Future Generation Computer Systems 175 (2026) 108119

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

A fine-grained task scheduling strategy for resource auto-scaling over fluctuating data streams

Yinuo Fan [a], Dawei Sun [a,*], Minghui Wu [a], Shang Gao [b], Rajkumar Buyya [c]

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

ARTICLE INFO

Keywords:
Stream computing systems
Fine-grained scheduling
Resource auto-scaling
Communication cost
Fluctuating data streams

ABSTRACT

Resource scaling is crucial for stream computing systems in fluctuating data stream scenarios. Computational resource utilization fluctuates significantly with changes in data stream rates, often leading to pronounced issues of resource surplus and scarcity within these systems. Existing research has primarily focused on addressing resource insufficiency at runtime; however, effective solutions for handling variable data streams remain limited. Furthermore, overlooking task communication dependencies during task placement in resource adjustment may lead to increased communication cost, consequently impairing system performance. To address these challenges, we propose Ra-Stream, a fine-grained task scheduling strategy for resource auto-scaling over fluctuating data streams. Ra-Stream not only dynamically adjusts resources to accommodate varying data streams, but also employs fine-grained scheduling to optimize system performance further. This paper explains Ra-Stream through the following aspects: (1) Formalization: We formalize the application subgraph partitioning problem, the resource scaling problem and the task scheduling problem by constructing and analyzing a stream application model, a communication model, and a resource model. (2) Resource scaling and heuristic partitioning: We propose a resource scaling algorithm to scale computational resource for adapting to fluctuating data streams. A heuristic subgraph partitioning algorithm is also introduced to minimize communication cost evenly. (3) Fine-grained task scheduling: We present a fine-grained task scheduling algorithm to minimize computational resource utilization while reducing communication cost through thread-level task deployment. (4) Comprehensive evaluation: We evaluate multiple metrics, including latency, throughput and resource utilization in a real-world distributed stream computing environment. Experimental results demonstrate that, compared to state-of-the-art approaches, Ra-Stream reduces system latency by 36.37 % to 47.45 %, enhances system maximum throughput by 26.2 % to 60.55 %, and saves 40 % to 46.25 % in resource utilization.

1. Introduction

Stream computing systems demonstrate exemplary performance in processing stream applications that demand low latency (on the order of milliseconds) and high throughput [1], such as traffic monitoring [2], anomaly detection [3], and Internet of Things [4]. To support the demanding requirements of stream applications, many stream processing systems have emerged such as Apache Flink and Spark Streaming [5]. Among them, Apache Storm is particularly well-suited for time-critical processing scenarios and has demonstrated extensive applicability in various fields. In recent years, it has established itself as a mainstream stream computing framework due to its excellent performance [6].

Resource scaling and task scheduling [7,8] are crucial for achieving low system latency and high system throughput-two key metrics for evaluating stream computing systems [9]. In real-world application scenarios, data stream rates are often not uniformly stable but instead fluctuate over time due to various factors [10]. Static scheduling schemes struggle to adapt to such fluctuations, especially with respect to resource utilization, resulting in two major issues: (1) When data stream rates are excessively high, the computational load on compute nodes becomes overwhelming, increasing latency and potentially causing system crashes. (2) When data stream rates are persistently low, computational resources allocated to tasks cannot be dynamically released, leading to substantial resource wastage.

* Corresponding author.
E-mail addresses: fanyinuocn@email.cugb.edu.cn (Y. Fan), sundaweicn@cugb.edu.cn (D. Sun), wuminghui@email.cugb.edu.cn (M. Wu), shang.gao@deakin.edu.au (S. Gao), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2025.108119
Received 24 March 2025; Received in revised form 1 July 2025; Accepted 1 September 2025
Available online 6 September 2025

Existing research [11,12] has made progress in mitigating the negative impacts of fluctuating data streams through optimized scheduling strategies. For instance, [13] proposed two resource allocation strategies that utilize greedy algorithms and genetic algorithms, respectively, to achieve efficient execution of stream applications. Similarly, [14] introduced a heuristic algorithm, which dynamically reconfigures stream applications based on variations in data stream rates and the availability of computational resources. However, the magnitude of communication cost directly impacts system performance, how to minimize communication cost during the resource scaling process remains a critical challenge.

Modeling stream applications as directed acyclic graphs (DAGs) and exploiting task dependencies to optimize system performance is a promising approach [15]. For example, [16] utilized a dynamic programming algorithm on the critical path of DAG to reduce communication cost. Similarly, [17] proposed partitioning the DAG into multiple subgraphs based on the communication volume between tasks and scheduling at the subgraph level. However, the deployment of tasks within a compute node directly influences communication cost, as the expenses associated with communication between tasks within the same process differ from those between tasks across processes. This is an indispensable factor to consider in task scheduling.

From the preceding analysis, there are three primary challenges in task scheduling: (1) How can the scheduling scheme be adjusted to accommodate fluctuating data streams? (2) How can effective resource utilization be achieved while ensuring optimal system performance? (3) How can task deployment within compute nodes be optimized to minimize communication cost? These challenges have sparked our research interest, as we aim to design a scheduling strategy that dynamically mitigates the negative impacts of fluctuating data streams while minimizing communication costs.

To address these challenges, we propose a fine-grained task scheduling strategy called Ra-Stream, which dynamically scales computational resources based on current data stream rate to effectively accommodate fluctuating data streams, achieving efficient resource utilization while preventing excessive loads on compute nodes. Additionally, Ra-Stream minimizes communication cost through fine-grained task deployment, further ensuring low-latency processing.

1.1. Contributions

This paper proposes a fine-grained task scheduling strategy (Ra-Stream) for resource auto-scaling over fluctuating data streams and improving the resource utilization and latency of distributed stream computing systems. The key contributions are as follows:

(1) Formalization of the problem: We construct and analyze the stream application model, communication model, and resource model to formalize the application subgraph partition problem, the resource scaling problem, and the task scheduling problem.
(2) Resource scaling and heuristic partitioning algorithms: We propose a resource scaling algorithm based on our heuristic subgraph partitioning algorithm to determine the minimum necessary number of compute nodes, achieving efficient resource utilization while accommodating fluctuating data streams.
(3) Fine-grained task scheduling algorithm: We propose a fine-grained task scheduling algorithm that minimizes communication cost through thread-level task deployment, thereby optimizing overall system performance. Additionally, we set two thresholds for compute nodes to avoid underloading or overloading.
(4) Implementation and evaluation: We implement and integrate Ra-Stream into Apache Storm and evaluate various metrics, including system latency, system maximum throughput, and resource utilization, in real-world fluctuating data stream scenarios. The experimental results confirm the effectiveness of Ra-Stream.

1.2. Paper organization

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 introduces the stream application model, communication model, and resource model. Section 4 provides the problem statement, including descriptions of subgraph partitioning, task scheduling, and resource scaling. Section 5 focuses on the system architecture and main algorithms of Ra-Stream. Section 6 details the experimental environment, parameter setup, and performance evaluation. The conclusions and future works are presented in Section 7.

2. Related work

We review related work in the field of stream computing, which can be broadly divided into two main areas: scheduling for stream computing systems and performance optimization of stream computing systems.

2.1. Scheduling for stream applications

Achieving excellent system performance in stream applications has been the focus of numerous researchers [18], who have devoted significant effort to improving scheduling strategies. However, it has been demonstrated that the scheduling problem is NP-hard, making it inherently challenging to find an optimal scheduling solution [19,20].

To reduce communication latency, [21] proposed SP-Ant, which places high-communication operators on the same compute node using a bin-packing algorithm and allocates low-communication operators using an ant colony optimization algorithm. However, this approach lacks consideration for the resource utilization of compute nodes, potentially leading to overloads that adversely affect overall system latency.

To optimize load balancing while reducing job execution costs, [22] introduced a Cost-Efficient Task Scheduling Algorithm (CETSA) alongside a Cost-Effective Load Balancing Algorithm (LBA-CE). These algorithms ensure a balanced workload in heterogeneous clusters while minimizing costs. However, they inadequately consider real-time fluctuations in data stream velocities, which do not align well with the dynamic nature of actual data flows.

To address the complexities and unpredictability of dynamic streaming workflow scenarios, [23] proposed a resource scheduling and provisioning method for processing dynamic stream workflows under latency constraints. This approach assumes that data communication overhead can be ignored; however, in real-world scenarios, communication overhead is a significant factor that cannot be overlooked.

To optimize resource utilization and enhance task reliability across the network, [24] applied a satisfiability modulo theory (SMT) constraint solver to determine the optimal processing quality at each node, ensuring target system reliability while minimizing resource consumption. However, this work lacks consideration of complex and variable application scenarios, limiting its applicability.

To adapt to dynamically changing workloads, [25] proposed MorphStream, which makes accurate scheduling decisions at runtime with minimal overhead, resulting in excellent performance improvements. However, this method requires the construction of a task dependency graph and the maintenance of multiple versions of state storage, which significantly increase memory resource consumption.

To achieve optimal performance by minimizing computational bottlenecks at the edge computing environments, [26] proposed a framework called Beaver for strategic placement of stream operators. While Beaver addresses variations in network latency and bandwidth, further improvements can be made in dynamically adjusting compute resource allocation to address fluctuating stream rates.

Similarly, [27] formulated an optimization strategy to reduce network latency in distributed environments using a broad spectrum of computational resources. However, it does not account for dynamic scaling based on stream variability.

Table 1
Comparison of Ra-Stream and related work.

Related work	Aspects			
	Scheduling object	Load balancing	Communication cost	Resource scaling
SP-Ant [21]	Task	×	✓	×
CETSA & LBA-CE [22]	Task	✓	✓	×
VM provisioner [23]	Task	×	×	✓
SMT [24]	Resource	×	×	✓
MorphStream [25]	Resource	×	×	✓
Beaver [26]	Resource	×	✓	✓
Ra-Stream (Ours)	Resource, Task	✓	✓	✓

In summary, these methods have substantially improved scheduling in stream computing systems. However, most of them fail to consider system performance optimization from multiple dimensions. For clarity and conciseness, a comparison of our work with the relevant studies is summarized in Table 1.

2.2. Optimization for stream computing systems

To improve the performance of stream computing systems, extensive work has focused on optimization through various approaches, including large parameter tuning [28] and tuple scheduling [29]. Below, we introduce and analyze several noteworthy works.

To mitigate the high resource costs associated with automatic tuning process, [28] introduced a general and efficient Spark tuning framework. This framework utilizes Bayesian optimization (BO) to tackle the generalized tuning problem involving multiple objectives and constraints. By tuning parameters according to the actual periodic execution of each job, the framework allows for online evaluation and parameter optimization.

To address the imbalance of workloads among downstream tasks, [29] proposed POTUS, a predictive online tuple scheduling strategy that directs data stream in a distributed manner to reduce response time in stream processing. Similarly, [30] introduced a popularity-aware differentiated distributed stream processing system called Pstream. Pstream employs a novel lightweight probabilistic counting scheme to identify hot keys in dynamic real-time stream. This approach effectively adapts to changes in dynamic popularity within high-velocity streams. Additionally, [31] developed Hone, a tuple scheduler that uses the online maximum backlog first (LBF) algorithm to minimize the maximum queue backlog across all tasks, thereby improving processing efficiency.

To reduce recovery time from system failures, [32] proposed A-FP4S, an adaptive fragments-based parallel state recovery mechanism. This mechanism divides each node's local state into multiple segments, which are periodically stored across neighboring nodes. During a failure, different sets of available segments are used to parallelize the reconstruction of the lost state. This approach offers significant scalability for managing lost states and can tolerate multiple node failures.

In summary, these works have made significant contributions to stream processing optimization. Our work focuses on optimizing stream computing systems through resource scaling and task scheduling, while topics such as parameter tuning, tuple scheduling, and state management fall outside the scope of this study.

3. System model

We formalize the resource scaling problem in distributed stream computing systems by defining the stream application model, resource model, and communication model. For clarity, we summarize the main notations used throughout the paper in Table 2.

3.1. Stream application model

The functionality of a stream application is typically defined by users through a logical topology [33]. This logical topology can be described as a Directed Acyclic Graph (DAG) [34], denoted as $G = (V(G), E(G))$.

Here, $V(G) = \{v_i | i \in 1, \dots, n\}$ is a finite set of n vertices. Each vertex v_i represents a specific function $fun(v_i)$, and the function of each vertex is set by users.

$E(G) = \{e_{i,j} | v_i, v_j \in V(G)\}$ is a finite set of directed edges. An edge $e_{i,j} \in E(G)$ indicates a data stream flowing from the upstream vertex v_i to the downstream vertex v_j.

Before processing the stream application G, users can specify the number of tasks for each vertex v_i, denoted as $V_i(G) = \{v_{i,k} | k \in \{1, \dots, m\}\}$, where m is the number of tasks for v_i. All tasks of vertex v_i share the same function, i.e., $fun(v_{i,1}) = fun(v_{i,2}) = \cdots = fun(v_{i,m})$.

The edges connecting tasks of upstream vertex v_i to tasks of downstream vertex v_j are represented as $E_{i,j}(G) = \{e_{i,k,\ j,l} | k \in \{1, \dots, m\},\ l \in \{1, \dots, s\}\}$, where s is the number of tasks for v_j. The weight of edge $e_{i,k,\ j,l}$ is denoted as $Tr(v_{i,k}, v_{j,l})$, indicating the tuple transmission rate between task $v_{i,k}$ and task $v_{j,l}$.

For clarity, we illustrate the logical topology of the commonly used WordCount stream application in Figs. 1 and 2. As shown in Fig. 1, the logical topology of WordCount consists of four vertices, represented as $V(G_{WordCount}) = \{v_1, v_2, v_3, v_4\}$, and three directed edges, $E(G_{WordCount}) = \{e_{1,2}, e_{2,3}, e_{3,4}\}$. The three edges represent the flow of data tuples within the topology. Following the data flow direction, the four vertices perform the following functions: "reading data tuples", "splitting sentences into words", "counting words", and "outputting results". In Fig. 2, the numbers of tasks for each vertex are 2, 3, 3 and 2,

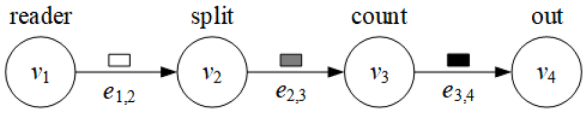

Fig. 1. Logical topology of WordCount.

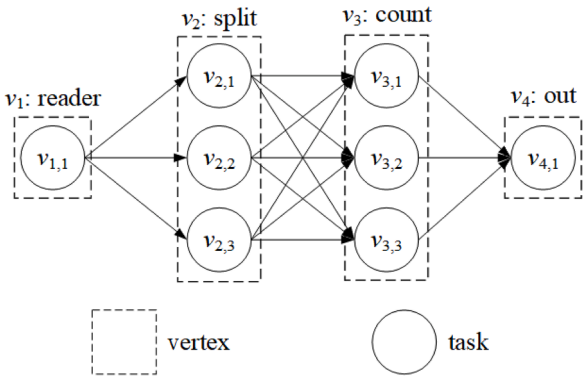

Fig. 2. Task topology of WordCount.

Table 2
Main notations used in the paper.

Notation	Description	Notation	Description
G	DAG of a stream application	$R_{v_{i,k},st}$	Resource utilization of task $v_{i,k}$ in st
$v_{i,k}$	Task k of vertex i	p_i	Acceptance probability of scheme x_i
$e_{i,k,\ j,l}$	Edge from task $v_{i,k}$ to task $v_{j,l}$	$E_{cut}(G)$	Set of cut edges in G
cn	Compute node	G_i^{sub}	Subgraph i of G
st	Statistical time frame	$W_{int}(G_i^{sub})$	Set of internal edges in G_i^{sub}
$Tr(v_{i,k}, v_{j,l})$	Tuple transmission rate between tasks $v_{i,k}$ and $v_{j,l}$	$W_{cut}(G)$	Sum of cut edges' weights
E_{st}^{Tr}	Average tuple transmission rate in st	$R^c_{G_i^{sub}}$	CPU requirement of subgraph G_i^{sub}
$num_{st}^{v_{i,k}}$	Number of tuples processed by $v_{i,k}$ in st	$R^m_{G_i^{sub}}$	Memory requirement of subgraph G_i^{sub}
$R^{io}_{G_i^{sub}}$	I/O requirement of subgraph G_i^{sub}	$U^c_{cn,st}$	CPU utilization of cn in st
$U^m_{cn,st}$	Memory utilization of cn in st	$U^{io}_{cn,st}$	I/O utilization of cn in st
σ_W	Variance of internal weight sums of subgraphs	r	Ratio of $W_{cut}(G)$ to the total weights of G

respectively. Tasks for the same vertex, for example, $v_{1,1}$ and $v_{1,2}$, share an identical function.

3.2. Communication model

During the execution of a stream application, tuples are transferred between vertices and, more specifically, between the tasks of those vertices, which constitutes communication. Once tasks are scheduled on the compute cluster, the resulting communication cost can be categorized into three types: inter-thread, inter-process and inter-compute node. Among them, inter-compute node communication incurs the highest cost, inter-process communication incurs moderate cost, and inter-thread communication incurs the lowest cost.

Due to the fluctuation of stream rates, the communication traffic at any given time point is inherently random, which makes it unsuitable for generalizability in practice. To address this, we calculate the mathematical expectation of tuples transmitted between two communicating tasks over a statistical time frame st spanning from onset time t_o to completion time t_c. Denoted as E_{st}^{Tr}, this expectation value represents the average transmission rate and mitigates the impact of abrupt fluctuations in data streams at specific time points, as expressed in Eq. (1):

$$E_{st}^{Tr} = \frac{\int_{t_o}^{t_c} E_t^{Tr} dt - max(E_t^{Tr}) - min(E_t^{Tr})}{t_c - t_o}, \tag{1}$$

where E_t^{Tr} is the tuple transmission at time t, and $t \in [t_o, t_c]$.

$Tr(v_{i,k}, v_{j,l})$ is the average tuple transmission rate in st from the task $v_{i,k}$ to the task $v_{j,l}$, it serves as a key input to the subgraph partitioning and scheduling algorithms and directly influences the optimization objective by quantifying the communication cost associated with separating interdependent tasks. $Tr(v_{i,k}, v_{j,l})$ satisfies Eq. (2):

$$Tr(v_{i,k}, v_{j,l}) = \begin{cases} 0, & \text{If no tuple transmission} \\ & \text{between } v_{i,k} \text{ and } v_{j,l}, \\ E_{st}^{Tr}, & \text{Otherwise.} \end{cases} \tag{2}$$

3.3. Resource model

The resources of compute nodes can be measured across various dimensions, such as CPU, memory, and I/O [35]. Based on our previous benchmarking experiments, we have observed that CPU, memory and I/O overutilization become bottlenecks in system operation. Consequently, this paper explicitly addresses the resource utilization of CPU, memory, and I/O in compute nodes.

A compute node can execute multiple tasks concurrently. Let the set of all tasks running on a compute node cn be denoted as T_{cn}, and the CPU utilization of cn within a statistical time frame st as $U^c_{cn,st}$.

The number of tuples processed by task $v_{i,k}$ running on cn in st is represented by $num_{st}^{v_{i,k}}$.

The CPU utilization of task $v_{i,k}$ in st can then be calculated by Eq. (3):

$$R^c_{v_{i,k},st} = \frac{num_{st}^{v_{i,k}}}{\sum_{v_{j,l} \in T_{cn}} num_{st}^{v_{j,l}}} \cdot U^c_{cn,st}, \tag{3}$$

where $R^c_{v_{i,k},st}$ represents the CPU utilization of $v_{i,k}$, $num_{st}^{v_{j,l}}$ denotes the number of tuples processed by cn, and $v_{j,l}$ is a task within T_{cn} on node cn in st.

Similarly, the memory utilization and I/O utilization of task $v_{i,k}$ running on cn in st can be obtained by Eqs. (4) and (5), respectively:

$$R^m_{v_{i,k},st} = \frac{num_{st}^{v_{i,k}}}{\sum_{v_{j,l} \in T_{cn}} num_{st}^{v_{j,l}}} \cdot U^m_{cn,st}, \tag{4}$$

$$R^{io}_{v_{i,k},st} = \frac{num_{st}^{v_{i,k}}}{\sum_{v_{j,l} \in T_{cn}} num_{st}^{v_{j,l}}} \cdot U^{io}_{cn,st}, \tag{5}$$

where $R^m_{v_{i,k},st}$ and $R^{io}_{v_{i,k},st}$ are the memory utilization and I/O utilization of $v_{i,k}$ on cn, respectively. $U^m_{cn,st}$ and $U^{io}_{cn,st}$ represent the memory utilization and I/O utilization of cn in st, respectively.

The total resource utilization $R_{v_{i,k},st}$ by task $v_{i,k}$ running on cn in st can then be calculated using a weighted combination of task $v_{i,k}$'s CPU, memory and I/O utilization, as shown in Eq. (6):

$$R_{v_{i,k},st} = \alpha \cdot R^c_{v_{i,k},st} + \beta \cdot R^m_{v_{i,k},st} + (1 - \alpha - \beta) \cdot R^{io}_{v_{i,k},st}, \tag{6}$$

where α and β are weighted factors that determine the relative importance of CPU, memory and I/O utilization for task $v_{i,k}$ running on cn, $\alpha, \beta \in [0, 1]$ and $\alpha + \beta < 1$.

4. Problem statement

We formalize the scheduling-related problems in the context of fluctuating data streams, including subgraph partitioning, resource scaling, and task scheduling.

4.1. Subgraph partitioning

The subgraph partitioning problem [36] can be described as follows: a user submits a stream application $G = (V(G), E(G))$ to a compute cluster consisting of n_{cn} available compute nodes cn. If G requires k compute nodes to run, where $k \leq n_{cn}$, then G should be partitioned into k subgraphs. The subgraph partitioning problem is formalized as: $G = \bigcup_{i=1}^{k} G_i^{sub}$, where G_i^{sub} represents a subgraph and $G_i^{sub} \subseteq G$.

The set of tasks in a subgraph G_i^{sub} is denoted as $T(G_i^{sub})$, and the relationships among the task sets are represented by Eq. (7):

$$\begin{cases} T(G) = \bigcup_{i=1}^{k} T(G_i^{sub}), \\ T(G_i^{sub}) \cap T(G_j^{sub}) = \emptyset, \forall i, j \in [1, k], i \neq j. \end{cases} \tag{7}$$

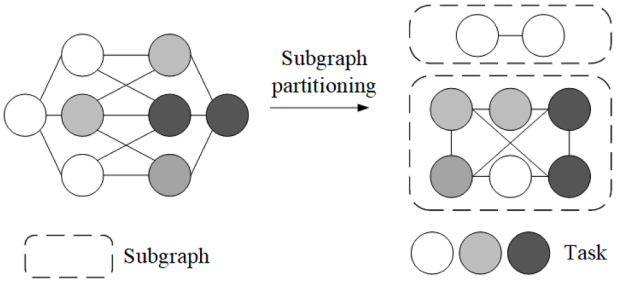

Fig. 3. Example of imbalanced graph partitioning scheme.

The subgraph partitioning also generates internal edges within subgraphs and cut edges connecting subgraphs. These relationships are described by Eq. (8):

$$\begin{cases} E(G_i^{sub}) = \{e_{i,k,\ j,l} \in E(G) | v_{i,k}, v_{j,l} \in T(G_i^{sub})\}, \\ E_{cut}(G) = \{e_{i,k,\ j,l} \in E(G) | v_{i,k} \in T(G_i^{sub}), v_{j,l} \in T(G_j^{sub})\}, \end{cases} \tag{8}$$

where $E(G_i^{sub})$ represents the set of internal edges within G_i^{sub}, $E_{cut}(G)$ denotes the set of cut edges connecting subgraphs, and $e_{i,k,\ j,l}$ is the edge connecting tasks $v_{i,k}$ and $v_{j,l}$.

The objective of graph partitioning is to minimize the weights of cut edges between different subgraphs while maximizing the weights of internal edges within subgraphs. Minimizing the weights of cut edges alone may result in an imbalanced partitioning scheme. For instance, Fig. 3 shows a scheme where one subgraph contains two tasks, while another contains six tasks, leading to an imbalance.

To achieve a balanced subgraph partitioning scheme, we aim to maximize the sum of the internal edge weights for each subgraph while keeping these sums nearly equal, all while minimizing the total weight of the cut edges. This is formalized in Eq. (9):

$$\begin{cases} \max \sum_{i=1}^{k} W_{int}(G_i^{sub}), \\ \min W_{cut}(G) = \min \sum_{e_{i,k,\ j,l} \in E_{cut}(G)} w(e_{i,k,\ j,l}), \\ W_{int}(G_1^{sub}) \approx W_{int}(G_2^{sub}) \approx, \ldots, \approx W_{int}(G_k^{sub}), \end{cases} \tag{9}$$

where $W_{int}(G_i^{sub})$ is the sum of internal edge weights within G_i^{sub}, $W_{cut}(G)$ is the sum of weights of cut edges $E_{cut}(G)$, and $w(e_{i,k,\ j,l})$ is the weight of edge $e_{i,k,\ j,l}$.

Fig. 4 illustrates a stream application G partitioned into four subgraphs. The objective of subgraph partitioning is to maximize and equalize the internal communication traffic within each subgraph while minimizing inter-subgraph communication. The subsequent section discusses the resource consumption problem related to these subgraphs.

4.2. Resource scaling

Computational resources are not infinite, making it essential to use them efficiently. In real-world applications, we need to consider the resource scaling problem during the scheduling process. Resource scaling refers to the process of configuring computational resources based on the current data stream rate, aiming to effectively prevent performance degradation due to insufficient resources while minimizing resource waste. A critical consideration before scheduling stream applications to the cluster is ensuring that no individual compute node becomes overloaded, while simultaneously minimizing resource wastage within those nodes.

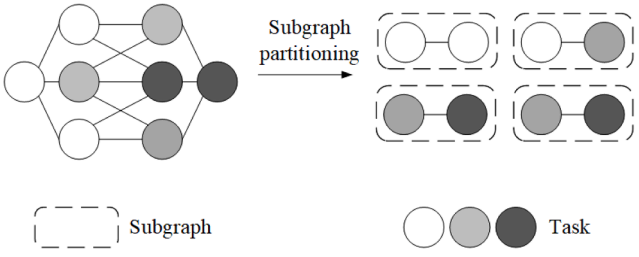

Fig. 4. Example of balanced graph partitioning scheme.

In the context of subgraphs partitioning, we need to consider the resource requirements of each subgraph to prevent resource overload when scheduling subgraph tasks to compute nodes. The resource demand for each subgraph can be determined by aggregating the resource requirements of all tasks within that subgraph, as described by Eq. (10):

$$\begin{cases} R^{c}_{G_i^{sub},st} = \sum_{v_{i,k} \in T(G_i^{sub})} R^{c}_{v_{i,k},st}, \\ R^{m}_{G_i^{sub},st} = \sum_{v_{i,k} \in T(G_i^{sub})} R^{m}_{v_{i,k},st}, \\ R^{io}_{G_i^{sub},st} = \sum_{v_{i,k} \in T(G_i^{sub})} R^{io}_{v_{i,k},st}, \end{cases} \tag{10}$$

where $R^{c}_{G_j^{sub},st}$, $R^{m}_{G_j^{sub},st}$ and $R^{io}_{G_j^{sub},st}$ represent the average CPU, memory and I/O resource demands of subgraph G_i^{sub} in statistical time frame st, respectively.

After the subgraph is scheduled to a compute node, that node may still have sufficient resources to run additional subgraphs. To ensure efficient utilization of computational resources, we calculate the CPU, memory and I/O resource requirements of the stream application G, as described by Eq. (11):

$$\begin{cases} R^{c}_{G,st} = \sum_{i=1}^{k} R^{c}_{G_i^{sub},st}, \\ R^{m}_{G,st} = \sum_{i=1}^{k} R^{m}_{G_i^{sub},st}, \\ R^{io}_{G,st} = \sum_{i=1}^{k} R^{io}_{G_i^{sub},st}, \end{cases} \tag{11}$$

where $R^{c}_{G,st}$, $R^{m}_{G,st}$ and $R^{io}_{G,st}$ are the average CPU, memory and I/O resource requirements of the stream application G in st, respectively.

Subsequently, we choose some compute nodes (m_{cn}) to serve as our operational nodes, indicated by $ON = \{on_i | i \in \{1, 2, \ldots, m_{cn}\}\}$, while the other nodes are considered idle nodes, represented by $IN = \{in_i | i \in \{1, 2, \ldots, n_{cn} - m_{cn}\}\}$. The available computational resources of these operational nodes must exceed the resource requirements of subgraphs. The available computational resources of these operational nodes can be calculated by Eq. (12):

$$\begin{cases} A^{c}_{ON,st} = \sum_{on_i \in ON} A^{c}_{on_i,st}, \\ A^{m}_{ON,st} = \sum_{on_i \in ON} A^{m}_{on_i,st}, \\ A^{io}_{ON,st} = \sum_{on_i \in ON} A^{io}_{on_i,st}, \end{cases} \tag{12}$$

where $A^{c}_{ON,st}$, $A^{m}_{ON,st}$ and $A^{io}_{ON,st}$ are the available CPU, memory and I/O resources of operational nodes, respectively. $A^{c}_{on_i,st}$, $A^{m}_{on_i,st}$ and $A^{io}_{on_i,st}$ are the available CPU, memory and I/O resources of an individual operational node.

As shown in Fig. 5, resource scaling consists of two operations: **resource shrink** and **resource extend**. As the data stream rate decreases, the computational resources required for running a stream application are reduced, enabling us to achieve similar system performance with fewer compute nodes. On the other hand, when the stream rate increases, the required resources grow, necessitating the deployment of additional compute nodes to sustain optimal system performance.

4.3. Task scheduling

After a stream application G is partitioned into k subgraphs, the subsequent task scheduling problem involves selecting k compute nodes

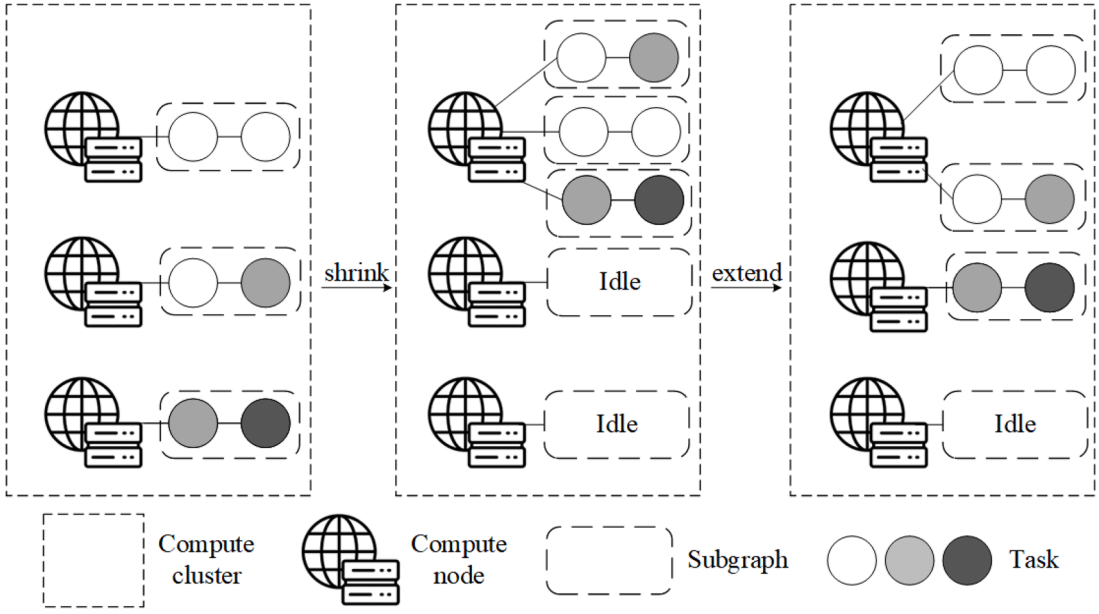

Fig. 5. Example of resource scaling.

from the available compute nodes and assigning the subgraphs to these nodes. A compute node may run multiple subgraphs; however, each subgraph can only be assigned to a single compute node. We denote $f^{de}_{G_i^{sub},cn_j}$ as the decision factor for mapping a subgraph G_i^{sub} to a compute node cn_j. It can be calculated by Eq. (13):

$$f^{de}_{G_i^{sub},cn_j} = \begin{cases} 1, \text{ If } A^c_{cn_j,st} > R^c_{G_i^{sub},st} \text{ and } A^m_{cn_j,st} > R^m_{G_i^{sub},st}, \\ 0, \text{ Otherwise.} \end{cases} \tag{13}$$

where $i \in \{1,2,\ldots,k\}$, and $j \in \{1,2,\ldots,n_{cn}\}$. $f^{de}_{G_i^{sub},cn_j}$ determines whether subgraph G_i^{sub} can be assigned to compute node cn_j. $A^c_{cn_j,st}$ and $A^m_{cn_j,st}$ are the available CPU and memory resources of compute node cn_j in statistical time frame st, respectively. $R^c_{G_i^{sub},st}$ and $R^m_{G_i^{sub},st}$ represent the CPU and memory resource requirements of G_i^{sub} in st.

Since multiple compute nodes may satisfy the resource requirements of G_i^{sub}, it is essential to identify the most suitable compute node for scheduling G_i^{sub}. For this purpose, we use $f^{fit}_{G_i^{sub},cn_j}$ as the fitness factor for scheduling subgraph G_i^{sub} to compute node cn_j. Subgraph G_i^{sub} will be scheduled to the compute node with the highest $f^{fit}_{G_i^{sub},cn_j}$, calculated by Eq. (14):

$$f^{fit}_{G_i^{sub},cn_j} = f^{de}_{G_i^{sub},cn_j} \cdot S^R_{G_i^{sub},cn_j}, \tag{14}$$

where $S^R_{G_i^{sub},cn_j}$ represents the resource surplus of cn_j after scheduling G_i^{sub} to it. The resource surplus is calculated by Eq. (15):

$$\begin{aligned} S^R_{G_i^{sub},cn_j} = {} & \gamma \cdot (A^c_{cn_j,st} - R^c_{G_i^{sub},st}) + \delta \cdot (A^m_{cn_j,st} - R^m_{G_i^{sub},st}) \\ & + (1-\gamma-\delta) \cdot (A^{io}_{cn_j,st} - R^{io}_{G_i^{sub},st}), \end{aligned} \tag{15}$$

where γ and δ are the weights used to balance CPU, memory and I/O resources, $\gamma, \delta \in [0,1]$ and $\gamma + \delta < 1$. The values of γ and δ are determined based on resource usage characteristics. For example, if CPU is a bottleneck while memory and I/O are sufficient, γ can be set to 1 and δ to 0. Conversely, if memory becomes more constrained, γ should decrease and δ should increase accordingly.

An example of subgraph scheduling is shown in Fig. 6. As previously mentioned, multiple compute nodes may satisfy the resource requirements of a subgraph, and the compute node with the highest fitness factor is selected.

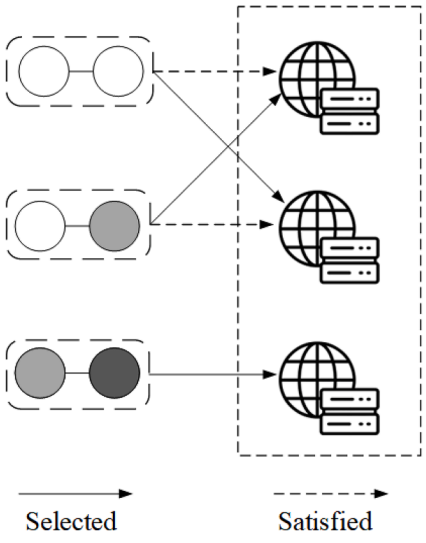

Fig. 6. Example of task scheduling at subgraph level.

5. Ra-Stream: architecture and algorithms

Based on the theoretical analysis presented earlier, we propose a fine-grained task scheduling strategy for resource auto-scaling called Ra-Stream, implemented on the Apache Storm platform. In this section, we provide an overview of the strategy, including its system architecture and the algorithms used for subgraph partitioning, resource scaling, and task scheduling. Since Apache Storm, Apache Flink, and Spark Streaming all utilize a master-slave architectural design, and rely on dynamic task allocation across distributed compute nodes, our scheduling strategy is theoretically applicable to the other two stream processing platforms.

Although some components of Ra-Stream employ standard techniques, our innovation lies in the introduction of a novel subgraph partitioning component that achieves a balanced communication-intensive subgraph partitioning. Furthermore, to address the issues of resource

overload and waste caused by fluctuations in data stream rates, we propose an innovative resource scaling algorithm that automatically adjusts computational resources based on current data stream rates, ensuring efficient utilization and conservation of computational resources. These methods provide a new approach to scheduling strategies in distributed stream computing systems.

5.1. System architecture

As shown in Fig. 7, the architecture of Ra-Stream consists of four main components: **Scheduling trigger**, **Subgraph partition**, **Resource scaling**, and **Data monitor**.

Scheduling trigger reads the current operational status of the stream application from the Database and determines whether to trigger a new scheduling event.

Subgraph partition divides the stream application into multiple balanced communication-intensive subgraphs according to the communication relationships among tasks in the stream application.

Resource scaling adjusts the subgraph partitioning results to determine the minimum number of compute nodes required, ensuring that no compute node is overloaded or resources are wasted after scheduling.

Data monitor is responsible for the real-time collection of resource utilization data from all compute nodes, as well as the resource requirements of tasks, data stream rates, and the volume of data transferred between tasks.

The Scheduler deploys tasks onto compute nodes in a fine-grained manner to minimize communication cost.

5.2. Subgraph partitioning

The objective of subgraph partitioning is to divide the stream application into multiple balanced communication-intensive subgraphs. Inspired by the principles of the simulated annealing algorithm [37], known for its effectiveness in global optimization, we develop a intelligent subgraph partitioning algorithm.

Given that it is not possible to determining the minimum number of compute nodes required to execute the stream application before scheduling it to the cluster, we set the number of subgraphs to be equal to the number of compute nodes in the cluster, that is, $k = n$. To achieve balanced subgraphs (refer to Eq. (9)), we incorporate the variance of the internal weight sums of the subgraphs, σ_W, into our objective function, as shown in Eq. (16):

$$\sigma_W = \sqrt{\frac{1}{k}\sum_{i=1}^{k}[W_{int}(G_i^{sub}) - \overline{W_{int}(G^{sub})}]^2}, \tag{16}$$

subject to:

$$\overline{W_{int}(G^{sub})} = \frac{1}{k}\sum_{i=1}^{k} W_{int}(G_i^{sub}), \tag{17}$$

where $\overline{W_{int}(G^{sub})}$ is the average weight of all subgraphs.

Due to the variations in data stream rates, edge weights differ, making it impractical to rely solely on absolute values in the objective function. Instead, we use the ratio of cut edge weights to the total weights of edges in G, as illustrated in Eq. (18):

$$r = \frac{W_{cut}(G)}{W_{cut}(G) + W_{int}(G^{sub})} = \frac{\sum_{e_{i,k,\ j,l} \in E_{cut}(G)} w(e_{i,k,\ j,l})}{\sum_{e_{i,k,\ j,l} \in E(G)} w(e_{i,k,\ j,l})}, \tag{18}$$

By combining Eqs. (16) and (18), we obtain our objective function $f(x)$, described in Eq. (19):

$$f(x) = \epsilon \cdot r + (1 - \epsilon) \cdot \sigma_W, \tag{19}$$

where x is the current subgraph partitioning scheme, ϵ is the weight factor that combines the ratio of cut edge weights r and the variance of internal weight sums of the subgraphs σ_W. In extreme cases, if the tuple transmission rates between tasks in the stream application are all equal, there will be no unbalanced subgraph partitioning schemes, and thus ϵ should be set to 1. Conversely, the value of ϵ should decrease as the differences in tuple transmission rates between tasks increase. After defining the objective function, we outline the algorithmic workflow for the initial subgraph partitioning phase in Algorithm 1.

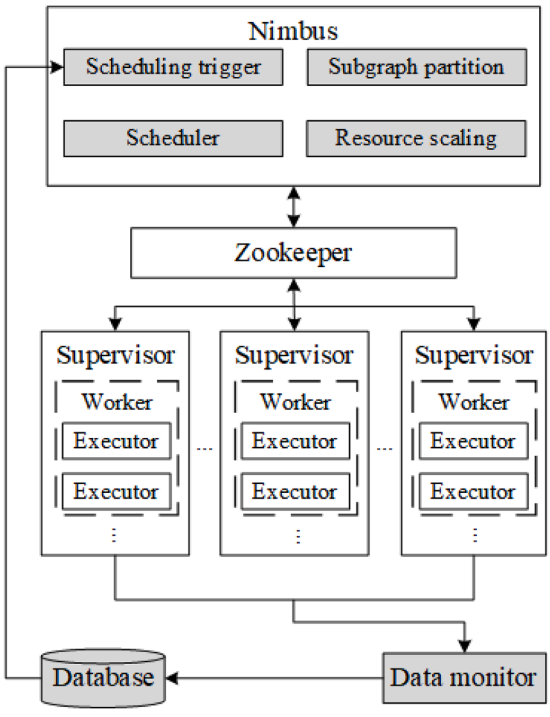

Fig. 7. Ra-Stream's architecture.

Algorithm 1: Subgraph partitioning.

Input: G, k, n, $Tr(v_{i,k}, v_{j,l})$, max_iter, T_0, ζ, T_f.
Output: Subgraph partitioning scheme X.

```
k ← n;
Initialize partition G to k subgraphs and denote as x_0;
X ← x_0;
Calculate f(x_0) according to Eq. (19) and denote as f(X);
Initialize current temperature T ← T_0;
Set the current iteration count current_iter ← 0;
while T > T_f do
    while current_iter < max_iter do
        Generate a neighboring solution x_i by moving a
          random task to a different subgraph;
        Calculate the f(x_i) according to Eq. (19);
        if f(x_i) ≤ f(X) then
            X ← x_i;
        end
        else
            Accept x_i with probability p_i according to Eq. (20);
        end
        current_iter ← current_iter + 1;
    end
    current_iter ← 0; T ← ζ · T;
end
return X.
```

The input to Algorithm 1 includes the stream application G, the number of subgraphs k, the number of compute nodes n, tuple transmission rate between tasks $Tr(v_{i,k}, v_{j,l})$, maximum number of iterations

max_iter, initial temperature T_0, cooling rate ζ, and final temperature T_f. The output is a balanced communication-intensive subgraph partitioning scheme X. Steps 1 to 6 prepare the algorithm for execution. Steps 8 to 17 form the inner loop, where new solutions are generated, the objective function is calculated, and the solution is accepted or rejected. Steps 7 to 20 form the outer loop, representing the cooling phase. The current temperature is set to start from the initial temperature T_0, and is repeatedly multiplied with the cooling rate ζ until the current temperature is less than or equal to the final temperature T_f. In steps 4 and 10, user can choose the value of ϵ in Eq. (19). The time complexity of Algorithm 1 is $O(log_{\frac{1}{\zeta}}(\frac{T_0}{T_f}) \cdot m)$, where m is the maximum number of iterations.

As Algorithm 1 operates, new subgraph partitioning schemes are progressively explored in the neighborhood of the current scheme. Each new scheme is evaluated against the current one using Eq. (19), and based on the Metropolis acceptance criterion, inferior subgraph schemes are accepted with a probability p_i, calculated using Eq. (20):

$$p_i = \exp^{(\frac{f(x_i)-f(X)}{T})}, \tag{20}$$

At high initial temperatures, inferior schemes are accepted with higher probability, promoting extensive exploration of the solution space and mitigating the risk of local convergence. As the temperature gradually decreases according to the cooling rate ζ, the probability of accepting poorer inferior schemes diminishes, and the search increasingly focuses on the neighborhoods of better partitioning solutions. Ultimately, as the temperature approaches the final value T_f, the algorithm converges near the globally optimal subgraph partitioning scheme X.

5.3. Resource scaling

To facilitate efficient utilization of computational resources, we evaluate the resource demands of subgraphs prior to task scheduling. This allows for the selection of compute nodes that satisfy these requirements, ensuring high system performance at minimal resource cost. For this purpose, we define two thresholds for each compute node: the underload threshold T_{under} and the overload threshold T_{over}. We aim to achieve a balanced state where, after scheduling subgraphs to compute nodes, the resource utilization of each node lies within the range T_{under} and T_{over}. To identify the subgraphs requiring adjustment, we introduce a Boolean factor $f^{adj}_{G_i^{sub}}$, which determines whether a subgraph G_i^{sub} needs adjustment according to its resource requirement $R_{G_i^{sub}}$. This is defined in Eq. (21):

$$f^{adj}_{G_i^{sub}} = \begin{cases} True, & (R_{G_i^{sub}} < T_{under}) \vee (R_{G_i^{sub}} > T_{over}), \\ False, & T_{under} < R_{G_i^{sub}} < T_{over}, \end{cases} \tag{21}$$

when $T_{under} < R_{G_i^{sub}} < T_{over}$, the subgraph can be directly scheduled to a compute node. However, when $R_{G_i^{sub}} < T_{under}$ or $R_{G_i^{sub}} > T_{over}$, the subgraph needs to be adjusted before scheduling. The resource scaling process is outlined in Algorithm 2.

The input to Algorithm 2 includes the subgraph partitioning scheme X, resource requirements of subgraphs $R_{G_i^{sub}}$, and the thresholds T_{under} and T_{over}. The output is the resource scaling scheme RS, comprising adjusted subgraphs. The number of subgraphs in RS indicates the minimum required compute nodes. Steps 4 to 18 handle adjustments for subgraphs with resource requirement below T_{under}, including subgraph merging (Steps 6-10) and inter-subgraph task adjustment (Steps 13-15). Steps 22-27 handle the adjustment of subgraphs whose resource demands exceed T_{over} through inter-subgraph task adjustment. The time complexity of Algorithm 2 is $O(m \cdot logn)$, where m is the number of tasks in a subgraph and n is the number of subgraphs in X.

Algorithm 2: Resource scaling.

Input: X, $R_{G_i^{sub}}$, T_{under}, T_{over}.
Output: Resource scaling scheme RS.

1 Sort the subgraphs in X in ascending order based on their resource requirements;
2 **while** $X \neq \emptyset$ **do**
  /* If target subgraph's resource requirement is less than underload threshold. */
3 **if** $R_{G_i^{sub}} < T_{under}$ **then**
4 $G^{sub}_{match} \leftarrow \emptyset$;
5 Identify a matched subgraph G_j^{sub} for merging via binary search;
6 $G^{sub}_{match} \leftarrow G_j^{sub}$;
  /* If merging candidate is found, merge it with the target. */
7 **if** $G^{sub}_{match} \neq \emptyset$ **then**
8 $G^{sub}_{temp} \leftarrow G_i^{sub} + G_j^{sub}$;
9 Put G^{sub}_{temp} into RS;
10 Remove G_i^{sub} and G_j^{sub} from X;
11 **end**
  /* If no merging candidate is found, find a task-adjusting candidate and move its task into the target. */
12 **else**
13 **while** $R_{G_i^{sub}} < T_{under}$ **do**
14 Identify a matched subgraph G_j^{sub} for task adjusting via binary search;
15 Select task $v_{i,k}$ with minimal impact on the weight of G_j^{sub};
16 Put $v_{i,k}$ into G_i^{sub} from G_j^{sub};
17 **end**
18 Put G_i^{sub} into RS;
19 Remove G_i^{sub} from X;
20 **end**
21 **end**
  /* If target subgraph's resource requirement is greater than overload threshold */
22 **else if** $R_{G_i^{sub}} > T_{over}$ **then**
23 **while** $R_{G_i^{sub}} > T_{over}$ **do**
24 Identify a matched subgraph G_j^{sub} for task adjusting via binary search;
25 Identify task $v_{i,k}$ with minimal impact on the weight of G_i^{sub};
26 Put $v_{i,k}$ into G_j^{sub} from G_i^{sub};
27 **end**
28 Put G_i^{sub} into RS;
29 Remove G_i^{sub} from X;
30 **else**
31 Put G_i^{sub} into RS;
32 **end**
33 **end**
34 **return** RS.

5.4. Fine-grained task scheduling

As described in Section 3.2, when we schedule the task set to compute nodes at the subgraph level, inter-node communication is effectively converted into intra-node communication to reduce communication costs. However, within a subgraph, the communication traffic between tasks may vary, leading to the presence of communication-intensive task pairs. To address this, our scheduling algorithm consists

of two components: scheduling subgraphs to compute nodes (coarse-grained) and scheduling tasks within the processes of compute nodes (fine-grained). Communication-intensive task pairs within a subgraph are allocated to the same process on a compute node to further minimize communication costs.

For tasks $v_{i,k}$ and $v_{j,l}$, the communication cost within a process is negligible compared to costs between nodes or processes. Therefore, we set intra-process communication cost to 0. To quantify the cost benefits of co-locating communication-intensive tasks, we define the cost savings from transforming inter-node to intra-node communication over a statistical time frame st using Eq. (22), and from inter-process to intra-process using Eq. (23):

$$C^{s,st}_{v_{i,k},v_{j,l}} = Tr(v_{i,k}, v_{j,l}) \cdot (C^{com}_{CN} - C^{com}_{CP}) \cdot st, \tag{22}$$

$$C^{s,st}_{v_{i,k},v_{j,l}} = Tr(v_{i,k}, v_{j,l}) \cdot C^{com}_{CP} \cdot st, \tag{23}$$

where C^{com}_{CN} and C^{com}_{CP} represent the per-tuple communication costs across nodes and across processes, respectively.

Before scheduling subgraphs to compute nodes, it is essential to consider the available computational resources of the current compute nodes. As previously mentioned, multiple compute nodes may satisfy the resource requirement of a subgraph. To handle this, we select compute nodes based on Eq. (14), ensuring that the selected compute node retains sufficient available resources to handle sudden increases in resource requirement caused by spikes in data streams. The detailed steps of our scheduling algorithm are outlined in Algorithm 3.

Algorithm 3: Fine-grained task scheduling.

Input: RS, CN, $T(G^{sub}_i)$, $Tr(v_{i,k}, v_{j,l})$.
Output: Scheduling scheme SS.

1 **for** G^{sub}_i *in* RS **do**
2 Find compute nodes that satisfy the resource requirement $R_{G^{sub}_i}$ of subgraph G^{sub}_i according to Eq. (13);
3 Calculate $f^{fit}_{G^{sub}_i,cn_j}$ for these compute nodes based on Eq. (14);
4 Schedule G^{sub}_i to the compute node with highest value of $f^{fit}_{G^{sub}_i,cn_j}$;
5 **while** *the number of tasks in* $T(G^{sub}_i) > 1$ **do**
6 Place the task pair $(v_{i,k}, v_{j,l})$ with the highest $Tr(v_{i,k}, v_{j,l})$ in the same process;
7 Remove $v_{i,k}$ and $v_{j,l}$ from $T(G^{sub}_i)$;
8 **end**
9 Remove G^{sub}_i from RS;
10 **end**
11 **return** SS.

The input to Algorithm 3 includes the resource scaling RS, compute node set CN, task set within each subgraph $T(G^{sub}_i)$, and the tuple communication rate between tasks $Tr(v_{i,k}, v_{j,l})$. The output is the fine-grained task scheduling scheme SS for application G. Steps 2 to 4 describe the coarse-grained scheduling process, which involves scheduling tasks to compute nodes. Steps 6 to 7 describe the fine-grained scheduling process, which involves placing communication-intensive task pairs in the same process. The time complexity of Algorithm 3 is $O(m \cdot n)$, where m is the number of subgraphs in RS and n is the number of tasks within a subgraph.

5.5. System implementation

The time complexity analysis for each algorithm shows that the overhead introduced by Ra-Stream remains within tolerable limits. While the Ra-Stream modules do introduce some time overhead, the impact is minimal and does not degrade system performance. These modules are designed to optimize system performance, and their effectiveness is validated through the experiments in Section 6.

Ra-Stream primarily utilizes Storm's built-in interfaces, `IMetric` and `IMetricConsumer`, to track and collect runtime information. This includes communication traffic between tasks in the stream application, resource loads on compute nodes, and available computational resources. The resource load on compute nodes can be monitored using Linux system interface commands. Ra-Stream's scheduler is implemented via Storm's built-in interface, `IScheduler`.

6. Performance evaluation

According to [38], we evaluate the proposed Ra-Stream in a real-world distributed computing environment. First, we present our experimental setup and parameter configurations. Next, we outline the datasets and stream applications used for testing. Finally, we provide a thorough analysis of the results.

6.1. Experimental setup

Our distributed compute cluster consists of 15 machines, including 1 management node and 14 compute nodes. Each compute node is powered by an Intel(R) Xeon(R) X5650 CPU (dual-core, 2.4 GHz), equipped with 2 GB of RAM, and a 100 Mbps Ethernet interface card. The management node is responsible for running Nimbus and Zookeeper to maintain the overall operation of the cluster, while the compute nodes run Supervisor to handle stream applications. Each compute node is configured to deploy a maximum of two Workers, with each Worker running up to two tasks. For clarity, we exhibit the software configurations in Table 3, and the parameter configuration of Ra-Stream in Table 4.

During the experiment, we use the public dataset Alibaba Tianchi [39] as the data source for the real-time word counting application (WordCount application), and use the public dataset provided by Backblaze [40] as the data source for the DEBS 2024 Grand Challenge: Telemetry data for hard drive failure prediction and predictive maintenance [41]. The topologies of the two applications are illustrated in Figs. 2 and 8, respectively.

In the DEBS 2024 topology (Fig. 8), 'Data Source' acts as the Spout, sending real-time data downstream. 'Data Filter' removes invalid or incomplete data. 'Event Detection' identifies specific event patterns. 'Aggregation' computes real-time statistics using a sliding window, and the aggregated results are stored in the database by 'Data Storage'. 'Anomaly Detection' identifies anomalies in the data stream and sends these events to 'Alert System'. Finally, 'Alert System' generates alert notifications and disseminates them through a message queue. State migration is involved

Table 3
Software configuration of experimental environment.

Software	Version
OS	Ubuntu 20.04 64 bit
Apache Storm	Apache-storm-2.1.0
JDK	jdk-8u171-linux-x64
Apache Zookeeper	Apache-Zookeeper-3.5.7
Python	Python 3.8.3
MySQL	MySQL-8.0.40
Apache Kafka	kafka-2.12-3.0.0

Table 4
Ra-Stream's parameter configuration.

Parameter	Value	Parameter	Value
α	0.50	β	0.35
ϵ	0.70	ζ	0.99
γ	0.50	δ	0.35
T_{under}	0.60	T_{over}	0.75

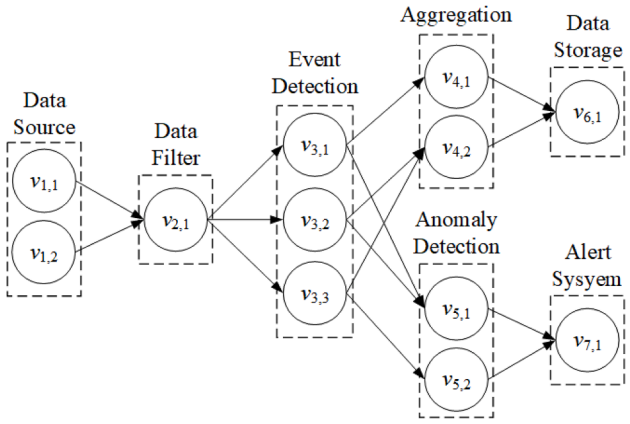

Fig. 8. Topology of DEBS 2024.

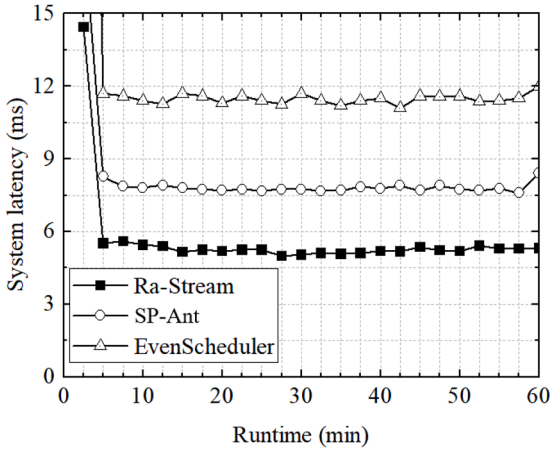

Fig. 9. System latency of WordCount under stable stream rates.

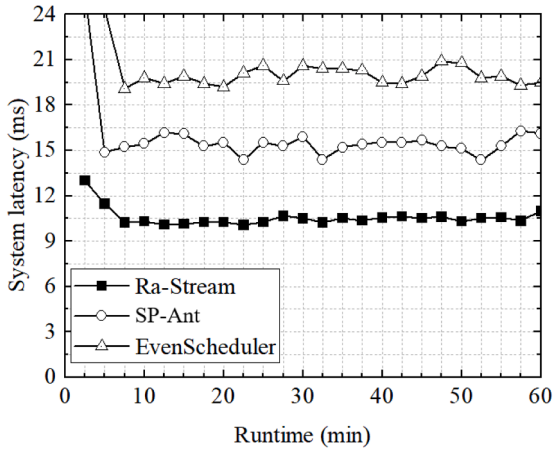

Fig. 10. System latency of DEBS 2024 under stable stream rates.

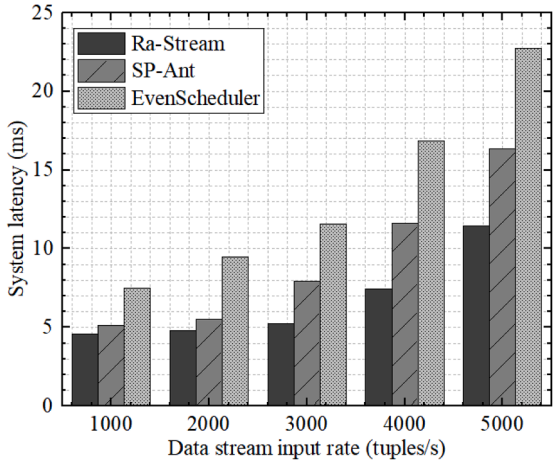

Fig. 11. System latency of WordCount under increasing stream rates.

during the task scheduling [42]. In this experiment, we employ Storm's check-point mechanism for state management [43].

6.2. System latency

System latency is defined as the time elapsed from the moment a tuple enters the system until it is fully processed. We assess the system latency of Ra-Stream under varying data stream rates and compare it with the state-of-the-art solutions, EvenScheduler [44] and SP-Ant [21]. EvenScheduler is the default scheduler in Storm and has been widely referenced in prior studies as a baseline [26,27,31]. SP-Ant, a recent and open-source scheduling framework, reduces communication overhead and achieves competitive system performance by dynamically adjusting operator assignments based on the computational capabilities of compute nodes.

Under a stable data stream rate of 3000 tuples/s, Ra-Stream demonstrates significantly lower system latency compared to the other two solutions. As shown in Fig. 9, the average system latencies of WordCount are 5.26 ms, 7.81 ms and 11.57 ms for Ra-Stream, SP-Ant and EvenScheduler, respectively. SP-Ant places communication-intensive tasks on the same computation node without considering the resource utilization of that node, resulting in node overload and consequently higher system latency.

Similarly, under the same stable data rate, Fig. 10 shows the average system latency of DEBS 2024 application. EvenScheduler and SP-Ant yield latency of 20.01 ms and 15.80 ms, respectively, while Ra-Stream achieves an average system latency of only 10.61 ms. Ra-Stream demonstrates greater stability compared to the apparent latency fluctuations observed with EvenScheduler and SP-Ant. This improved stability is attributed to Ra-Stream's ability to prevent compute node overloads, ensuring that each task has sufficient resources to process tuples.

Under increasing data stream rates, Ra-Stream consistently demonstrates reduced system latency compared to SP-Ant and EvenScheduler. We set an initial data stream rate of 1000 tuples/s. After system metrics such as latency stabilize, we collect performance data for 10 min. The data stream rate is then increased in 1000 tuples/s increments, with stabilization and data collection occurring at each step, up to 5000 tuples/s. As shown in Figs. 11 and 12, the average system latencies of WordCount and DEBS 2024 increase for all solutions as the data stream rate grows. However, Ra-Stream experiences a smaller increase in latency compared to SP-Ant and EvenScheduler, maintaining a performance advantage.

In summary, Ra-Stream demonstrates excellent system latency, whether the data stream rate is stable or increasing. This is because Ra-Stream effectively prevents overload situations in the cluster's compute nodes, guaranteeing that each task in the stream application has ample computational resources to steadily process tuples. In addition, Ra-Stream minimizes communication costs between tasks, further reducing system latency.

6.3. System throughput

Maximum system throughput is defined as the highest data stream rate that a system can steadily process without failure. In stream computing systems, if any task within the stream application fails as the data stream rate increases, the corresponding data stream rate is identified as the system's maximum throughput. To determine the maximum

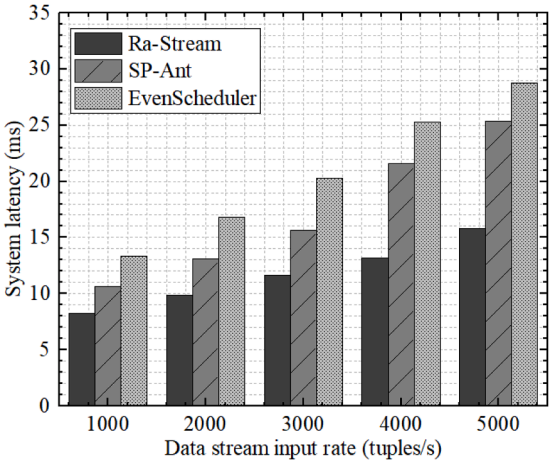

Fig. 12. System latency of DEBS 2024 under increasing stream rates.

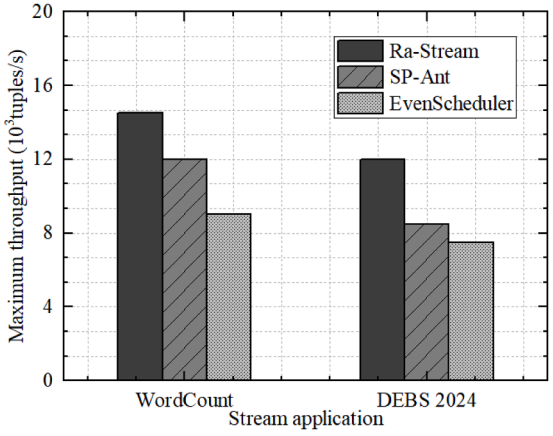

Fig. 13. Maximum throughput of WordCount and DEBS 2024 under Ra-Stream, SP-Ant and EvenScheduler.

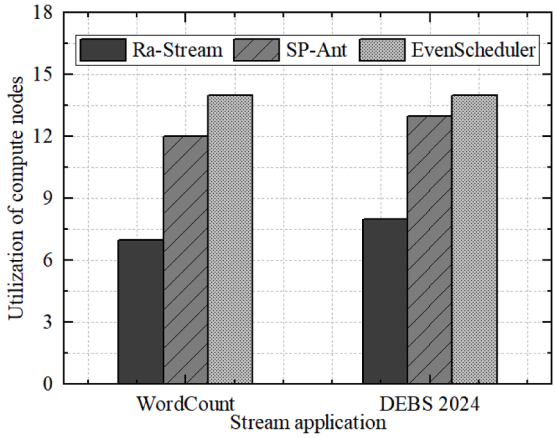

Fig. 14. Number of compute nodes utilized in WordCount and DEBS 2024 under stable stream rates.

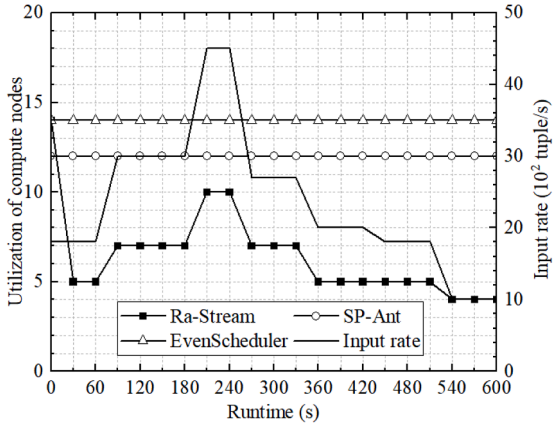

Fig. 15. Number of compute nodes utilized in WordCount under fluctuating stream rates.

throughput for Ra-Stream, SP-Ant, and EvenScheduler, we deploy them on our cluster and run stream applications under incremental data rates. For fairness, the computational resource configurations, datasets, and stream applications used are kept identical across all experiments.

Under an increasing data stream rate with increments of 500 tuples/s, Ra-Stream demonstrates a higher maximum throughput than the others. As shown in Fig. 13, for WordCount and DEBS 2024, Ra-Stream achieves maximum throughputs of 14,500 tuples/s and 12,000 tuples/s, respectively. These values represent a significant improvement over EvenScheduler and SP-Ant.

The superior performance of Ra-Stream is attributed to its ability to automatically configure the scheduling scheme for tasks based on the current data stream rate, thereby preventing task failures caused by insufficient computational resources. Before generating the scheduling scheme, Ra-Stream systematically adjusts tasks among subgraphs. This approach ensures that, once tasks are scheduled to compute nodes at the subgraph level, each task is allocated sufficient resources to execute.

6.4. Resource utilization

Resource utilization refers to the proportion of utilized resources relative to the total resources at runtime. The focus on runtime resource utilization for stream applications is a prevalent topic in current research. For example, one key consideration is how to achieve optimal system performance with a reduced number of compute nodes. However, the number of compute nodes used does not directly reflect the resource utilization within an individual compute node. For instance, excessively low resource utilization in a compute node indicates significant resource wastage, which is clearly undesirable. To address this, we employ two metrics to assess Ra-Stream's resource utilization: the number of compute nodes used within the cluster and the average resource utilization within compute nodes. The average resource utilization consists of CPU, memory and I/O resources, which can be calculated by Eq. (6).

Under a stable data stream rate of 3000 tuples/s, Ra-Stream can adjust the number of compute nodes to adapt to the data rate. As shown in Fig. 14, for the two stream applications, WordCount and DEBS 2024, Ra-Stream utilizes 7 and 8 compute nodes, respectively. These figures are significantly fewer than the number of compute nodes utilized by SP-Ant and EvenScheduler.

Under fluctuating data stream rates, Ra-Stream automatically adjusts the utilization of compute nodes to accommodate current data stream conditions. To simulate real-world fluctuating workloads, we define a time-varying data stream profile. For example, the data stream rate increases from 1800 tuples/s to 3000 tuples/s at 60 s, and decreases from 2700 tuples/s to 2000 tuples/s at 330 s. This enables the observation of Ra-Stream's dynamic resource scaling and performance adaptability under fluctuating load conditions. In our experiment, the peak data stream rate is set to 4500 tuples/s at 210 s. As shown in Figs. 15 and 16, Ra-Stream dynamically adjusts the number of compute nodes for both WordCount and DEBS 2024 applications in response to changing data rates. Initially, Ra-Stream's number of compute nodes matches that of EvenScheduler. Subsequently, Ra-Stream generates a new scheduling plan suitable for the current data rate 1800 tuples/s at runtime 30s,

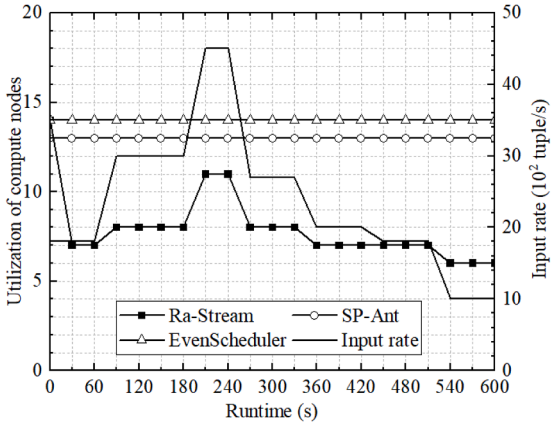

Fig. 16. Number of compute nodes utilized in DEBS 2024 under fluctuating stream rates.

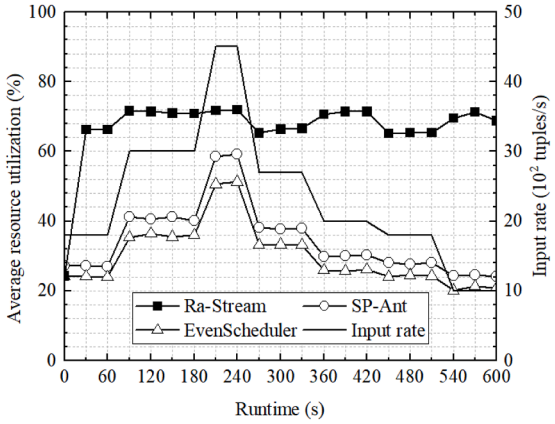

Fig. 17. Average resource utilization in WordCount under fluctuating stream rates.

adjusting the node number (from 14 to 5 in WordCount and from 14 to 7 in DEBS 2024).

Throughout the runtime, Ra-Stream continues to adapt to the fluctuating streams. For example, when the data rate increases from 3000 tuples/s to 4500 tuples/s at runtime 210s, Ra-Stream scales up the node number (from 7 to 10 in WordCount, and from 8 to 11 in DEBS 2024) to ensure stable system operation. Conversely, when the rate decreases from 4500 tuples/s to 2700 tuples/s at runtime 270s, Ra-Stream reduces the node number (from 10 to 7 in WordCount, and from 11 to 8 in DEBS 2024), optimizing resource utilization.

Under fluctuating data stream rates, Ra-Stream also achieves more efficient average resource utilization. As shown in Figs. 17 and 18, whether running WordCount or DEBS 2024, Ra-Stream consistently achieves higher and more stable resource utilization compared to SP-Ant and EvenScheduler. In contrast, SP-Ant and EvenScheduler exhibit inefficient resource utilization, which fluctuates with changes in data rates, leading to wasted resources.

In summary, Ra-Stream not only scales compute node resources automatically in response to data stream fluctuations, ensuring the stable operation of stream applications with a minimum number of nodes, but also achieves more efficient average resource utilization, greatly reducing resource waste. Ra-Stream demonstrates a substantial improvement in resource utilization.

7. Conclusion and future work

In scenarios involving fluctuating data streams, minimizing system latency, reducing resource costs, and ensuring efficient resource

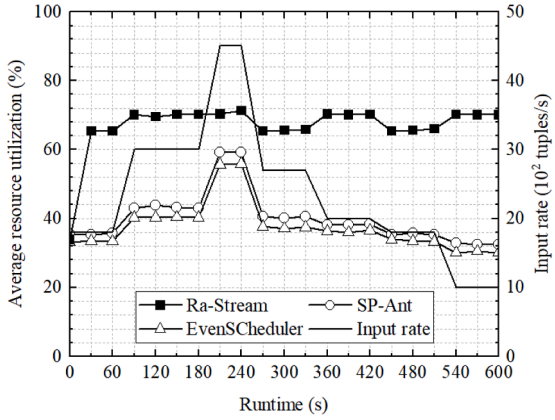

Fig. 18. Average resource utilization in DEBS 2024 under fluctuating stream rates.

utilization are critical challenges in current stream computing research. Achieving these objectives requires a scheduling strategy capable of sensing changes in data flow rates and accounting for the dependencies among tasks within stream applications. Such a strategy should automatically adjust computational resources in response to variations in data streams, while minimizing communication costs through effective task deployment.

To meet these requirements, this paper proposes a task scheduling strategy with automated resource scaling designed to handle fluctuating data streams. Ra-Stream achieves strong system performance in fluctuating data stream scenarios with reduced resource costs. Compared to state-of-the-art approaches, Ra-Stream reduced system latency by approximately 36.37 % to 47.45 %, increased system maximum throughput by around 26.2 % to 60.55 %, and saves approximately 40 % to 46.25 % in resource utilization. Despite these advantages, there remain areas for further enhancement, including support for greater task parallelism, integration of energy-aware scheduling for reduced power consumption, and improved scalability across other stream processing frameworks.

In the future, we aim to further explore the following areas:

(1) Task parallelism: Investigating the parallelism of tasks within stream applications to further reduce system latency.
(2) Energy consumption: Integrating energy consumption metrics of compute clusters to achieve additional economic and environmental benefits.

CRediT authorship contribution statement

Yinuo Fan: Conceptualization, Methodology, Validation, Writing – original draft; **Dawei Sun:** Methodology, Writing – review & editing, Funding acquisition; **Minghui Wu:** Validation, Writing – review & editing; **Shang Gao:** Formal analysis, Investigation, Writing – review & editing; **Rajkumar Buyya:** Methodology, Writing – review & editing.

Data availability

Data will be made available on request.

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.

Acknowledgement

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.

References

[1] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, The VLDB J. 33 (2) (2024) 507–541.
[2] Z. Wen, R. Yang, B. Qian, Y. Xuan, L. Lu, Z. Wang, H. Peng, J. Xu, A.Y. Zomaya, R. Ranjan, JANUS: latency-aware traffic scheduling for IoT data streaming in edge environments, IEEE Trans. Serv. Comput. 16 (6) (2023) 4302–4316.
[3] B. Pishgoo, A.A. Azirani, B. Raahemi, A hybrid distributed batch-stream processing approach for anomaly detection, Inf. Sci. 543 (2021) 309–327.
[4] Y. Sasaki, A survey on IoT big data analytic systems: current and future, IEEE Internet Things J. 9 (2) (2021) 1024–1036.
[5] S. Zhang, J. Soto, V. Markl, A survey on transactional stream processing, The VLDB J. 33 (2) (2024) 451–479.
[6] M.D. de Assuncao, A. da Silva Veith, R. Buyya, Distributed data stream processing and edge computing: a survey on resource elasticity and future directions, J. Netw. Comput. Appl. 103 (2018) 1–17.
[7] X. Liu, R. Buyya, Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions, ACM Comput. Surv. (CSUR) 53 (3) (2020) 1–41.
[8] N. Tantalaki, S. Souravlas, M. Roumeliotis, A review on big data real-time stream processing and its scheduling techniques, Int. J. Parallel Emergent Distrib. Syst. 35 (5) (2020) 571–601.
[9] G. Van Dongen, D. Van den Poel, Evaluation of stream processing frameworks, IEEE Trans. Parallel Distrib. Syst. 31 (8) (2020) 1845–1858.
[10] M. Barika, S. Garg, A.Y. Zomaya, R. Ranjan, Online scheduling technique to handle data velocity changes in stream workflows, IEEE Trans. Parallel Distrib. Syst. 32 (8) (2021) 2115–2130.
[11] S. Gurusamy, R. Selvaraj, Resource allocation with efficient task scheduling in cloud computing using hierarchical auto-associative polynomial convolutional neural network, Expert Syst. Appl. 249 (2024) 123554.
[12] S. Wang, G.-s. Zeng, Two-stage scheduling for a fluctuant big data stream on heterogeneous servers with multicores in a data center, Cluster Comput. 27 (2) (2024) 1581–1597.
[13] M. Barika, S. Garg, A. Chan, R.N. Calheiros, Scheduling algorithms for efficient execution of stream workflow applications in multicloud environments, IEEE Trans. Serv. Comput. 15 (02) (2022) 860–875.
[14] P. Ntumba, N. Georgantas, V. Christophides, Adaptive scheduling of continuous operators for IoT edge analytics, Future Gener. Comput. Syst. 158 (2024) 277–293.
[15] X. Fu, B. Tang, F. Guo, L. Kang, Priority and dependency-based DAG tasks offloading in fog/edge collaborative environment, in: 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 2021, pp. 440–445.
[16] A. Al-Sinayyid, M. Zhu, Job scheduler for streaming applications in heterogeneous distributed processing systems, J. Supercomput. 76 (12) (2020) 9609–9628.
[17] L. Eskandari, J. Mair, Z. Huang, D. Eyers, I-Scheduler: iterative scheduling for distributed stream processing systems, Future Gener. Comput. Syst. 117 (2021) 219–233.
[18] 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.
[19] H. Röger, R. Mayer, A comprehensive survey on parallelization and elasticity in stream processing, ACM Comput. Surv. (CSUR) 52 (2) (2019) 1–37.
[20] M. Mortazavi-Dehkordi, K. Zamanifar, Efficient deadline-aware scheduling for the analysis of big data streams in public cloud, Cluster Comput. 23 (1) (2020) 241–263.
[21] M. Farrokh, H. Hadian, M. Sharifi, A. Jafari, SP-ant: an ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters, Expert Syst. Appl. 191 (2022) 116322.
[22] H. Li, J. Xia, W. Luo, H. Fang, Cost-efficient scheduling of streaming applications in apache flink on cloud, IEEE Trans. Big Data 9 (4) (2022) 1086–1101.
[23] A. Brown, S. Garg, J. Montgomery, K.C. Ujjwal, Resource scheduling and provisioning for processing of dynamic stream workflows under latency constraints, Future Gener. Comput. Syst. 131 (2022) 166–182.
[24] A. Momtaz, R. Medhat, B. Bonakdarpour, Resource optimization of stream processing in layered internet of things, in: 2023 42nd International Symposium on Reliable Distributed Systems (SRDS), 2023, pp. 221–231.
[25] Y. Mao, J. Zhao, S. Zhang, H. Liu, V. Markl, Morphstream: adaptive scheduling for scalable transactional stream processing on multicores, Proc. ACM Manage. Data 1 (1) (2023) 1–26.
[26] P. Kang, S.U. Khan, X. Zhou, P. Lama, High-throughput real-time edge stream processing with topology-aware resource matching, in: 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2024, pp. 385–394.
[27] R. Ecker, V. Karagiannis, M. Sober, S. Schulte, Latency-aware placement of stream processing operators in modern-day stream processing frameworks, J. Parallel Distrib. Comput. 199 (2025) 105041.
[28] Y. Li, H. Jiang, Y. Shen, Y. Fang, X. Yang, D. Huang, X. Zhang, W. Zhang, C. Zhang, P. Chen, et al., Towards general and efficient online tuning for spark, Proc. VLDB Endow. 16 (12) (2023) 3570–3583.
[29] X. Huang, Z. Shao, Y. Yang, POTUS: predictive online tuple scheduling for data stream processing systems, IEEE Trans. Cloud Comput. 10 (4) (2020) 2863–2875.
[30] H. Chen, F. Zhang, H. Jin, PStream: a popularity-aware differentiated distributed stream processing system, IEEE Trans. Comput. 70 (10) (2020) 1582–1597.
[31] W. Li, D. Liu, K. Chen, K. Li, H. Qi, Hone: mitigating stragglers in distributed stream processing with tuple scheduling, IEEE Trans. Parallel Distrib. Syst. 32 (08) (2021) 2021–2034.
[32] H. Xu, P. Liu, S.T. Ahmed, D. Da Silva, L. Hu, Adaptive fragment-based parallel state recovery for stream processing systems, IEEE Trans. Parallel Distrib. Syst. 34 (8) (2023) 2464–2478.
[33] Y. Wang, Z. Tari, X. Huang, A.Y. Zomaya, A network-aware and partition-based resource management scheme for data stream processing, in: Proceedings of the 48th International Conference on Parallel Processing, 2019, pp. 1–10.
[34] R. Pathan, P. Voudouris, P. Stenström, Scheduling parallel real-time recurrent tasks on multicore platforms, IEEE Trans. Parallel Distrib. Syst. 29 (4) (2017) 915–928.
[35] H. Li, J. Wu, Z. Jiang, X. Li, X. Wei, Task allocation for stream processing with recovery latency guarantee, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 379–383.
[36] J. Jiang, Z. Zhang, B. Cui, Y. Tong, N. Xu, StroMAX: partitioning-based scheduler for real-time stream processing system, in: Database Systems for Advanced Applications: 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part II 22, 2017, pp. 269–288.
[37] A.G. Nikolaev, S.H. Jacobson, Simulated annealing, Handb. metaheuristics 146 (2010) 1–39. https://doi.org/10.1007/978-1-4419-1665-5_1
[38] 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), 2018, pp. 1507–1518.
[39] Aliyun, 2021, https://tianchi.aliyun.com/dataset/.
[40] Hard drive test data, 2025, https://www.backblaze.com/cloud-storage/resources/hard-drive-test-data.
[41] L. De Martini, J. Tahir, C. Doblander, S. Frischbier, A. Margara, The DEBS 2024 grand challenge: telemetry data for hard drive failure prediction, in: Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems, 2024, pp. 223–228.
[42] B. Del Monte, S. Zeuch, T. Rabl, V. Markl, Rethinking stateful stream processing RDMA, in: Proceedings of the 2022 International Conference on Management of Data, 2022, pp. 1078–1092.
[43] Y. Zhuang, X. Wei, H. Li, M. Hou, Y. Wang, Reducing fault-tolerant overhead for distributed stream processing with approximate backup, in: 2020 29th International Conference on Computer Communications and Networks (ICCCN), 2020, pp. 1–9.
[44] EvenScheduler. https://github.com/apache/storm/blob/v2.1.0/storm-server/src/main/java/org/apache/storm/scheduler/EvenScheduler.java.

Yinuo Fan is a postgraduate student at the School of Information Engineering, China University of Geosciences, Beijing, China. He received his Bachelor Degree from North China University of Science and Technology, Tangshan, China in 2023. His research interests include big data stream computing, distributed systems, and reinforcement learning.

Dawei Sun is a Professor in the School of Information Engineering, China University of Geosciences, Beijing, PR China. He received his PhD degree in computer science from Northeastern University, China in 2012, and conducted the Postdoctoral research in the department of computer science and technology at Tsinghua University, China in 2015. His current research interests include big data computing, cloud computing and distributed systems. In these areas, he has authored over 100 journal and conference papers.

Minghui Wu is a PhD student at the School of Information Engineering, China University of Geosciences, Beijing, China. He received his Bachelor Degree in Network Engineering from Zhengzhou University of Aeronautics, Zhengzhou, China in 2020. His research interests include big data stream computing, distributed systems, and blockchain.

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

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Quantum Cloud Computing and Distributed Systems (qCLOUDS) 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 850 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 173 with 160,500+ citations). He is among the world' s top 2 most influential scientists in distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. 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.