of topology



between instances. Therefore, reducing inter-node and inter-worker communication within the topology can significantly benefit the system's performance. To optimize the overall communication cost of the topology, it is necessary to convert inter-node and inter-worker communication into inter-instance communication to the greatest extent possible.

Let $R_{j,k}$ represent the size of the data flow between two communicating instances j and k, E is the set of instances, $l^w_{E_i}$ is the worker where instance E_i is located, and $l^n_{E_i}$ is the node where E_i is located. With these representations, the relationship between instances is described by Eq. (10).

$$\begin{cases} I_h, & l^w_{E_i} = l^w_{E_k} \\ I_m, & (l^n_{E_i} = l^n_{E_k}) \wedge (l^w_{E_i} \neq l^w_{E_k}), \\ I_s, & l^n_{E_i} = l^n_{E_k} \end{cases} \tag{10}$$

where I_h denotes high interconnection, I_m denotes moderate interconnection, and I_s denotes slight interconnection between instances. Once the topology is submitted, the total amount of data flow in the topology remains fixed without congestion, and this is calculated by Eq. (11).

$$R = \sum_{I_s} Rj,k + \sum_{I_m} Rj,k + \sum_{I_h} Rj,k, \tag{11}$$

when the total data flow between nodes $\sum_{I_s} Rj,k$ is at its minimum, and the total data flow between workers $\sum_{I_m} Rj,k$ is also minimized, the total data flow between instances $\sum_{I_h} Rj,k$ reaches its maximum. We use $CT_{m,n}$ to represent the communication cost between node m and node n, which is calculated by Eq. (12).

$$CT_{m,n} = \begin{cases} \frac{\sum Rj,k}{Bm,n} & , l^n_{E_i} \neq l^n_{E_k}, \\ 0 & , l^n_{E_i} = l^n_{E_k}, \end{cases}, \tag{12}$$

where $B_{m,n}$ represents the network bandwidth between node m and node n. If two instances are on the same node, the communication cost between them is considered negligible.

Due to the costs caused by DSC systems during resource scheduling or operator reconfiguration, which cannot be ignored [33], we introduce a performance optimization threshold referred to as Tr_{improve} to determine whether a reconfiguration should be performed. Tr_{improve} is set by the user, and reconfiguration only occurs when Tr_{change} exceeds Tr_{improve}, as described by Eq. (13).

$$Tr_{\text{change}} > Tr_{\text{improve}}, \tag{13}$$

Tr_{change} is calculated by Eq. (14)

$$Tr_{\text{change}} = \frac{Tr_{\text{old}} - Tr_{\text{new}}}{Tr_{\text{old}}}, \tag{14}$$

where Tr_{old} is the inter-node traffic before reconfiguration, Tr_{new} is the inter-node traffic after reconfiguration, and Tr_{change} is the percentage of improvement on the inter-node traffic after reconfiguration. A smaller Tr_{change} value indicates more frequent reconfiguration, resulting in higher reconfiguration cost.

Instance overloading or underloading can affect resource efficiency and system performance [34]. During reconfiguration, it is necessary to consider potential overload issue on worker node. An overloaded worker node may cause a drastic increase in the average processing time of jobs, leading to queuing of tuples until reaching the final timeout. Therefore, it is necessary to set a threshold for the CPU utilization of worker node. As per (3), if the reconfiguration outcome exceeds the CPU utilization threshold of a node, the load will be redistributed to other nodes with lighter loads.

3 Er-Stream: system architecture and implementation

Based on the discussion in Sect. 3, we propose an elastic reconfiguration strategy for operators, Er-Stream. This strategy optimizes operator parallelism and minimizes inter-node network communication costs in real time, guided by the data input rate. As a result, Er-Stream ensures high throughput, low latency, improved CPU utilization, and efficient resource usage within the system.

3.1 System architecture

To obtain real-time data generated during topology operations to support Er-Stream, we have improved the original Storm system structure. Please note that Apache Storm is merely the platform upon which our experiments are based; our strategy is equally applicable to other stream processing frameworks such as Apache Flink, Kafka Streams, and Apache Spark. This improved structure adds four custom modules: Data Monitor, Scheduler, Database, and Optimizer. Figure 4 illustrates Er-Stream's system architecture.

Data monitor is responsible for collecting information on CPU load for each thread, data flow between executors, and the average tuple arrival rate and processing rate of each operator within a specific time window.

To obtain the CPU load of each thread, the getThreadCpuTime method from the ThreadMXBean class is used and then multiplied by the CPU frequency of the working node. The data stream size transmitted between executors is determined by counting the number of tuples sent by the upstream executor received by each executor. The data stream transmission rate is computed by dividing the number of recorded tuples by the time window size.

Scheduler reconfigures executor resources by calling the pluggable interface IScheduler provided by Storm. It replaces Storm's default scheduling algorithm and executes the generated reconfiguration decisions. The IScheduler interface provides two input parameters: Topologies, which contains information about the cluster's topology, and Cluster, which contains details about the cluster's current state.

Database manages the storage of each operator's configuration information, the load information collected by the Data Monitor, and the size of the data flow

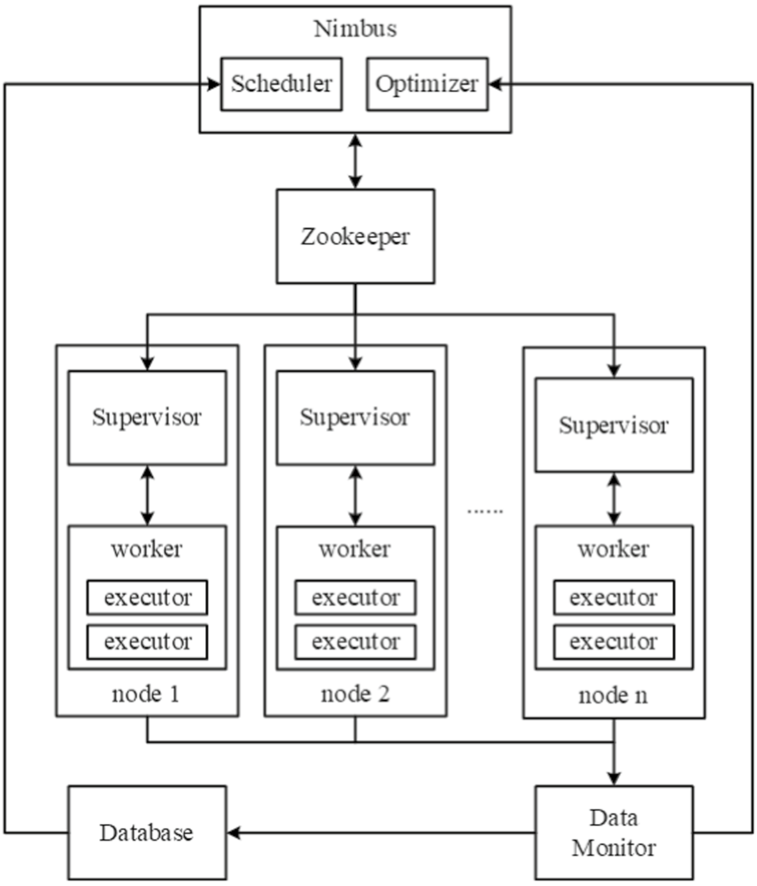


Fig. 4 Er-Stream's architecture

between executors. This stored information serves as the data basis for determining reconfiguration trigger conditions and executor assignment.

Optimizer calculates the optimal parallelism scheme for each operator based on the average tuple arrival rate and processing rate of each operator acquired form the Data monitor. Additionally, it reads parameters such as the maximum number of executor and latency constraints defined in the configuration file for resource-limited and latency-constraint scenarios.

3.2 Elastic strategy for parallelism

For different resource-limited and latency-constraint scenarios, we propose different elastic strategies for adjusting operator parallelism.

In resources-limited scenarios, our approach aims to maximize system performance by making effective utilization of available resources, with a specific focus on reducing latency. An example of how Er-Stream adjusts operator parallelism is shown in Fig. 5. The Wordcount topology contains three operators, one Spout and two Bolt. The Spout is used to distribute data, Bolt1 splits data, and Bolt2 statistics data. The initial allocation assigns minimal required resources: Spout has 2 instances, while Bolt1and Bolt2 have 3 and 1 instances, respectively.

Er-Stream processing follows these steps: (1) Use the latency-constraint model to calculate the average processing latency for input tuples across all operators. (2) Sequentially allocate resources, prioritizing operators with the greatest impact on the average processing latency of the entire topology. These two steps iterate until all resources are assigned, and each assignment of resources adds an instance to the operator. For example, Er-Stream observes that Bolt1, responsible for data splitting, lacks sufficient resources, causing delays in processing data from the Spout and resulting in high tuple processing latency. Therefore, resources are added for Bolt1 to optimize latency. Finally, the operator parallelism scheme becomes 3:7:5 from 2:3:1, effectively minimizing latency.

In latency-constraint scenarios, optimizing resource utilization while meeting latency requirements is important. Figure 6 illustrates the initial assignment of minimal required resources to operators: Spout with 2 instances, and Bolt1 and Bolt2 with 3 and 1 instances, respectively. Er-Stream's process follows these steps: (1) Use the latency-constraint model to calculate the average processing latency for input tuples across all operators. (2) Sequentially allocate resources, prioritizing operators with the greatest impact on the overall processing latency. (3) Constantly check whether the user-defined latency requirements have been met. These three steps iterate until the system latency aligns with the latency constraint, adding an instance to an operator with each resource assignment. Finally, the operator parallelism scheme adjusts to 2:5:4 from 2:3:1. Each resource allocation maximizes current processing latency, allowing the system to meet the latency constraint while utilizing minimal resources.

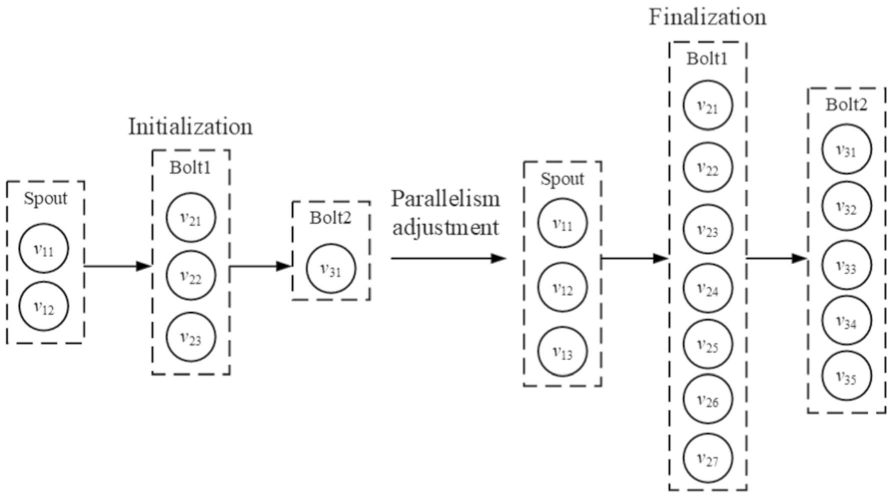


Fig. 5 An example of operator parallelism adjustment in resource-limited scenarios

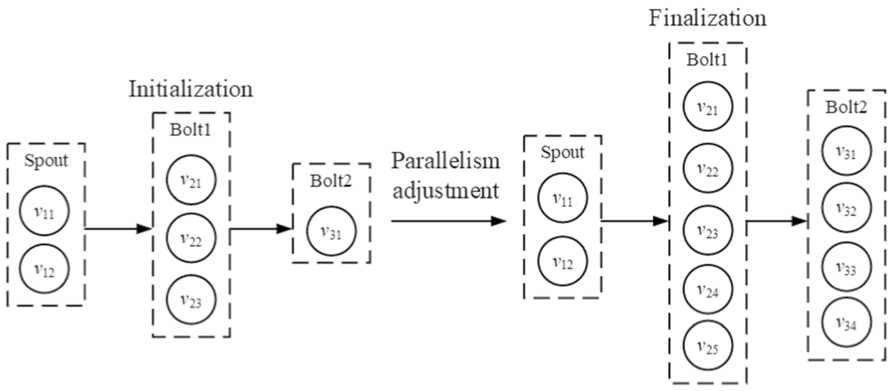


Fig. 6 An example of operator parallelism adjustment in latency-constraint scenarios

3.3 Elastic Strategy in resource-limited or latency-constraint scenarios

In resource-limited scenarios, we calculate the average arrival rate of operators' input tuples λ_i, and the average processing rate of each instance μ_i. Using the latency-constraint model, we derive the average processing latency $E[T_i](S_i)$ for each operator's input tuples. Resources are then allocated to the operator that impacts the entire topology's average processing latency the most. This process iterates until all resources are allocated. The final outcome adjusts operator parallelism to minimize the topology's average processing latency for input tuples.

In latency-constraint scenarios, we calculate the values of λ_i and μ_i, along with the average processing latency $E[T_i](S)$ of the input tuples for all operators using the latency-constraint model. Then, resources are assigned to the operator that has the greatest impact on the average processing latency of the entire topology. Hence, the latency decreases as the resources gradually increases until the latency constraint $T_{\max}$ is just met. Also, this resources assignment minimizes the number of instances, saving the system's computing resources.

The elastic strategy algorithm for operator parallelism in resource-limited or latency-constraint scenarios is described in Algorithm 1.

Algorithm 1 Elastic strategy for operator parallelism in resource-limited or latency-constraint scenarios.

```
Require: $F_{\text{new}}$: New firmware, $K_{\text{priv-Dilithium}}$: Private key, $PK_{\text{manufacturer}}$: Manufacturer's
    public key
Ensure: Secured firmware signature
1: $F^{\#}_{new} \leftarrow$ Dilithium_Sign($K_{\text{priv-Dilithium}}, F_{new}$)
2: $F_{new}$ and $F^{\#}_{new}$ are transmitted to the IoT device
3: Valid $\leftarrow$ Dilithium_Verify($F_{\text{new}}, F^{\#}_{\text{new}}, PK_{\text{manufacturer}}$)
4: if Valid then
5:     IoT device installs the firmware
6: else
7:     Reject and alert for a potential attack
8: end if
```

The algorithm's input includes statistics within a time period *T*: λ_i, μ_i and the maximum set of resources $c_{\max}$, the set latency constraint $T_{\max}$, and the type of scenario *RL*. We utilize a Boolean variable, *RL*, to represent the type of scenario. If the value of *RL* is "Ture," it indicates resource-limited scenarios; if RL equals "False," it signifies latency-constraint scenarios. Its output is the operator parallelism scheme $\{c_1, c_2 \ldots c_N\}$ for each operator in resource-limited or latency-constraint scenarios.

Steps 1 to 6 initialize the number of resources c_i assigned to all operators within the topology. They round $\frac{\lambda_i}{\mu_i}$ up and assign a value to c_i. Firstly, it satisfies the minimum operating requirement $minc_i$ for each operator, which can be calculated by (1). Then, it sums c_i to verify if it satisfies (2). If it does not satisfy, it means that the minimum resources requirement cannot be met; no subsequent assignment will be made.

Steps 8 to 15 are focus on resource-limited scenarios, iterate through all operators and record the operator that has the greatest impact on the average processing latency of the entire topology after adding an instance to each operator and mark it as *maxS*, and set the index of the operator to *k*. The impact of adding an instance to an operator on the average processing latency of the entire topology L_i can be calculated by (15).

$$L_i = \lambda_i * [E[T_i](c_i) - E[T_i](c_i + 1)] \tag{15}$$

where λ_i is the average arrival rate of operator input tuples, and $[E[T_i](c_i)\text{-}E[T_i](c_i+1)]$ is the reduction in the average tuple processing latency of the operator after adding an instance to it. If the total resources assigned to the operators do not exceed c_{max}, an instance is assigned to *max*S each time to minimize the total processing time of the entire topology.

Steps 18 to 25 are focus on latency-constraint scenarios, iterate through all operators and record the operator that has the greatest impact on the average processing latency of the entire topology after adding an instance to each operator, and mark it as *maxS*, and set the index of the operator to *k*. The impact of adding an instance to the operator on the average processing latency of entire topology's input tuple is L_i. Under the condition that the current system latency T_i is not less than T_{max}, resources are assigned to *maxS* each time to minimize the total processing time of all tuples. The time complexity of Algorithm 1 is $O(n*m)$, where *n* is the number of resources and *m* is the number of operators.

3.4 Operator reconfiguration

The difference between the default task scheduling strategy and Er-Stream is evident in the topological logic structure and the physical structure. In Fig. 7a, the default scheduling strategy of Storm uses a round robin method, evenly distributing each operator to nodes based solely on the total number of operators. However, this approach overlooks communication, performance, and load issues between nodes. For instance, if two operators with high data transmission rates are assigned to different nodes, it may lead to increased system latency.

In contrast, Er-Stream employs a different approach by first categorizing and then allocating tasks. As depicted in Fig. 7 (b), Er-Stream aims to place communicating operators on the same node whenever possible. This strategy significantly reduces communication costs between nodes and ultimately enhances system performance.

To prevent data processing timeouts caused by node overload, it is necessary to set a CPU utilization threshold for nodes. When a node's CPU utilization exceeds the threshold, it stops accepting new operators, redirecting them to nodes with lighter loads.

Following the configuration of operator parallelism in the topology, we initiate operator reconfiguration and gather data transmission volumes between two operators stored in the database. By adding up the origin and destination operators that are deployed on different nodes, we can capture overall cluster traffic. Information collected includes: tuple transfer rate $E_{i,j}$ between executor communicating with each other within period T, tuple transfer rate $W_{i,j}$ between workers, CPU usage U_{wi} of a worker, and total CPU usage U_{ni} of all nodes in the cluster. The reconfiguration algorithm is divided into two phases.

In the first phase, $E_{i,j}$ values are sorted in descending order. Sequentially, the two instances i and j with the highest communication volume in the $E_{i,j}$ sequence are selected and placed into the worker with the lowest CPU utilization, maximizing the distribution.

In the second phase, $W_{i,j}$ are sorted in descending order. Consecutively, the two Workers i and j associated with the largest communication traffic in the $W_{i,j}$ sequence are placed into the node with the lowest CPU utilization. These two phases iterate until all instances and workers are allocated.

As the algorithm pursues an optimal solution each time, the final allocation result is also the optimal solution to minimize network communication cost between

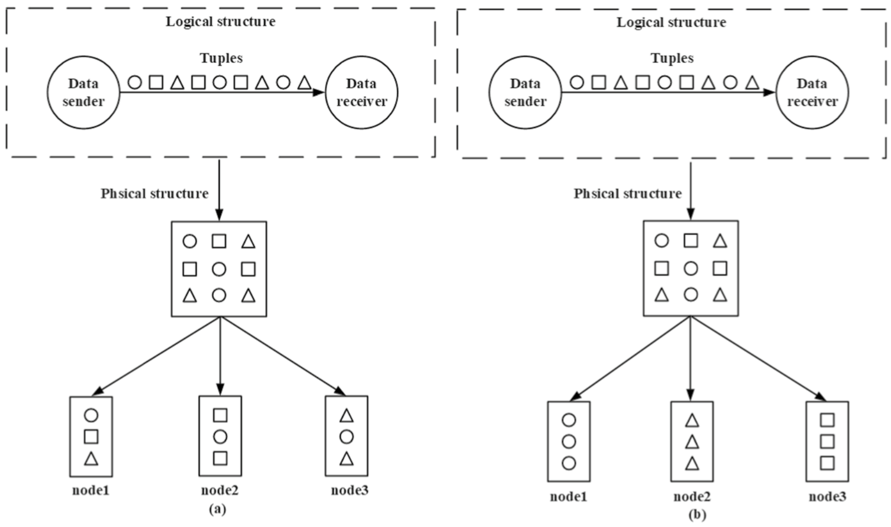


Fig. 7 Comparison between default task scheduling DefaultScheduler (**a**) and Er-Stream (**b**)

nodes. It is important to monitor potential node overload issues during the allocation process. Once a node becomes overloaded, average job processing time skyrockets, resulting in queued tuples that eventually time out. Thus, setting a threshold for worker node CPU utilization becomes necessary.

We use $N=\{n_1, n_2, ..., n_N\}$ to denotes nodes in the cluster, while $C=\{c_{n1}, c_{n2}, ..., c_{nN}\}$ denotes the available CPU resources of these nodes. With u as the maximum allowable CPU utilization for each node, it can be seen that for any $n_s \in N$, the total CPU resources occupied by instances running on the node should meet condition (3). If the allocation result exceeds the CPU utilization threshold of the node, the load will be redistributed to nodes with lower loads.

The reconfiguration strategy between nodes is described in Algorithm 2. The input consists of statistics of time period T: $W_{i,j}$ and U_{ni}. $W_{i,j}$ is the tuple transfer rate between workers communicating with each other; U_{ni} is the CPU utilization of the nodes. The output is the optimal allocation scheme of $\{n_1, n_2...n_N\}$ for $\{w_1, w_2...w_N\}$.

Algorithm 2 Reconfiguration strategy between nodes.

```
Require: $F_{\text{new}}$: New firmware, $K_{\text{priv-SPHINCS+}}$: Private key, $PK_{\text{manufacturer}}$: Public key,
    Hash function $H$
Ensure: Secure firmware update
1: Compute firmware hash: $h_{F_{\text{new}}} \leftarrow H(F_{\text{new}})$
2: Sign firmware hash with SPHINCS+: $\sigma_{\text{SPHINCS+}} \leftarrow$
   SPHINCS+_Sign($K_{\text{priv-SPHINCS+}}, h_{F_{\text{new}}}$)
3: Distribute firmware update: $F^*_{\text{new}} = (F_{\text{new}}, h_{F_{\text{new}}}, \sigma_{\text{SPHINCS+}})$
4: Verify SPHINCS+ signature: Valid $\leftarrow$ SPHINCS+_Verify($h_{F_{\text{new}}}, \sigma_{\text{SPHINCS+}}, PK_{\text{manufacturer}}$)
5: if Valid then
6:     Install firmware
7: else
8:     Abort installation (Signature invalid)
9: end if
```

Step 1 is to sort $W_{i,j}$ in descending order to obtain the sequence DW. Steps 3 to 8 describe the reconfiguration strategy when neither worker i nor worker j is allocated. Steps 9 to 26 describe the reconfiguration strategy when one of the workers is assigned. This process involves traversing all nodes to find the solution that minimizes the communication cost Int between nodes. Its time complexity is $O(n*m)$, where n is the number of worker pairs and m is the number of nodes.

The reconfiguration strategy between workers is similar to the reconfiguration strategy between nodes, although the inputs and outputs differ. Specifically, the input for the reconfiguration strategy between workers is tuple transmission rate between communicating instances $E_{i,j}$, CPU utilization of workers U_{wi}, while the output is the optimal allocation scheme $\{w_1,w_2...w_N\}$ for $\{e_1,e_2...e_N\}$. The time complexity of Algorithm 2 is $O(n*m)$, where n is the number of instance pairs and m is the number of workers.

Considering the system cost that influences system performance during operator reconfiguration, we add a performance optimization threshold to determine whether to perform reconfiguration based on the calculated reconfiguration scheme. We set the inter-node traffic before scheduling as Trold, the scheduled inter-node traffic as Tr_{new}, Tr_{change} as the percentage of traffic improvement between nodes after reconfiguration. Tr_{improve}, set by the user, triggers operator reconfiguration only if Tr_{change} reaches at least Tr_{improve}, as shown in Eqs. (12) and (13).

The workflow of Er-Stream is as follows: After the user submits the topology to the system, Data Monitor collects information such as data transmission and CPU load for each operator and then stores these data in Database. The optimizer utilizes the information gathered by Data Monitor to generate parallelism configuration schemes for operators in real-time according to Algorithm 1. Finally, the scheduler, using the information stored in the database, schedules instances based on Algorithm 2.

4 Performance evaluation

In this section, we present the cluster environment, server hardware and software parameter, and the system performance experiments conducted on Er-Stream in two application scenarios: resource limited and latency constraint. The experiment compares Er-Stream with two other algorithms: Storm's default algorithm (Storm) and the online scheduling algorithm (OnlineScheduler) proposed by [20]. According to [36], four key aspects—system latency, system throughput, average CPU utilization, and resource usage—are evaluated to assess the optimization effectiveness of Er-Stream. The analysis of the results demonstrates its optimization capabilities. In our experiments, we observed that the system stabilizes within 10 min, but to mitigate variability and ensure reliable results, a duration of 17 min was chosen for each experimental group. To ensure the reliability of the results, each set of experimental conditions was repeated more than 10 times. The average values and standard deviations of key performance metrics, including latency, throughput, and resource utilization, were calculated to provide a more reliable assessment.

4.1 Experimental environment and parameter setup

There are a total of 13 nodes configured in the computing cluster, with 1 node running Nimbus and Storm UI, 1 node running MySQL database, 1 node running ZooKeeper to coordinate the communication between the master and slave nodes, and the remaining 10 working nodes running Supervisor for business logic processing. Each compute node is powered by an Intel(R) Xeon(R) X5650 CPU. For clarity, we list the hardware configurations and the software configurations in Tables 2 and 3, respectively.

The test cases selected by Er-Stream are WordCount and Top_N. WordCount is a common test topology for counting the frequency of words in English text. It consists of one Spout and two Bolts. The task topology of WordCount is shown in Fig. 8. Spout sends data tuple, Bolt1 splits the text in data tuples according to specific regulations, and Bolt2 counts text in data tuples from Bolt 1.

Top_N is a common test topology consisting of one Spout and three Bolts for counting the top N hotspots. The task topology of Top_N is shown in Fig. 9. The Spout sends data tuples, while Bolt1 and Bolt2 are responsible for scoring, statistical analyzing, and data sorting. Bolt3 merges and sorts the data sent from the preceding operators, and ultimately outputs the top N statistics.

We use the public dataset Alibaba Tianchi [37] as the data source for applications in our experiments. The data stream rate fluctuates between 100 and 225 tuples/s, aligning with the characteristics of data streams observed in real-world scenarios.

5 Performance results of latency

5.1 In resource-limited scenarios

In DSC systems, latency refers to the time interval between when a tuple is sent by the data source and when it is fully processed. The initial operator parallelism of the WordCount topology is set to (3, 4, 8), with a maximum number of 15 available

Table 2 Hardware configurations of the cluster

Type	CPU cores (vCPU)	Memory (GB)	Bandwidth (Mbps)	Disk (GB)	Number
1	2	4	100	40	1
2	2	2	100	20	6
3	1	1	100	20	6

Table 3 Software configurations of the cluster

Software	Version
OS	Ubuntu 20.04.5 64 bit
Apache storm	Apache Storm-2.1.0
Apache zookeeper	Apache-Zookeeper-3.4.14
JDK	jdk-8u171-linux-×64
Python	Python-3.8.3
MySQL	MySQL-5.7.0

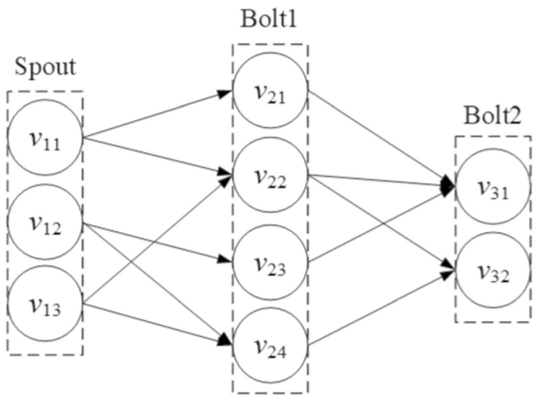


Fig. 8 Task topology of WordCount

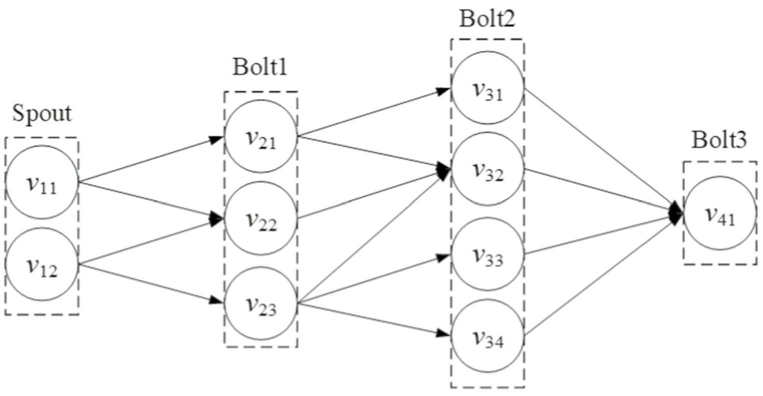


Fig. 9 Task topology of Top_N

executor resources, while the initial configuration of operator parallelism in Top_N topology is set to (5, 4, 1, 4), and the maximum number of available executors in this topology is 14. For a duration of 17 min, we test the latency of both algorithms running the same topology and recorded the results through Storm UI.

The latency comparison in resource-limited scenarios for WordCount is shown in Fig. 10. In terms of operator parallelism, both Storm and OnlineScheduler lack optimization regarding operator parallelism and always maintain the initial number of instances for each operator, while Er-Stream collects system data periodically and assigns resources to each operator for increasing parallelism. Within the first minute of system operation, due to the loading of the system profile and the enforcement of the reconfiguration strategies, both Storm and Er-Stream incur relatively high latency of 17.61 ms and 12.78 ms, respectively, while OnlineScheduler registers high latency at 23.94 ms due to real-time monitoring data collection and scheduling strategy computation. As the system runs for 5 min, the system latency gradually flattens out; Storm and OnlineScheduler hold the system latency around 17 ms and 13 ms, respectively. In contrast, Er-Stream, at the 8th minute, adjusts operator parallelism to (3, 7, 5) based on system data. This adjustment momentarily increases latency to 13.75 ms due to the need for parallelism reconfiguration. However, it rapidly decreases and stabilizes at 11.82 ms. After this adjustment, the system's latency reduces by 7.5% compared to the beginning, nearly 30% and 9.6% compared to Storm and OnlineScheduler, respectively.

The latency comparison in resource-limited scenarios of Top_N is given in Fig. 11. Storm initially incurs a relatively high latency of 27.54 ms, after which the latency eventually stabilizes around 27 ms, while the latency of OnlineScheduler stabilizes around 22 ms. Er-Stream adjusts operator parallelism to (5, 4, 3, 2) at 8th minute and allocates more resources for the latency bottleneck operator. This adjustment briefly increases the latency to 18.49 ms, followed by a rapid decrease, stabilizing around 15 ms. After the parallelism adjustment, the latency is reduced by approximately 44% and 32% compared to Storm and OnlineScheduler, respectively.

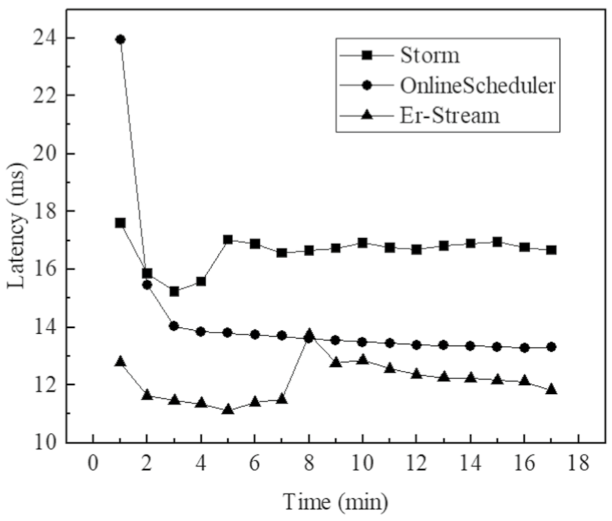


Fig. 10 Latency comparison in resource-limited scenarios for WordCount

5.2 In latency-constraint scenarios

The WordCount topology is initially configured with operator parallelism set at (2, 7, 6), and the maximum number of available executors in this topology is fixed at 15. When using Er-Stream, we set a maximum latency constraint of 20 ms.

The latency comparison in latency-constraint scenarios for Top_N is given in Fig. 12. We selected the time frame from the 17th to the 33rd minute of the

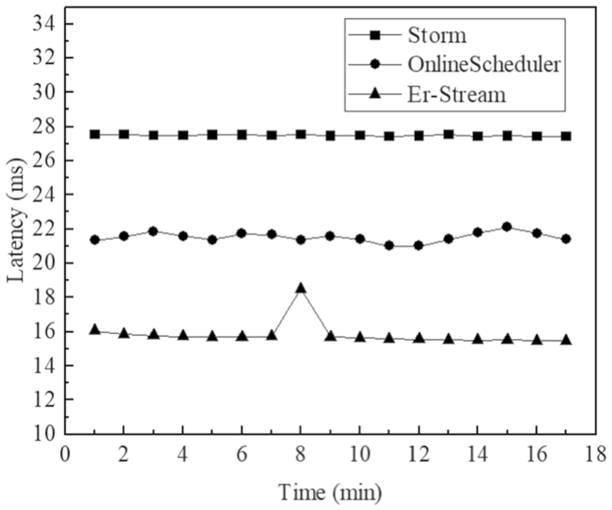


Fig. 11 Latency comparison in resource-limited scenarios for Top_N

application's runtime, as the system gradually stabilizes during this period, allowing for a more accurate reflection of the latency characteristics of the three schedulers. Storm exhibits relatively high latency of 48.79 ms due to the lack of online parallelism and network communication cost by the 17th minute. Eventually, it levels off at around 38 ms by the 27th minute. OnlineScheduler reduces latency by optimizing the network communication cost between nodes; latency stabilizes around 22 ms. In contrast, Er-Stream, with its periodic operator parallelism adjustments and reconfiguration strategies, maintains consistent latency, stabilizing at approximately 20 ms. The trends of latency for running WordCount with three schedulers are similar to those of running Top_N with the same schedulers in latency-constraint scenarios.

5.3 Performance results of throughput

5.3.1 In resource-limited scenarios

In DSC systems, throughput refers to the number of tuples processed by the system per second. We measure the throughput of the system under Storm, OnlineScheduler, and Er-Stream configurations. Extracted throughput data include measurements at the 5th, 10th, and 15th minutes.

The throughput comparison in resource-limited scenarios for WordCount is given in Fig. 13. The throughput of the system using Storm is relatively stable, and the overall average throughput is about 81 tuples/s. OnlineScheduler exhibits a lower throughput than Er-Stream, recording approximately 85 tuples/s at the 5th minute. However, after periodically scheduling, the OnlineScheduler notably enhances the system's overall throughput, reaching 139.91 tuples/s by the 10th

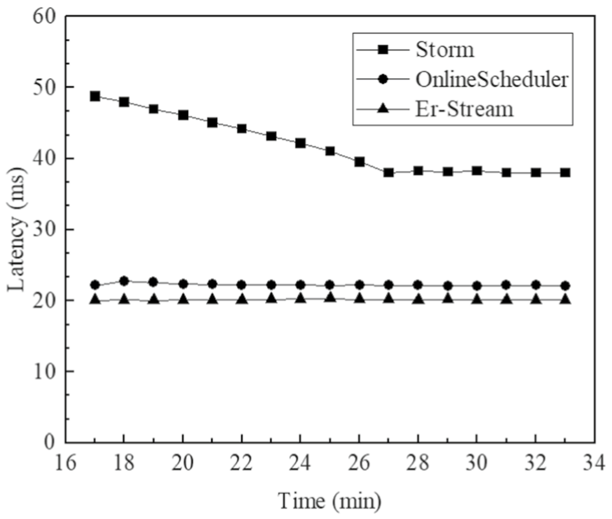


Fig. 12 Latency comparison in latency-constraint scenarios for Top_N

minute, demonstrating its effectiveness. But at the 15th minute, the throughput drastically drops to 78.18 tuples/s, revealing instability. In contrast, Er-Stream shows a steady increase in throughput over time, attributed to its strategic adjustments in operator parallelism and reconfiguration strategies. The overall average throughput under Er-Stream reaches 186.42 tuples/s. These experimental results highlight the considerable performance enhancement achieved by Er-Stream. Er-Stream consistently outperforms Storm and OnlineScheduler, with an overall average throughput approximately 1.89 times higher than OnlineScheduler. The trends of throughput for running Top_N with three schedulers are similar to those of running WordCount with the same schedulers in resource-limited scenarios.

5.3.2 In latency-constraint scenarios

The throughput comparison in latency-constraint scenarios is given in Fig. 14. Storm experiences fluctuating throughput, registering 43.63 tuples/s at the 5th minute, 54 tuples/s at the 10th minute, and 47.27 tuples/s at the 15th minute, with an average throughput of 48.31tuples/s. Similar to the Storm, the OnlineScheduler experiences fluctuations in throughput. However, owing to its optimization of inter-node communication cost, OnlineScheduler achieves a throughput of 65.36 tuples/s at the 10th minute. At 15th minute, its throughput drops to 58.18tuples/s. The overall average throughput of the system using OnlineScheduler is about 57.54 tuples/s.

Conversely, Er-Stream exhibits a substantial surge in throughput, reaching 98.18 tuples/s at the 10th minute, a notable optimization compared to others. By the 15th minute, the throughput decreases to 60 tuples/s following the trend observed in others. However, Er-Stream maintains a 26.9% and 25.2% higher throughput than Storm and OnlineScheduler, respectively. The trends of throughput for running WordCount

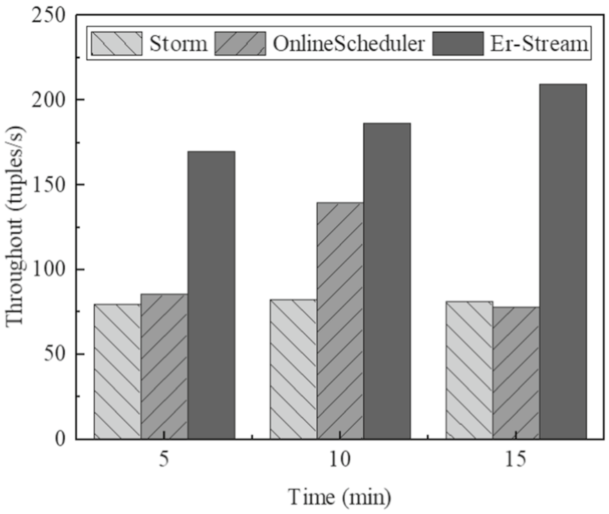


Fig. 13 Throughput comparison in resource-limited scenarios for WordCount

with three schedulers are similar to those of running Top_N with the same schedulers in latency-constraint scenarios.

5.4 Performance results of average CPU utilization

5.4.1 In resource-limited scenarios

The average CPU utilization comparison in resource-limited scenarios for WordCount is given in Fig. 15. Er-Stream registers the highest average CPU utilization among the three. Upon investigating, we find that: the Storm's task assignment approach employs polling, leading to an excessive use of nodes, resulting in their CPU utilization being limited to about 37%. This method underutilizes resources. OnlineScheduler and Er-Stream algorithms aim to assign operators to the same node, thus enhancing node CPU utilization and resource efficiency. Specifically, OnlineScheduler and Er-Stream exhibit CPU utilizations of about 46.3% and 56.7%, respectively. The experiment indicates that Er-Stream improves CPU resource efficiency.

5.4.2 In latency-constraint scenarios

The average CPU utilization comparison in latency-constraint scenarios for Top_N is given in Fig. 16. The data show that the average CPU utilization of the Storm, OnlineScheduler, and Er-Stream is about 37.3%, 48.7%, and 55.4%, respectively. The results demonstrate a relative improvement in CPU efficiency with Er-Stream.

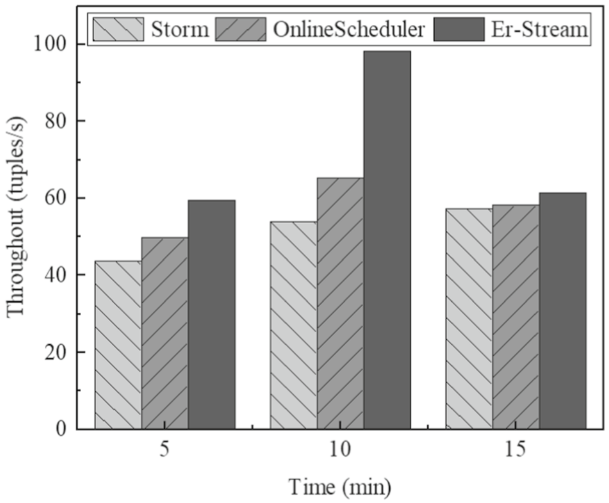


Fig. 14 Throughput comparison in latency-constraint scenarios of Top_N

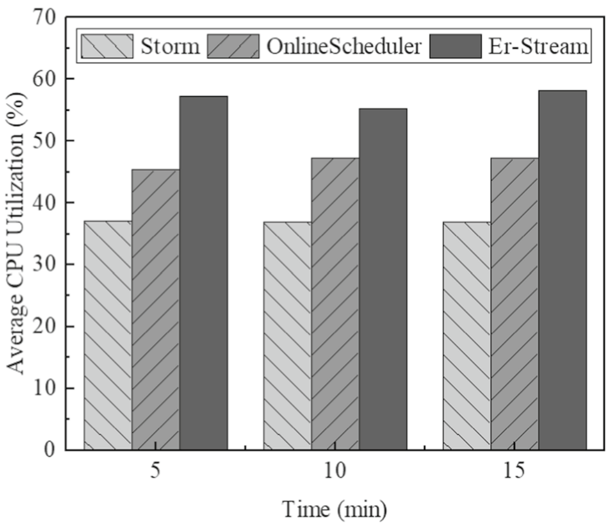


Fig. 15 Average CPU utilization comparison in resource-limited scenarios for WordCount

5.5 Performance results of resource usage

The resource usage comparison in latency-constraint scenarios for WordCount is given in Fig. 17.

For the first 7 min, both the OnlineScheduler and the Er-Stream operate with parallelism of operators set at (2, 7, 6). At the 8th minute, Er-Stream adjusts the operator parallelism to (2, 5, 4) by collecting system data and calculating the optimal parallelism, resulting in a 26.6% reduction in resource usage. Like OnlineScheduler, Storm does not scale resources dynamically.

6 Related work

6.1 Operator parallelism

DSC systems require the optimization of operator parallelism and efficient resource utilization to improve performance such as latency and throughput. Some existing stream processing systems, such as Apache Storm, lack mechanisms for setting operator parallelism, necessitating manual configuration by users. However, improper configurations can significantly impact system performance. Researchers have conducted numerous studies on operator parallelism, proposing various effective methods.

For example, Tang et al. [22] proposed an elastic strategy called DRS + based on Apache Storm for auto-scaling during resource scheduling. The strategy combines resource auto-scaling and load reduction, introducing the *RLA tradeoff* method to achieve a balance between resource consumption, system latency, and

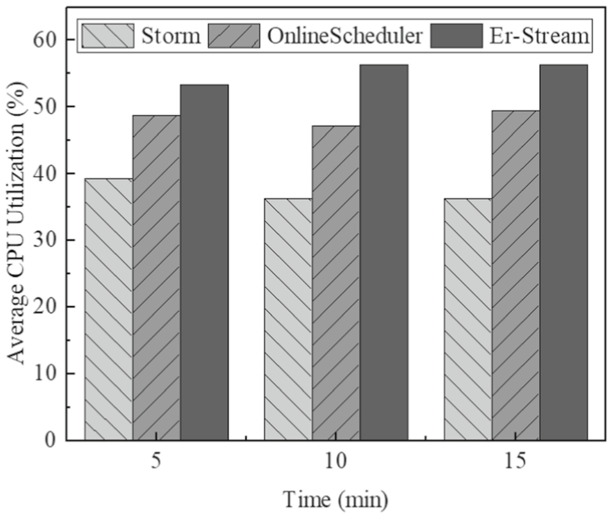


Fig. 16 Average CPU utilization comparison in latency-constraint scenarios for Top_N

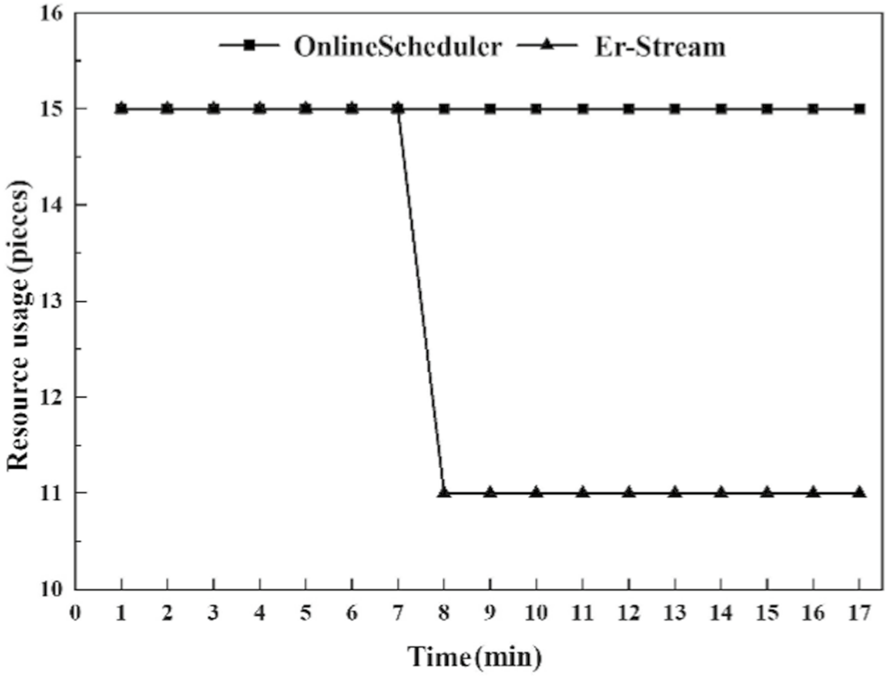


Fig. 17 Resource usage comparison in latency-constraint scenarios for WordCount

result accuracy. The strategy aligns with the real-time response characteristics demanded by existing stream processing systems, achieving low resource consumption and high utility.

In efforts to improve the utilization of underlying resources and manage varying workloads, Cardellini et al. [23] designed and implemented resilience and stateful task migration mechanisms. These innovations allow Storm to dynamically adjust the number of executors at runtime, scaling them up or down as per requirements.

Additionally, Li et al. [24] proposed two algorithms—the Min Latency and Max Throughput algorithms—to calculate optimal operator parallelism for achieving minimum latency and maximum throughput, respectively, while operating under resource constraints. Experiments in a cloud environment demonstrated the effectiveness of these algorithms in resource allocation concerning latency and throughput.

Auto-scaling of operators has emerged as an excellent method for DSC systems. In their work, Liu et al. [25] proposed the fast and accurate auto-scaling method called QAAS. This method uses operator performance models to automatically scale operator parallelism within Flink jobs. QAAS maintains job stability despite load changes, minimizes the number of job adjustments, and reduces data backlog by 50%. In addition, it achieves nearly double resource savings compared to the linear model.

In another study aimed at estimating resource utilization in stream processing applications, Lombardi et al. [26] proposed a fine-grained model which supports independent scaling of operator and resource. Their work introduced proposed ELYSIUM, an elastic scaling framework for stream processing system. Additionally, the proposed ELYSIUM manages operator parallelism and resources. This framework efficiently mitigates throughput degradation while conserving resources.

All the aforementioned methods optimize operator parallelism and enhance system performance; there are areas where further improvements can be made. For instance, the method proposed by [23] emphasizes runtime CPU utilization but overlooks other crucial parameter such as latency and throughput. To provide a comprehensive perspective, we conduct a comparison of our work with the aforementioned methods, which is presented in Table 4 for clarity.

6.2 Operator reconfiguration

In DSC systems, operator reconfiguration and task scheduling are often interrelated. Operator reconfiguration involves parallelizing computing tasks, while task scheduling focuses on allocating these parallel tasks efficiently to available resources for execution. By combining operator reconfiguration with task scheduling, DSC systems can achieve effective data processing and real-time computing.

Subsequently, we will introduce the following reconfiguration strategies and task scheduling strategies, and analyze the merits of these works.

Table 4 Comparison of our work with related works in operator parallelism

Performance	Related work				Er-Stream
	[22]	[23]	[24]	[25, 26]	
Increasing throughput	✗	✗	✓	✗	✓
Increasing CPU utilization	✓	✓	✗	✗	✓
Saving resource	✓	✗	✗	✓	✓

In a study outlined in [21], researchers integrated load balancing, operator instance collocation, and horizontal scaling into an optimization problem. This integration led to the development of ALBIC, a method focused on optimizing the collocation of operator instances. ALBIC effectively maintains a balanced system load, resulting in minimal runtime load. Additionally, it maximizes system configuration without compromising load balancing or incurring significant adaptation costs.

Furthermore, Mao et al. [27] proposed Trisk, a control plane that supports multiple reconfigurations. Trisk supports generic reconfiguration based on task-centric abstractions and encapsulates basic operations. This approach enables the description of reconfiguration through composed basic operations on abstractions, achieving a generic, efficient, and user-friendly reconfiguration process for DSC systems.

In recent years, swarm intelligence algorithms (SI) have found applications in operator configuration and task scheduling. For example, Farrokh et al. [28] proposed operator SP-Ant, a method for operator scheduling and reconfiguration based on the ant colony algorithm. SP-Ant initially uses bin-packing algorithm for initial configuration of operators. Then, an iterative process using the evolutionary ant colony optimization algorithm explores and develops the best operator configuration scheme by considering communication costs between operators.

Several researchers have optimized the scheduling problem using various methods. For instance, Li et al. [29] proposed the cost-efficient task scheduling algorithm (CETSA) and the cost-efficient load balancing algorithm (LBA-CE). These were designed to address certain issues existing in Flink's default task scheduling algorithm.

Online tuple scheduling demonstrates excellent performance. Huang et al. [30] proposed POTUS, an online predictive scheduling method that reduces data stream response time through distributed data stream direction. Li et al. [31] proposed Hone, an online tuple scheduler specifically designed to balance queue backlogs

Table 5 Comparison of our work with related works in operator reconfiguration

Performance	Related work					Er-Stream
	[16]	[27]	[28]	[29]	[30, 31]	
Reducing processing latency	✗	✓	✓	✗	✓	✓
Reducing communication cost	✗	✗	✓	✓	✗	✓
Reducing scheduling cost	✗	✓	✓	✓	✗	✓
Avoiding overload	✓	✗	✗	✗	✗	✓

among various tasks, reducing stragglers in DSP jobs and subsequently reducing the end-to-end processing latency of tuples. While these mentioned operator reconfiguration and scheduling strategies have successfully optimized one or more parameters within DSC systems, some overlook aspects such as the methods to reduce communication costs at runtime. A comparison between our work and theirs is presented in Table 5.

7 Conclusions and future work

In this paper, we tackle the challenge of enhancing system performance within resource-limited and latency-constraint scenarios by introducing Er-Stream, an elastic reconfiguration strategy for operators. Our work can be summarized as follows:

In terms of operator parallelism, we leverage the open-loop Jackson queuing network to find operators which significantly impact the average latency. We prioritize allocating available resource to these operators to minimize latency across the system.

To reduce communication cost, we place instances with the highest communication traffic on the same worker whenever possible. This approach is mirrored in the assignment of workers to nodes, minimizing communication overhead.

To avoid unnecessary scheduling, we set two thresholds. The CPU utilization threshold prevents worker and node overloads, while the performance optimization threshold guides reconfiguration decisions, reducing unnecessary costs.

As part of future work, we will be focusing on:

1. Fluctuating Data Streams: Test Er-Stream in fluctuating data stream environments. Our current tests were conducted in fixed-rate streaming settings, but real scenarios often involve significant data stream fluctuations. Studying how to predict and adapt operator parallelism for such fluctuations is crucial.
2. Multidimensional Resource Consideration: While we have considered CPU resources in this article, future efforts will involve expanding our model to encompass memory and network bandwidth limitations. Adapting operator parallelism based on these dimensions will enhance the reconfiguration strategy's effectiveness and adaptability.

Acknowledgements This work is supported by the National Natural Science Foundation of China under Grant No. 62372419 and the Fundamental Research Funds for the Central Universities under Grant No.265QZ2021001. This paper is a substantial extension of a short paper [35] presented at ICPADS 2023.

Author contribution Dawei Sun involved in conceptualization, methodology, validation, writing—original draft, funding acquisition. Yinuo Fan took part in validation, writing—original draft, formal analysis. Chengjun Guan involved in methodology, investigation, writing—original draft. Jia Rong took part in methodology, investigation, writing, data curation. Shang Gao involved in formal analysis, writing—review & editing. Rajkumar Buyya took part in methodology, writing—review & editing, funding acquisition.

Data availability No datasets were generated or analyzed during the current study.

Declarations

Competing Interests The authors declare no competing interests.

References

1. Gonzalez-Guerrero P, Stan MR (2019) Asynchronous stream computing for low power IoT. In: 2019 IEEE 62nd international midwest symposium on circuits and systems (MWSCAS), pp 1135–1138 (2019). IEEE
2. Juneja A, Das NN (2019) Big data quality framework: pre-processing data in weather monitoring application. In: 2019 International conference on machine learning, big data, cloud and parallel computing (COMITCon), pp 559–563. IEEE
3. Imai S, Patterson S, Varela CA (2017) Maximum sustainable throughput prediction for data stream processing over public clouds. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp 504–513. IEEE
4. Apache Storm, http://storm.apache/org/, last accessed 03 Dec 2023
5. Apache Samza, http://samza.apache.org/, last accessed 03 Dec 2023
6. Apache Flink, http://flink.apache.org/, last accessed 2023/12/03
7. Twitter Heron, https://github.com/apache/incubator-heron, last accessed 03 Dec 2023
8. Spark streaming. https://spark.apache.org/, last accessed 03 Dec 2023
9. Alghamdi MI, Jiang X, Zhang J, Zhang J, Jiang M, Qin X (2017) Towards two-phase scheduling of real-time applications in distributed systems. J Netw Comput Appl 84:109–117
10. Assuncao MD, Silva Veith A, Buyya R (2018) Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J Netw Comput Appl 103:1–17
11. Gedik B, Schneider S, Hirzel M, Wu K-L (2013) Elastic scaling for data stream processing. IEEE Trans Parallel Distrib Syst 25(6):1447–1463
12. Eskandari L, Mair J, Huang Z, Eyers D (2018) T3-scheduler: a topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. Futur Gener Comput Syst 89:617–632
13. Cardellini V, Lo Presti F, Nardelli M, Russo Russo G (2018) Optimal operator deployment and replication for elastic distributed data stream processing. Concurr Comput: Pract Exp 30(9):4334
14. Fang J, Zhang R, Fu TZ, Zhang Z, Zhou A, Zhou X (2018) Distributed stream rebalance for stateful operator under workload variance. IEEE Trans Parallel Distrib Syst 29(10):2223–2240
15. Hesse G, Matthies C, Glass K, Huegle J, Uflacker M (2019) Quantitative impact evaluation of an abstraction layer for data stream processing systems. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp 1381–1392. IEEE
16. Fu TZ, Ding J, Ma RT, Winslett M, Yang Y, Zhang Z (2017) DRS: auto-scaling for real-time stream analytics. IEEE/ACM Trans Netw 25(6):3338–3352
17. Nardelli M, Cardellini V, Grassi V, Presti FL (2019) Efficient operator placement for distributed data stream processing applications. IEEE Trans Parallel Distrib Syst 30(8):1753–1767
18. Zhang Z, Jin P, Wang X, Liu R, Wan S (2019) N-storm: efficient thread-level task migration in Apache Storm. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp 1595–1602. IEEE
19. Buddhika T, Stern R, Lindburg K, Ericson K, Pallickara S (2017) Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. IEEE Trans Parallel Distrib Syst 28(12):3553–3569
20. Aniello L, Baldoni R, Querzoni L (2013) Adaptive online scheduling in storm. In: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, pp 207–218 (2013)
21. Madsen KGS, Zhou Y, Cao J (2017) Integrative dynamic reconfiguration in a parallel stream processing engine. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp 227–230. IEEE

22. Tang K, Hao Z, Cai R, Fu TZ, Yang Y, Wang L, Winslett M, Zhang Z (2020) Drs+: load shedding meets resource auto-scaling in distributed stream processing. In: 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp 292–301
23. Cardellini V, Nardelli M, Luzi D (2016) Elastic stateful stream processing in storm. In: 2016 International Conference on High Performance Computing & Simulation (HPCS), pp 583–590. IEEE
24. Li W, Zhang Z, Shu Y, Liu H, Liu T (2022) Toward optimal operator parallelism for stream processing topology with limited buffers. J Supercomput 78(11):13276–13297
25. Liu S, Li Y, Yang H, Dun M, Chen C, Zhang H, Li W (2024) QAAS: quick accurate auto-scaling for streaming processing. Front Comput Sci 18(1):181201
26. Lombardi F, Aniello L, Bonomi S, Querzoni L (2017) Elastic symbiotic scaling of operators and resources in stream processing systems. IEEE Trans Parallel Distrib Syst 29(3):572–585
27. Mao Y, Huang Y, Tian R, Wang X, Ma RT (2021) Trisk: task-centric data stream reconfiguration. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 214–228
28. Farrokh M, Hadian H, Sharifi M, Jafari A (2022) Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters. Expert Syst Appl 191:116322
29. Li H, Xia J, Luo W, Fang H (2022) Cost-efficient scheduling of streaming applications in Apache Flink on cloud. IEEE Trans Big Data
30. Huang X, Shao Z, Yang Y (2020) Potus: Predictive online tuple scheduling for data stream processing systems. IEEE Trans Cloud Comput 10(4):2863–2875
31. Li W, Liu D, Chen K, Li K, Qi H (2021) Hone: Mitigating stragglers in distributed stream processing with tuple scheduling. IEEE Trans Parallel Distri Syst 32(8)
32. Li H, Wu J, Jiang Z, Li X, Wei X (2017) Task allocation for stream processing with recovery latency guarantee. In: 2017 IEEE international conference on cluster computing (CLUSTER), pp 379–383. IEEE
33. Wang C, Meng X, Guo Q, Weng Z, Yang C (2017) Automating characterization deployment in distributed data stream management systems. IEEE Trans Knowl Data Eng 29(12):2669–2681
34. Kalavri V, Liagouris J, Hoffmann M, Dimitrova D, Forshaw M, Roscoe T (2018) 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), pp 783–798
35. Sun D, Guan C, Fan Y, Rong J, Gao S (2023) A latency guaranteed scheduling strategy under performance constraints in big data stream computing environments. In: 2023 International Conference on Parallel and Distributed Systems (ICPADS)
36. Karimov J, et al (2018) Benchmarking distributed stream data processing systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE)
37. Aliyun (2025) User behavior data from taobao for recommendation. [Online] Available at: https://tianchi.aliyun.com/dataset/649?t=1679727494514. Accessed 22 Feb 2025

Authors and Affiliations

Dawei Sun[1] · Yinuo Fan[1] · Chengjun Guan[1] · Jia Rong[2] · Shang Gao[3] · Rajkumar Buyya[4]

✉ Dawei Sun
sundaweicn@cugb.edu.cn

Yinuo Fan
fanyinuocn@email.cugb.edu.cn

Chengjun Guan
guanchengjun@email.cugb.edu.cn

Jia Rong
jiarong@acm.org

Shang Gao
shang.gao@deakin.edu.au

Rajkumar Buyya
rbuyya@unimelb.edu.au

1 School of Information Engineering, China University of Geosciences, Beijing 10083, People's Republic of China

2 Department of Data Science and AI, Monash University, Clayton, VIC 3800, Australia

3 School of Information Technology, Deakin University, Geelong, VIC 3216, Australia

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