Future Generation Computer Systems 166 (2025) 107673

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Straggler mitigation via hierarchical scheduling in elastic stream computing systems

Minghui Wu [a], Dawei Sun [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] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Keywords:
Stream computing systems
Straggler mitigation
Load balancing
Stateful operators
Two-level router

ABSTRACT

Skewed data distribution leads to certain tasks or nodes handling much more data than others, thereby slowing down their execution speed and classifying them as stragglers. Existing solutions attempt to establish a well-balanced workload to mitigate stragglers by using either data stream grouping or task scheduling. This "one size fits all" approach only considers single-level requirements and fails to address the diverse needs of the system across multiple levels, ultimately limiting its performance. To address these issues and mitigate stragglers effectively, we propose a hierarchical collaborative strategy called Ms-Stream. It aims to balance the data stream workloads among tasks and maintain load difference among compute nodes within an acceptable range. This paper discusses this strategy from the following aspects: (1) Ms-Stream constructs models for topology, grouping, and resource, along with the formalization of problems, including data stream grouping, task subgraph partitioning, and task deployment. (2) Ms-Stream employs a lightweight two-level grouping method to support dynamic workload assignment for stateful tasks, selectively offloading resources from task stragglers to others. (3) Ms-Stream allocates communication-intensive tasks to the same group through the directed acyclic graph representations of streaming applications, concurrently ensuring the equitable distribution of computation-intensive tasks across groups. (4) Ms-Stream deploys task groups to compute nodes with varying resource capacities following the descending maximum padding priority rule for a balanced workload. Performance metrics such as system throughput and latency are evaluated with real-world streaming applications. Experimental results demonstrate the significant improvements made by Ms-Stream, reducing maximum system latency by 61% and increasing maximum throughput by more than 2x compared to existing state-of-the-art works.

1. Introduction

Skewed data distribution is a primary factor contributing to the issue of load imbalance, leading to certain tasks or nodes executing at a slower pace and consequently impacting the overall performance of distributed stream computing systems [1]. These stragglers not only increase system latency in data processing, degrading user experience, but also create a need for additional computing resources to handle peak loads, thereby raising operational costs [2]. Unpredictable variations in data size and velocity received by stream computing systems result in inherently unstable processing, causing fluctuations in the workload of tasks or nodes [3,4]. This, in turn, poses a greater challenge for systems in managing skewed data streams.

The skewed data distribution primarily results in under-utilization of resources, manifesting in two main aspects: (1) Data with similar characteristics are aggregated to the same tasks for processing, leading to significantly higher loads in certain tasks (i.e., task stragglers), thus impacting system response time [5]. (2) Inappropriately deployed tasks with unbalanced workloads by the scheduler can lead to the aggregation of high-load tasks onto the same compute nodes [6]. Consequently, certain nodes remain idle for extended periods, while others (i.e., node stragglers) are overwhelmed with processing an extensive amount of data [7], further exacerbating the wastage of computational resources. In such an environment, it is important for the system to employ a task-based and node-based load balancing strategy to optimize the system performance.

Load balancing approaches for distributed stream computing systems have been extensively investigated with the objective of optimizing system performance and resource allocation [8]. Many efforts

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

https://doi.org/10.1016/j.future.2024.107673
Received 31 March 2024; Received in revised form 2 December 2024; Accepted 6 December 2024
Available online 14 December 2024

focus on refining data stream grouping techniques to mitigate stragglers in stateful tasks, which arise due to uneven data distribution and varying state update frequencies between tasks. For example, a popularity-aware differentiated stream computing system [9] employs shuffle grouping to allocate hotkeys, identified through a lightweight probabilistic counting scheme, while using key grouping for less frequent keys. An online predictive scheduling scheme [10] steers data streams in a distributed manner to balance workload across tasks and minimize data stream processing response time. Novel algorithms [11] for grouping data streams have also been proposed, which adapt to changing network bandwidth to reduce the high state migration costs associated with frequent data stream rescheduling.

While these methods have proven effective in enhancing system performance by balancing the workload of tasks, there remain many challenges that warrant attention. Firstly, modifications to the grouping rules necessitate the rerouting of certain data streams. This, in turn, calls for the caching of mapping relationships between tuple key values and task identifiers. Should the cached data volume exceed manageable limits, it could give rise to increased query time and memory utilization, thereby adding to the system's overall complexity. Secondly, although these methods can distribute data evenly among tasks in operators, addressing the load balancing issue from a single perspective, such as data grouping, data flow scheduling, or at the task or node level, only provides limited improvement in system performance [12].

Other efforts have primarily focused on optimizing task scheduling to mitigate node stragglers. The objectives include minimizing inter-node communication and achieving an equitable distribution of resource loads among the nodes [13]. A two-phase mapping mechanism was used in [14], where tasks are grouped to reduce inter-group communication in the first phase, and more capable nodes are prioritized for task assignment in the second phase. A trade-off between the communication latency of operators and the workload of worker nodes was examined in [15] by first placing highly communicative operators on the same worker nodes and then iteratively rescheduling only the less communicative operators online. These studies employ a round-robin algorithm for the iterative deployment of task groups to nodes with higher computational capabilities. While effectively balancing resource loads among logical task groups and optimizing inter-group communication, they do not necessarily ensure the balanced distribution of resource loads among the physical compute nodes. The improper deployment of task groups to compute nodes can worsen the imbalance in workload among nodes [16].

Optimizing load balancing in distributed streaming computing systems presents a multifaceted challenge that necessitates careful consideration at various levels [3]. Focusing solely on load balancing between tasks within the operators falls short of maximizing system throughput, just as concentrating solely on load balancing between compute nodes falls short of minimizing system response time.

In terms of optimizing inter-task load, ensuring an equitable distribution of data among tasks within operators does not necessarily achieve an equitable workload distribution across nodes. This is primarily due to two significant reasons: (1) The parallelism of each operator in streaming applications can vary, resulting in different resource consumption averages for tasks within these operators. (2) Some operators might involve complex calculations that consume more computing resources, while others may perform simpler operations that use fewer resources. When tasks with high resource load are deployed to the same node, it can result in a pronounced imbalance in node workload [17].

In terms of optimizing inter-node load, dynamically fine-tuning task deployment via the scheduler can keep the load difference between compute nodes within an appropriate range. However, skewed data streams can cause some tasks in operators to process more data, leading to longer queuing time and further affecting the system latency [18]. Therefore, a load skew-aware scheduling strategy at multiple tiers is urgently expected to mitigate both task and node stragglers in distributed stream computing systems.

Motivated by the above discussions, we propose a hierarchical collaborative framework called Ms-Stream. It aims at continuously optimizing system performance through the integration of various factors across multiple levels. Briefly, Ms-Stream dynamically reschedules data tuples at runtime based on the load difference between task stragglers and others in stateful operators. It constructs a two-tier router to enable each task in stateful operators to manage multiple partitioned data, effectively mitigating the complexity and reducing the size of traditional routers.

Further, two fine-grained models for resource allocation and resource placement are introduced to mitigate node stragglers. (1) Resource allocation: Communication-intensive tasks are assigned to the same group to minimum inter-node communication, while computation-intensive tasks are evenly distributed across groups. (2) Resource placement: Task groups are deployed to compute nodes with varying resource capacities, aiming to minimize the load difference between node stragglers and other nodes.

Through this multi-tier collaboration, Ms-Stream can maintain a prolonged online state and effectively handle data streams with skewed distribution, while supporting features such as high throughput, fast response time and efficient resource utilization.

1.1. Contributions

This paper proposes a hierarchical collaborative strategy (Ms-Stream) for mitigating stragglers and improving the throughput and latency of distributed stream computing systems. The key contributions are as follows:

(1) A two-tier routing table for workload allocation is implemented by combining hash-based and key-based data grouping. This table relocates some partitions from task stragglers to other tasks in stateful operators, adapting to skewed data streams.
(2) Communication-intensive tasks are grouped together via Graph Convolutional Network (GCN), while guaranteeing a well-balanced distribution of computation-intensive tasks among these groups.
(3) Task groups are assigned to compute nodes with diverse resource capacities through the descending maximum padding priority rule for a balanced workload distribution across nodes.
(4) Ms-Stream is integrated into the Apache Storm platform and evaluated on metrics such as system throughput and latency. Experimental results show that Ms-Stream provides promising improvements compared to existing state-of-the-art solutions.

1.2. Paper organization

The rest of the paper is organized as follows: Section 2 reviews the related work on data stream grouping and streaming application scheduling. Section 3 introduces the Ms-Stream system model, including models for topology, grouping, and resources. Section 4 formalizes the problems caused by stragglers in distributed stream computing systems, especially focusing on data stream grouping, task subgraph partitioning and task deployment. Section 5 introduces the optimization methods to address the problems identified in Section 4. Section 6 explains the framework and main algorithms of Ms-Stream. Section 7 analyzes the performance evaluation results with metrics of system throughput and latency. Section 8 concludes our work and presents directions for future work.

2. Related work

In this section, we review state-of-the-art work in two related areas: data grouping and task scheduling for stream processing. A comparison of five key aspects between Ms-Stream (our work) and relevant research

Table 1
Related work comparison.

Related work	Aspects				
	Scheduling object	Methodology	Objective	Time complexity	Heterogeneous
Li et al. [6]	Task	Cost-effective assignment	Inter-node load balancing	$O(k \cdot logk + n^2)$	✔
SP-Ant [15]	Task	Ant colony algorithm	Communication-aware	$O(it \cdot n^2)$	✔
Hone [19]	Tuple	Largest-Backlog-First	Inter-task load balancing	$O(u)$	✔
Dalton [20]	Tuple	Reinforcement learning	Inter-task load balancing	Null	✗
PS-UIM [21]	Task	Heuristic algorithm	Resource-aware	$O(n \cdot k \cdot nw)$	✔
POTUS [22]	Tuple	Predictive tuple grouping	Inter-task load balancing	$O(u^2 \cdot rt)$	✔
LLFD [23]	Tuple	Least-Load Fit Decreasing	Inter-task load balancing	$O(u \cdot ck \cdot rt)$	✔
Brown et al. [24]	Task	Resource provisioning	Resource-aware	Null	✔
Our work	Task, Tuple	Two-Level grouping, GCN, Descending maximum padding priority	Inter-task load balancing, Inter-node load balancing	$O(ts)$, $O(it \cdot u^2 \cdot (d+2))$, $O(k \cdot logk)$	✔

is summarized in Table 1. Under column "Time complexity", u represents the number of tasks for the operator, k is the number of nodes, n is the total number of tasks, it is the number of algorithm iterations, nw is the number of workers per node, ck is the number of candidate tuples, rt is the number of tuples migrated, and ts is the size of the routing table.

2.1. Data stream grouping

Data stream grouping is a crucial operation in stream computing systems, redirecting data tuples from an unbounded data stream to tasks in operators based on specific rules [25]. This process plays a vital role in enabling real-time aggregation, data distribution, and event correlation, making it essential for efficiently processing real-time data streams in applications such as stream analytics and monitoring. In recent years, an increasing number of researchers have focused on optimizing data grouping to achieve low latency and high throughput in stream computing environments.

To optimize workload distribution and reduce system latency, a tuple scheduler [19] employing an online Largest-Backlog-First (LBF) algorithm was introduced. This approach effectively manages tuple scheduling to minimize queue backlogs and balance queue backlogs in tasks, mitigating task stragglers when workloads exhibit variance. However, it may face limitations when dealing with stateful operators.

To mitigate the system performance degradation incurred by workload imbalance across tasks, a finer-grained control method [22] was proposed. By distributing data stream tuples between tasks, this method adapts well to variations in data streams and workload discrepancies. It mainly employs the power of predictive scheduling to achieve a tunable trade-off between communication cost reduction and system queue stability. Unfortunately, it lacks the capacity to redirect data streams of stateful operators.

To distribute workload effectively among stateful operators, a key-based workload partitioning strategy [23] was proposed for dynamic workload assignment. This approach combines hash-based and explicit key-based routing strategies to specify destination worker threads for some keys while using a hashing function for others. However, managing a large number of data tuples in routing tables can increase tuple emission time and memory overhead.

To address load imbalance caused by skewed data among stateful operators, a learned partitioning method [20] was proposed for distributed stream systems. This method relies on reinforcement learning, providing rewards for hotkeys based on a cost model that captures load variations and continuously optimizes the learned model for load balancing among operator tasks. However, resource load variability can lead to a large state space, increasing the time required for the agent to identify the target operator for tuple dispatch.

In summary, balancing workload among tasks in operators by optimizing data stream grouping has been extensively studied. Most approaches either disregard workload balancing among tasks in stateful operators or overlook the complexity of data structures established for workload assignment. To address these limitations, we develop a two-tier routing table that is lightweight and ensures minimal consumption of system resources.

2.2. Task scheduling

Task scheduling plays an important role for improving system throughput and reducing response time [26]. In stream computing systems, the allocation of computational resources relies on efficient task scheduling, which dynamically assigns computational tasks to available nodes, considering resource requirements and system load. Finding an optimal task scheduling has proven challenging, as this scheduling problem is NP-hard [27]. Furthermore, this complexity is plain by the fact that tasks scheduling is performed in an online environment, and has to consider both the system availability and scheduling efficiency.

To achieve reliable and efficient processing of unpredictable stream workflows, strategies for scheduling and provisioning dynamic workflow scenarios with various complexities and degrees of unpredictability [24] were investigated. These strategies primarily focus on task prioritization and give precedence to scheduling critical tasks. However, they ignore communication overhead between tasks.

To minimize system response time, a novel stream processing scheduling [15] using an ant colony algorithm was proposed to make a trade-off between the communication latency of operators and the utilization of worker nodes. This algorithm finds the best operator assignment plan by considering the inter-node communication latency of operators and collocating highly communicative operators on the same worker nodes. However, it overlooks workload variations among nodes, which can impact system performance.

To avoid inefficient utilization of computing resources caused by interference among stream processing tasks, an optimal scheduling method [21] was proposed for processing big data streams on heterogeneous servers in a multi-core environment. This approach uses a fine-grained strategy for core scheduling and a coarse-grained strategy for compute node scheduling to improve resource utilization in clusters.

To minimize job execution cost and balance load in cluster, a cost-efficient task scheduling algorithm and a cost-efficient load balancing algorithm [6] were introduced. The task scheduling algorithm reduces overall cost consumption but may result in load imbalance within the cluster. This limitation is addressed by the cost-efficient load balancing algorithm which strikes a balance between load distribution and cost optimization.

In summary, most of the aforementioned approaches aimed to establish effective resource management for distributed stream computing systems, often focusing on one perspective or level, e.g., the data grouping or task scheduling perspective, or the task or node level. However, optimizing performance from one single perspective or level can limit the extent of performance improvement. Moreover, effective coordination is essential, considering the interconnected nature of task- node-levels. In our work, we construct a multi-level collaborative framework to mitigate both task and node stragglers, optimizing system performance at both the task level and node level.

Table 2
Description of primary symbols used in this paper.

Symbol	Description
$R_{o_{i,k}}$	Resources consumed by task $o_{i,k}$ in operator o_i
R_{o_i}	Set of resource load of all tasks in operator o_i
R_{an}	Set of resource load of activated nodes
R_g	Set of resource load of task groups
W	Diagonal matrix consisting of resource load of each task group
A	Diagonal matrix consisting of resource load of each compute node
B	Matrix consisting of resource load of each task
E	Matrix consisting of communication rates between tasks
X	Deployment decision matrix
Y	Matrix consisting of probability for each task's affiliation with groups
LIT	Load imbalance degree between tasks
LIN	Load imbalance degree between nodes
$\sum$	Sum of all elements in the matrix

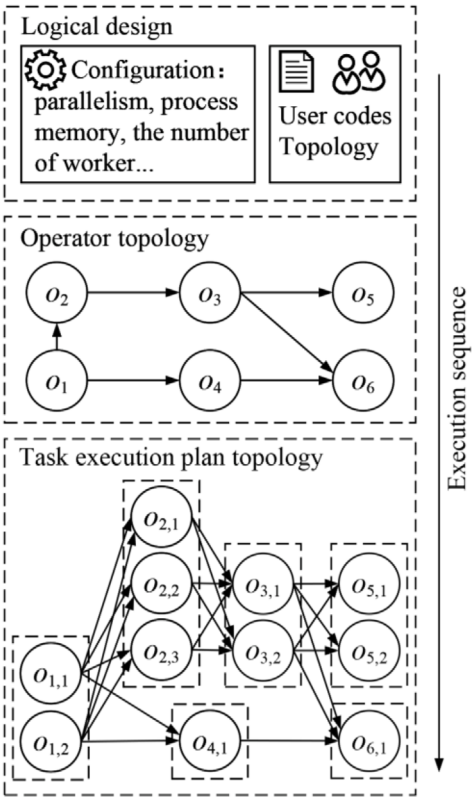

Fig. 1. Initialization of a streaming application.

3. System model

Before introducing the Ms-Stream strategy and its related algorithms, we first explain the topology model, the data grouping model and the resource model in distributed stream computing environments. For enhanced clarity, Table 2 provides the primary notations used throughout the paper.

3.1. Topology model

The primary function of a streaming application is to process and analyze continuously generated data streams, enabling real-time monitoring, analysis, decision-making, and response [28]. These applications empower enterprises to make rapid, data-driven decisions based on real-time information, improving business operations and enhancing user experiences.

The logic topology of each streaming application can be represented as a directed acyclic graph (DAG) [26]. It consists of a set of operators and a set of directed edges, defined as $G_L = \{O(G_L), Ed(G_L)\}$. Here, $O(G_L) = \{o_i | i \in 1, \ldots, n\}$ denotes a finite set of n operators, and the function $f(o)$ of each operator o is completely unique, that is if $\forall o_i, o_j \in O(G_L)$, then $\nexists f(o_i) = f(o_j)$, meaning the function of each operator is different. $Ed(G_L) = \{e_{i,j} | 1 \le i, j \le n \text{ and } i \ne j\}$ denotes a finite set of directed edges, where the weights associated with the edges represent communication costs between operator o_i and operator o_j.

If a streaming application's logical topology G_L is submitted to the data center, multiple task instances will be initialized for each operator according to the parallelism set by user, where for $\forall o_{i,k}, o_{i,m} \in o_i$, $\exists f(o_{i,k}) = f(o_{i,m})$, meaning the function of each task instance within the same operator is identical. This dependency between task instances is described as a directed acyclic task topology $G_T = \{O(G_T), Ed(G_T)\}$, where $O(G_T) \subseteq O(G_L)$, $Ed(G_T) \subseteq Ed(G_L)$. $\forall o_i \in O(G_L)$, $\exists k \in \{1, \ldots, u\}$, $\{o_{i1}, \ldots, o_{ik}, \ldots, o_{iu}\}$ are tasks of operator o_i, and $\{o_{i1}, \ldots, o_{iu}\} \subset O(T)$.

As shown in Fig. 1, a streaming application is submitted to the cluster through the scheduler by the user. The initial operator topology consists of 6 operators, denoted as $\{o_1, o_2, \ldots, o_6\}$, each serving a different function. A task topology is then constructed based on the configuration of operators, where operators o_4 and o_6 each have 1 parallel task, o_1, o_3 and o_5 each have 2 parallel tasks, and o_2 has 3 parallel tasks. Tasks within the same operator have the same function, for example, $f(o_{1,1}) = f(o_{1,2})$, indicating that the functions for tasks $o_{1,1}$ and $o_{1,2}$ are identical.

3.2. Grouping model

Data stream grouping refers to the process of dividing a data stream into different substreams to enable parallel processing [29]. In a stream processing system, the input data streams need to be distributed to different tasks or operators for parallel computation. Given a data stream $ds_k = \{dt_1, dt_2, \ldots, dt_i, \ldots\}$, all the data tuples in the stream ds_k are intelligently assigned to tasks in downstream operators through a grouping function $F(ds_k)$. This grouping function $F(ds_k)$ strategically directs tuples to tasks based on their characteristics. As a result, the data stream ds_k is efficiently decomposed into multiple substreams $\{ds_{k,1}, \ldots, ds_{k,m}, \ldots, ds_{k,n}\}$, where each individual substream, exemplified by $ds_{k,m}$, is precisely routed to tasks $o_{k,m}$. The grouping model can be described by Eq. (1).

$$F\left(ds_k\right) = \left\{dt_1, dt_2, \ldots\right\} \rightarrow \left\{ds_{k,1}, \ldots, ds_{k,m}, \ldots, ds_{k,n}\right\} \tag{1}$$

The relationship between data stream ds_k and these substreams can be described as (2).

$$ds_k = \bigcup_{m=1}^{n} ds_{km} \tag{2}$$

An upstream task in o_{k-1} emits data tuples to n tasks $\{o_{k,1}, o_{k,2}, \ldots, o_{k,n}\}$ in the downstream operator o_k through the grouping function $F(ds_k)$. An effective data grouping strategy is one data tuple is only emitted to one task in operators. Therefore, the relationship between operator o_k's substreams can be described by (3).

$$ds_{k,1} \bigcap ds_{k,2}, \ldots, \bigcap ds_{k,n} = \emptyset \tag{3}$$

3.3. Resource model

A streaming application's task topology G_T is a real-time data processing application designed to continuously analyze and process data streams. It utilizes various resources, primarily comprising communication and computational resources [30]. By effectively leveraging these resources, streaming applications can efficiently process data streams in real-time, adapting to dynamically changing data scenarios. Proper configuration and management of heterogeneous resources are crucial for ensuring scalability, performance, and reliability in streaming applications [31].

(1) Communication resource. We implement the built-in IMetric interface on the Storm platform to collect the communication rates between tasks. If two tasks are deployed on different compute nodes, it gives rise to a communication rate between these tasks, denoted as $cr(o_{i,k}, o_{j,m})$, and it can be calculated by Eq. (4).

$$cr(o_{i,k}, o_{j,m}) = \begin{cases} 0, & \text{If } o_{i,k} \text{ and } o_{j,m} \text{ are deployed on same node,} \\ e_{cr}, & \text{Otherwise.} \end{cases} \tag{4}$$

where e_{cr} denotes the average communication rate during the time interval $[t_s, t_e]$, and t_s and t_e denote the start time and end time of this interval set by the user, respectively. As the data stream rate may experience transient fluctuations, we can mitigate their impact by subtracting the maximum and minimum rates and calculating the average rate e_{cr}. It can be obtained by Eq. (5).

$$e_{cr} = \frac{\int_{t_s}^{t_e} e_{cr}^t dt - \max(e_{cr}^t) - \min(e_{cr}^t)}{t_e - t_s}, \tag{5}$$

where e_{cr}^t denotes the communication rate between tasks at time t, $t \in [t_e, t_s]$.

(2) Computing resource. High CPU consumption indicates the execution of computationally intensive tasks, potentially leading to slower responsiveness. Concurrently, increased memory usage may induce system instability, augmented swapping operations, and an overall degradation in performance. To model computing resource, we collect CPU and memory consumption data from nodes using the built-in Top command in Linux.

At time t, a compute node cn_i may run multiple different tasks. We define this set of tasks on compute node cn_i as T_{cn_i}. In addition, we denote the CPU utilization of the compute node cn_i as L_{cn_i}, and the number of data tuples processed by each task $o_{i,k}$ of compute node cn_i as $pr_{o_{i,k},cn_i}$, where $o_{i,k} \in T_{cn_i}$.

Then, the CPU utilization of each task $o_{i,k}$ running on the compute node cn_i can be calculated by Eq. (6).

$$R_{o_{i,k}}^c = \frac{w_{o_{i,k}} \cdot pr_{o_{i,k},cn_i}}{\sum_{o_{j,m} \in T_{cn_i}} w_{o_{j,m}} \cdot \sum_{o_{j,m} \in T_{cn_i}} \left(pr_{o_{j,m},cn_i} \cdot \rho_{o_{i,k},cn_i} \right)} \cdot L_{cn_i}, \tag{6}$$

where $R_{o_{i,k}}^c$ denotes the CPU utilization of task $o_{i,k}$ on compute node cn_i. $w_{o_{i,k}}$ denotes the complexity of task $o_{i,k}$ in operator o_i. $\rho_{o_{i,k},cn_i}$ denotes the decision variable of task, and it can be obtained by Eq. (7).

$$\rho_{o_{i,k},cn_i} = \begin{cases} 1, & o_{i,k} \text{ is running,} \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

Similarly, at time t, the memory utilization of compute node cn_i can be defined as M_{cn_i}. The memory utilization of each task $o_{i,k}$ running on compute node cn_i can be calculated by Eq. (8).

$$R_{o_{i,k}}^m = \frac{\beta_{o_{i,k}} \cdot pr_{o_{i,k},cn_i}}{\sum_{o_{j,m} \in T_{cn_i}} \beta_{o_{j,m}} \cdot \sum_{o_{j,m} \in T_{cn_i}} \left(pr_{o_{j,m},cn_i} \cdot \rho_{o_{i,k},cn_i} \right)} \cdot M_{cn_i}, \tag{8}$$

where $R_{o_{i,k}}^m$ denotes the memory utilization of task $o_{i,k}$ on compute node cn_i. $\beta_{o_{i,k}}$ denotes the space complexity of task $o_{i,k}$.

Then, the resources consumed by task $o_{i,k}$ on node cn_i at time t, denoted as $R_{o_{i,k}}$, can be calculated by Eq. (9).

$$R_{o_{i,k}} = \mu \cdot R_{o_{i,k}}^c + (1 - \mu) \cdot R_{o_{i,k}}^m, \mu \in [0, 1], \tag{9}$$

where μ denotes the weighting factor between the CPU and memory utilization.

4. Problem formulation

In this section, we formalize the problems related to stragglers in tasks or nodes in distributed stream computing systems. These problems mainly include data grouping, subgraph partitioning, and task deployment.

4.1. Data grouping

If tasks within operators experience skewed workloads, with certain tasks processing significantly more data than others (referred to as task stragglers), it can lead to several adverse effects. Firstly,it may result in ineffective resource utilization, where some tasks are excessively utilized while others remain idle, potentially wasting computational resources and affecting overall system efficiency. Secondly, task stragglers, handling larger volume of data, can become performance bottlenecks for the entire streaming application, impacting overall task execution time and resulting in a decrease in system throughput.

We define the load imbalance degree between tasks within operator o_i as LIT. It measures the degree of load skew between tasks within an operator and can be calculated by Eq. (10).

$$LIT = \max \left(\frac{\max\left(R_{o_i}\right) - \overline{R}_{o_i}}{\overline{R}_{o_i}}, \frac{\overline{R}_{o_i} - \min\left(R_{o_i}\right)}{\overline{R}_{o_i}} \right), \tag{10}$$

and

$$\overline{R}_{o_i} = \frac{\sum_{k=0}^{card(o_i)} R_{o_{i,k}}}{card(o_i)}, \tag{11}$$

where R_{o_i} and $card(o_i)$ denote the set of resource load of tasks and the number of tasks in operator o_i, respectively.

To mitigate the impact of task stragglers on system performance, it is advisable to maintain LIT within an appropriate range, e.g., maximum χ. Therefore, the data grouping strategy problem should satisfy the condition $LIT \leq \chi$.

4.2. Subgraph partitioning

A streaming application G_T is deployed to k compute nodes $\{cn_1, cn_2, \ldots, cn_k\}$. We construct a matrix E based on the communication loads between tasks $T = \{o_{1,1}, o_{1,2}, \ldots, o_{i,k}, \ldots, o_{j,m}\}$, where $o_{j,m}$ represents the mth task of the final operator o_j in this streaming application, and it can be described by Eq. (12).

$$E = \left[e_{l,b}\right] = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{n,1} & e_{1,1} & \cdots & e_{n,n} \end{bmatrix} \tag{12}$$

where n denotes the number of tasks in the streaming application G_T. $e_{l,b}$ denotes the communication load between the bth task number and the lth task number in the task set T.

The optimization objective for the subgraph partitioning problem is to minimize communication loads between compute nodes and evenly distribute the workload among them. This can be generalized as shown in Eq. (13).

$$\begin{aligned} \min Z = {} & \lambda \cdot \sum_{1 \leqslant h,s \leqslant k} \left[c\left(g_h, g_s\right) + c\left(g_s, g_h\right) \right] \\ & + (1 - \lambda) \cdot \sum \left| \left(\frac{W}{k} - \frac{\sum W}{k^2} \right) \odot I \right| \\ = {} & \lambda \cdot \sum_{h}^{k} c\left(g_h, \widetilde{g_h}\right) \\ & + (1 - \lambda) \cdot \sum \left| \left(\frac{W}{k} - \frac{\sum W}{k^2} \right) \odot I \right| \end{aligned} \tag{13}$$

where λ denotes the weighting factor between communication load and computing load. $\sum$ denotes the sum of all elements in the matrix. W is a diagonal matrix that comprises the resource workloads of the task groups resulting from the partitioning of tasks in G_T into subgraphs. I is an $k \times k$ identity matrix. g_h denotes the hth task group (subgraph in topology $G_T = \{g_1, \ldots, g_h, \ldots, g_s, \ldots, g_k\}$). $\widetilde{g_h}$ denotes the complement

of subset g_h in topology G_T. $c\left(g_h, g_s\right)$ denotes the communication load between group g_h and group g_s, which can be calculated by Eq. (14).

$$\begin{aligned} c\left(g_h, g_s\right) &= \sum_{l \in g_h} \sum_{b \in g_s} e_{l,b} \\ d_l &= \sum_{b \in g_s} e_{l,b} \end{aligned} \tag{14}$$

We define vector $v_t = (v_{t,1}, \ldots, v_{t,i}, \ldots, v_{t,n})^{\mathrm{T}}$, where $1 \leqslant t \leqslant k$ and $1 \leqslant i \leqslant n$ are group indicators, and the size of v_t equals the number of tasks in topology G_T. $v_{t,i}$ can be calculated by Eq. (15).

$$v_{t,i} = \begin{cases} 1, & \text{the } i\text{th task belongs to the } t\text{th group} \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

The communication load generated by all tasks in task group g_h can be calculated by Eq. (16).

$$\sum_{l \in g_h} d_l = v_h^{\mathrm{T}} D v_h \tag{16}$$

where v_h denotes the tasks belonging to the hth group. D is a diagonal matrix with the lth diagonal element as d_l.

In addition, the communication load between tasks in task group g_h can be calculated by Eq. (17).

$$c\left(g_h, g_h\right) = v_h^{\mathrm{T}} E v_h \tag{17}$$

where E denotes the matrix of communication loads between tasks in set T.

Then, the communication load between task group g_h and other groups can be obtained by subtracting the intra-group communication load in group g_h from the total communication load of all tasks in group g_h, and it can be calculated by Eq. (18).

$$\begin{aligned} c\left(g_h, \widetilde{g_h}\right) &= \sum_{l \in g_h} \sum_{b \in \widetilde{g_h}} e_{l,b} = v_h^{\mathrm{T}} D v_h - v_h^{\mathrm{T}} E v_h \\ &= v_h^{\mathrm{T}} (D - E) v_h \end{aligned} \tag{18}$$

where $\widetilde{g_h}$ denotes the complement of subset g_h in G_T.

Therefore, the objective function of subgraph partitioning can be reformulated as Eq. (19).

$$\begin{aligned} \min Z = \lambda \cdot \sum_{t=1}^{k} v_t^{\mathrm{T}} (D - E) v_t \\ + (1 - \lambda) \cdot \sum \left| \left(\frac{W}{k} - \frac{\sum W}{k^2} \right) \odot I \right| \end{aligned} \tag{19}$$

4.3. Task deployment

The challenge in task deployment involves choosing k nodes from a pool of m available compute nodes and assigning k subgraphs to them, with the constraint that each subgraph, although it may have multiple compute nodes as potential candidates, can only be deployed to one compute node. Based on this description, we construct a deployment decision matrix X, which can be described by Eq. (20).

$$X = \left[x_{i,j}\right] = \begin{bmatrix} x_{1,1} & \cdots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{k,1} & \cdots & x_{k,m} \end{bmatrix}, \tag{20}$$

where $x_{i,j}$ can be calculated by Eq. (21).

$$x_{i,j} = \begin{cases} 1, & \text{the } i\text{th task group is} \\ & \quad \text{deployed to the } j\text{th node} \\ 0, & \text{otherwise.} \end{cases} \tag{21}$$

And matrix X must satisfy conditions given by Eq. (22).

$$\begin{cases} \sum_{j=1}^{m} x_{i,j} = 1, \quad i = 1, 2, \ldots, k, \\ \sum_{i=1}^{k} x_{i,j} = 1, \quad j = 1, 2, \ldots, m. \end{cases} \tag{22}$$

Based on the matrix X, the objective function J of task deployment can be generalized as Eq. (23).

$$\min J = \sigma\left(XA + WX\right), \tag{23}$$

where $\sigma(\cdot)$ denotes the standard deviation of $(\cdot)$. Matrix A denotes the resource load of compute nodes and is a $m \times m$ diagonal matrix with the ith diagonal element as a_i.

a_i can be calculated by Eq. (24).

$$a_i = \mu \cdot L_{cn_i}^{c} + (1 - \mu) \cdot M_{cn_i}^{c}, \mu \in [0, 1], \tag{24}$$

and matrix W denotes the resource load of task groups and can be calculated by Eq. (25).

$$W = V^{\mathrm{T}} B V, \tag{25}$$

where $V = (v_1, v_2, \ldots, v_k)^{\mathrm{T}}$ and matrix B denotes the resource load of each task in task topology G_T. B is a diagonal matrix with the ith diagonal element as b_i. b_i is the resource load of the ith task in task set T.

Based on above description, the objective function of task deployment can be reformulated as Eq. (26). If the objective function J can be minimized, the resource load between the compute nodes deployed for topology G will be balanced as much as possible, ensuring efficient resource utilization in cluster.

$$J = \sum \left| \left(\frac{XA + WX}{k} - \frac{\sum (XA + WX)}{k^2} \right) \odot X \right| \tag{26}$$

5. Ms-Stream: optimizer models

In this section, we present three optimizer models designed to address the three aforementioned challenges, namely, optimizers for data grouping, subgraph partitioning, and task deployment.

5.1. Data grouping optimizer

In the streaming application G_T, a stateful operator o_i comprises u tasks denoted as $o_i = \{o_{i,1}, o_{i,2}, \ldots, o_{i,u}\}$, where $o_{i,u}$ is the final task of operator o_i. To balance the workload among these u tasks in operator o_i, data tuples$\{dt_1, dt_2, \ldots, dt_i, \ldots\}$ from upstream operators are grouped using a two-tier router. This router is achieved by the map function $M(dt_i)$, which distributes data tuple dt_i in partition p_w to the specified task $o_{i,k}$, as described by Eq. (27).

$$M(dt_i) = p_w \rightarrow o_{i,k}, \tag{27}$$

where partition p_w denotes a set of data tuples with similar characteristics, $p_w \in P, P = \{p_1, \ldots, p_w, \ldots, p_m\}$, where m is the number of partitions and $m > k$. Partition p_w is a conceptual entity and can be calculated by Eq. (28).

$$p_w = Hash\left(dt_i\left(key\right)\right) \% card\left(P\right), \tag{28}$$

where $dt_i\left(key\right)$ denotes the key values of tuple dt_i, and $card(P)$ denotes the size of set P.

As shown in Fig. 2, downstream operator o_i contains two tasks $o_{i,0}, o_{i,1}$. Upstream tasks emit data tuples to operator o_i through this router. Firstly, the data tuple determines the partition number to be emitted using the hash function. Secondly, the map function is employed to locate the task number associated with the partition number. Finally, the tuple is emitted to the designated task. The two-tier router in Fig. 2 offers notable advantages over traditional routers [9,32]. Traditional routers, such as Megaphone router [32], maintain state information for each input tuple, requiring global queries to route tuples to task instances, which can degrade performance with large data volumes. However, our two-tier router avoids per-tuple state management. Instead, it maps partitions (logical groups of tuples with the same characteristics) to downstream tasks, resulting in a lighter, faster design with reduced retrieval time.

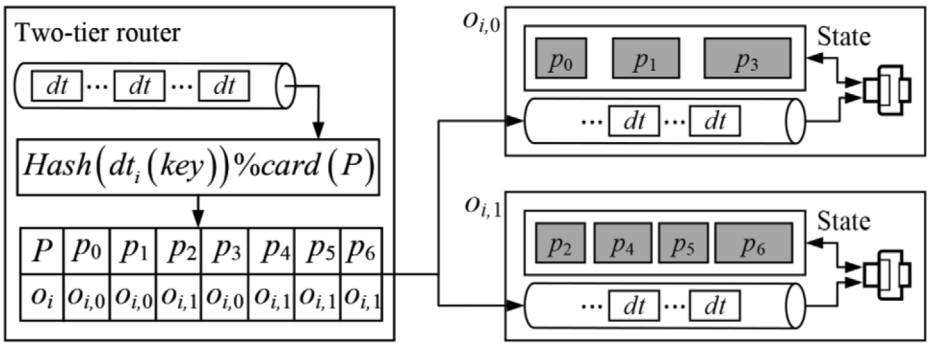

Fig. 2. Two-tier router in Ms-Stream.

If the workload disparity between tasks in operator o_i satisfies the condition $LI > \chi$, significant load imbalance occurs in the stream computing system. In this case, we need to sort the partition P in ascending order based on their resource consumption and migrate some data from task $o_{i,h}$ with high workload to task $o_{i,l}$ with low workload, satisfying conditions given by Eq. (29). In this process, we utilize the built-in data transmission channels of the Storm platform to migrate these states. These states are transmitted in the form of data streams among operator tasks.

$$\begin{aligned} &\frac{R_{o_{i,h}} - \overline{R}_{o_i}}{\overline{R}_{o_i}} \leqslant \chi \\ \Rightarrow &\frac{R_{o_{i,h}} \cdot \left(pr_{o_{i,h},cn} - \sum_{q \in MS} p_q\right) - \overline{R}_{o_i} \cdot pr_{o_{i,h},cn}}{\overline{R}_{o_i} \cdot pr_{o_{i,h},cn}} \leqslant \chi \\ \Rightarrow &pr_{o_{i,h},cn} - \frac{(1+\chi) \cdot \overline{R}_{o_i} \cdot pr_{o_{i,h},cn}}{R_{o_{i,h}}} \leqslant \sum_{q \in MS} p_q, \end{aligned} \tag{29}$$

where MS denotes the set of migrated partitions, and $MS \subset P$.

5.2. Subgraph partitioning optimizer

Recently, deep learning approaches have been employed to address subgraph partitioning challenges. Our proposed solution for distributed stream computing systems relies on the Generalizable Approximate Partitioning (GAP) framework [33], specifically designed for grouping tasks in streaming applications. As shown in Fig. 3, the subgraph partitioning model of Ms-Stream mainly includes two modules: the graph embedding module and the graph partitioning module.

(1) Graph embedding module. The primary objective of the graph embedding module is to acquire task embeddings by leveraging both the graph structure and task features. In this module, we employ Graph Convolution Network (GCN) [34] and Graph Sample and Aggregate (GraphSAGE) [35] techniques to master graph representations formed by tasks in the streaming application. GCN leverages convolutional operations on the graph's adjacency matrix to learn task representations. GraphSAGE samples and aggregates information from the neighbors of each task, providing flexibility in choosing different neighborhood sampling and aggregation strategies for generating task representations.

(2) Graph partitioning module. This module is designed to perform graph partitioning by taking task embeddings as input and producing the probability Y for each task's affiliation with groups in $\{g_1, g_2, \ldots, g_k\}$. This module is a fully connected layer followed by softmax, and its training objective is to minimize the specified loss function.

We implement a different loss function LF based on GAP to evaluate the balancedness of groups, which can be described by Eq. (30).

$$LF = Bc\left[c\left(g_h, \widetilde{g}_h\right)\right] + Bn[Y] + Br[Y] \tag{30}$$

In the first term of Eq. (30), $Bc\left[c\left(g_h, \widetilde{g}_h\right)\right]$ denotes the probability that two tasks are not in the same group, which can be calculated by Eq. (31).

$$Bc\left[c\left(g_1, \ldots, g_k\right)\right] = \sum (Y \oslash \Gamma)(1 - Y) \odot E \tag{31}$$

where Γ can be calculated by Eq. (32).

$$\Gamma = Y^{\mathrm{T}} H, \tag{32}$$

and H is the vector that represents the degree of tasks.

In the second term of Eq. (30), $Bn[Y]$ represents the balancedness of the number of tasks in each group, which can be calculated by Eq. (33).

$$\begin{aligned} Bn[Y] &= \sum_{j=1}^{k} \left(\sum_{i=1}^{n} Y_{i,j} - \frac{n}{k} \right)^2 \\ &= \sum \left(\mathbf{1}^{\mathrm{T}} Y - \frac{n}{k} \right)^2, \end{aligned} \tag{33}$$

where n denotes the total number of tasks in the streaming application G_T and k denotes the number of groups.

In the third term of Eq. (30), $Br[Y]$ denotes the balancedness of resource load of each group, which can be calculated by Eq. (34).

$$\begin{aligned} Br[Y] &= \sum_{j=1}^{k} \left(\sum_{i=1}^{n} Y_{i,j} \cdot B_{i,i} - \frac{\sum B}{k} \right) \\ &= \sum \left(Y^{\mathrm{T}} B \mathbf{1} - \frac{\sum B}{k} \right)^2 \end{aligned} \tag{34}$$

where matrix B denotes the resource load of each task in G_T and is a $n \times n$ diagonal matrix. $\mathbf{1}$ is a $n \times 1$ column vector and each element in this vector is "1".

Based the above description, the loss function of the subgraph partitioning model can be reformulated as Eq. (35).

$$\begin{aligned} LF = &\sum (Y \oslash \Gamma)(1 - Y) \odot E \\ &+ \sum \left(\mathbf{1}^{\mathrm{T}} Y - \frac{n}{k} \right)^2 + \sum \left(Y^{\mathrm{T}} B \mathbf{1} - \frac{\sum B}{k} \right)^2 \end{aligned} \tag{35}$$

Through this loss function, the model continuously inputs the topology, iteratively updating parameters until it converges to an optimal solution.

5.3. Task deployment optimizer

Graph partitioning primarily aims to minimize communication between subgraphs and balance resource loads among them. After partitioning, each subgraph has a different resource load. If these subgraphs are not appropriately deployed to compute nodes, nodes hosting subgraphs with higher resource loads may become stragglers, impacting the overall application performance. Therefore, Ms-Stream employs a descending maximum padding priority strategy to further optimize the cluster's resource efficiency.

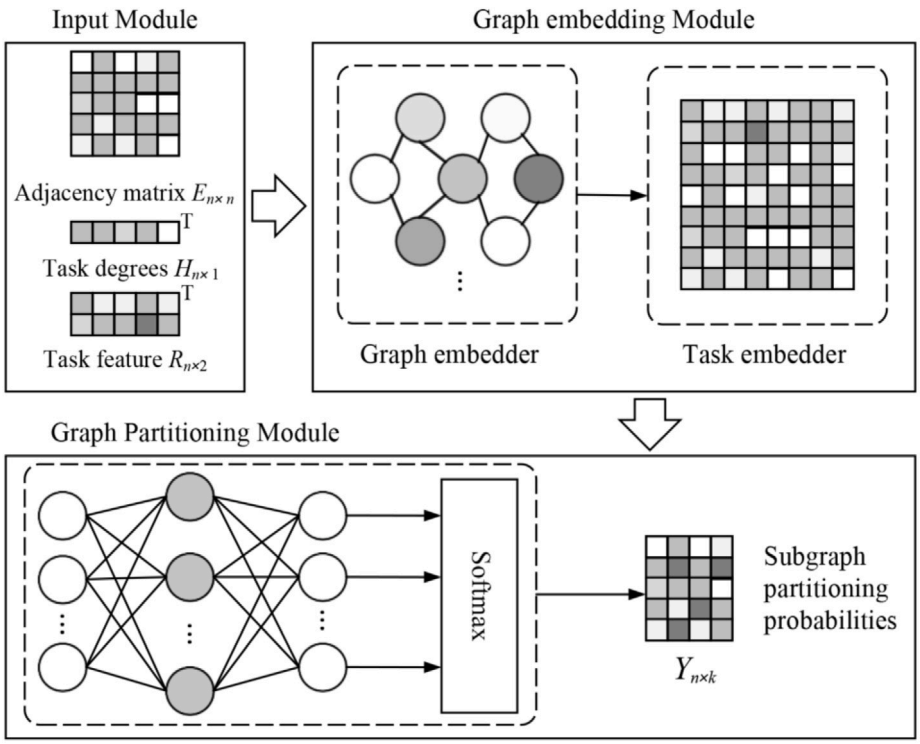

Fig. 3. Subgraph partitioning model.

We classify the compute nodes $Cn = \{cn_1, cn_2, \ldots, cn_m\}$ in the cluster into activated nodes $An = \{an_1, an_2, \ldots, an_u\}$ and inactive nodes $In = \{in_1, in_2, \ldots, in_v\}$. Activated nodes An are those nodes that are already running other tasks. Inactive nodes In are those nodes that are not running any tasks and are in sleep state. The relationship between them can be described by Eq. (36).

$$Cn = An \cup In, An \cap In = \emptyset \tag{36}$$

If the number of nodes in In are over-activated when deploying a streaming application, the cluster will incur more energy consumption. Therefore, nodes in An are used with higher priority than nodes in In.

Assume there exists task groups $\{g_1, g_2, \ldots, g_k\}$ generated by the subgraph partition optimizer. We deploy these k groups to the m available compute nodes and define the resource threshold for nodes as ξ. Firstly, an ascending sort algorithm is applied to the resource load $R_{an} = \{R_{an_1}, R_{an_2}, \ldots, R_{an_u}\}$ of activated nodes, while the resource load $R_g = \{R_{g_1}, R_{g_2}, \ldots, R_{g_k}\}$ of each task group is subjected to a descending sort. Secondly, a bijection is established between the elements of the sorted sets R_g and R_{an}. We compare the first elements of sets R_g and R_{an}. If the condition satisfies Eq. (37), a key–value pair $\langle g_z, an_e \rangle$ is formed, and both elements are removed from R_g and R_{cn}.

$$R_{g_z} + R_{cn_e} \leqslant \xi, R_{g_z} \in R_g, R_{cn_e} \in R_{cn}, \tag{37}$$

where R_{g_z} and R_{cn_e} are the first elements in set R_g and R_{an}, respectively. In cases where this condition Eq. (37) is not satisfies, the element R_{g_z} in R_g is mapped to In and is removed after this mapping. This iteration continues until there are no more elements in R_g.

6. Ms-Stream: framework and algorithms

Based on the above analysis, we combine the aforementioned three optimization models into a hierarchical straggler-aware scheduling strategy. In this section, we introduce the Ms-Stream framework and algorithms for data grouping, subgraph partitioning, and task deployment.

6.1. System framework

As shown in Fig. 4, the Ms-Stream framework includes five levels: user level, topology level, deployment level, data level, and resource level. Ms-Stream primarily utilizes the built-in IMetric interface of Storm for tracking the runtime information of each stream application and thereby ensuring minimal monitoring overhead.

The user level enables users to build logical topologies by clearly defining the internal logic and data dependencies of streaming applications, primarily using the Spout and Bolt interfaces provided by Storm. The topologies vary depending on the unique functions defined by different users. Although optimizing these topologies poses a challenge, enhancing user comprehension of the applications helps in developing more effective topologies. Once submitted to the Storm platform, these topologies can be further optimized.

The topology layer is primarily responsible for continuously monitoring and assessing the load distribution among nodes in real-time. Should it identify an imbalance in this distribution that adversely affects the system's overall performance, the topology layer will proactively initiate a redeployment for tasks across the nodes. This task redeployment process mainly pursues two core objectives: minimizing the amount of communication load between nodes and achieving a relative balance in the workload across nodes.

The deployment layer, activated after the topology layer redistributes tasks among nodes, primarily handles the task groups received from the topology layer and deploys them across the cluster. Its key objectives encompass achieving a relative balance in the resource loads of the cluster's nodes, ensuring a balanced load on nodes where task groups are deployed, and minimizing the number of activated nodes without compromising performance. We implement the IScheduler interface built into the Storm platform to achieve these objectives. After passing through the topology layer and the deployment layer, we obtain a set of task-to-node mappings. The IScheduler implementation class redeploys the tasks to the corresponding computational nodes based on this mapping set.

The data layer plays a crucial role in the system architecture, primarily responsible for monitoring and evaluating the workload among

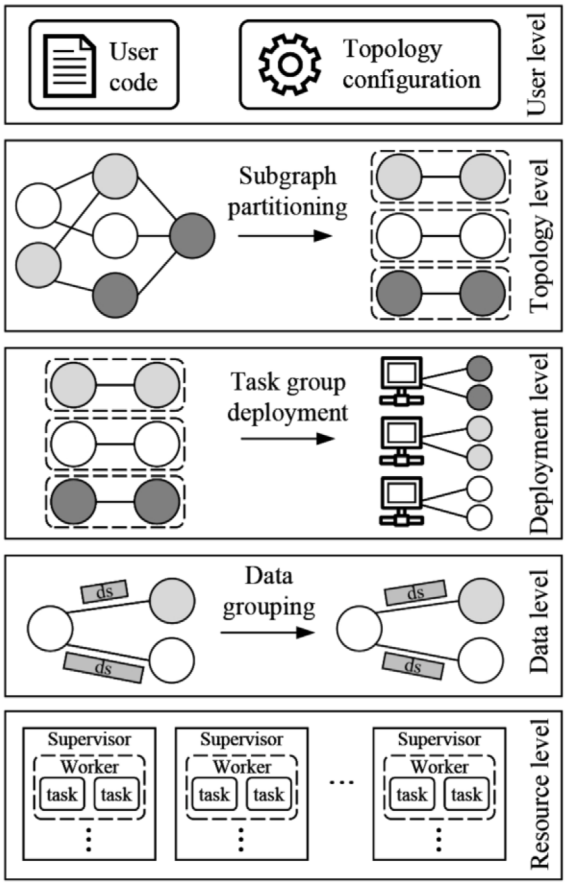

Fig. 4. Ms-Stream framework.

various tasks within the system. To achieve efficient data management, this layer utilizes an innovative two-tier routing table design, allowing a single task to effectively manage multiple data partitions. When faced with the challenge of imbalanced workload distribution among tasks, the data layer employs intelligent algorithms to reassign the distribution of partitions across tasks. This readjustment aims to optimize data stream and enhance the overall performance of the system.

The resource layer is primarily responsible for effectively running streaming applications and their associated monitoring components, ensuring the efficient utilization of resources, while also maintaining the stable operation of systems. Moreover, the resource layer must collaborate closely with other layers of the system to ensure smooth information stream and accurate data processing. In this process, multi-level strategic coordination is important.

6.2. Data grouping algorithm

At the data level, an uneven distribution of data streams among tasks within operators can lead to system inefficiencies. To address this issue, it is essential to balance the workload across these tasks. By optimizing performance at the task level, we can significantly enhance the global efficiency of the entire task topology. This approach not only alleviates the stress on overburdened tasks but also ensures a more effective and harmonious utilization of available resources. The algorithm for balancing the workload between tasks within operator is described in Algorithm 1.

The input of Algorithm 1 is data tuple dt, and its output is the task number $o_{i,k}$ to which the tuple will be emitted. Step 1 initializes the routing table, which stores the mapping relationships between partitions and tasks. Step 3 to step 7 focus on determining the target

Algorithm 1: Data grouping algorithm.

Input: data tuple dt;
Output: TaskID $o_{i,k}$;
1 Initialize the routing table $M(P,O) = \{(p_1, o_{i,1}), (p_2, o_{i,1}), ..., (p_m, o_{i,u})\}$;
2 /*Data stream processing logic*/
3 **if** *dt is the data stream* **then**
4 Get the key value of the data tuple, denoted as key ;
5 Get the hash value of key, noted as $hkey$;
6 $p_w = hkey\%size(P)$;
7 Find the TaskID $o_{i,k}$ corresponding to p_w from $M(P,O)$;
8 /*Control stream processing logic*/
9 **else**
10 Get the partition number p_w from data tuple dt ;
11 Get the TaskID $o_{i,k}$ from data tuple dt ;
12 Modify the TaskID corresponding to p_w in $M(P,O)$ to $o_{i,k}$;
13 **return** *TaskID* $o_{i,k}$

instance number for dispatching the data tuple. This involves a two-step process: initially computing the partition number associated with the data tuple, and then using this partition number to consult the routing table in order to find the corresponding instance number. Steps 9 to 12 primarily respond to the data migration strategy. Balancing the load among tasks will result in the migration of some partitions from lagging tasks to those with lower workloads. The three steps in this algorithm are capable of synchronizing the routing table with these changes in partition numbers.

In Algorithm 1, the main functions involve handling data streams and control streams. Initially, it is designed to handle data streams by receiving data tuples and routing them to the appropriate tasks for processing. The secondary function is to manage control flows by detecting and responding to variations in the workload of downstream tasks. In cases of workload imbalance among these downstream tasks, a reorganization of the partitions within these tasks is necessary, leading to alterations in the corresponding instances of each partition. This algorithm ensures system reliability by synchronizing these dynamic relationships between partitions and tasks in the routing table.

6.3. Subgraph partitioning algorithm

At topology level, if there is an imbalance in the workload distribution among nodes, it becomes essential to minimize the resource load differences between straggler nodes and other nodes by dynamically redistributing task deployments across compute nodes at runtime. We define the load imbalance degree LIN between nodes as shown in Eq. (38).

$$LIN = \max\left(\frac{m \cdot \max(A)}{\sum A} - 1, 1 - \frac{m \cdot \min(A)}{\sum A}\right), \tag{38}$$

where Matrix A denotes the resource load of compute nodes and is an $m \times m$ diagonal matrix.

We define the threshold value for load imbalance degree as η. If $LIN > \eta$, the system will trigger a rescheduling. During the phase of online rescheduling, it is crucial to partition the task topology into subgraphs. This approach is designed to minimize communication latency in the system and achieve a more balanced distribution of workload across the nodes. The details of this process are described in Algorithm 2.

The input of Algorithm 2 includes the matrix B of task resource load, the matrix E of communication load between tasks, and the number m of subgraphs. The output of this algorithm is a probability matrix Y, where the element $Y_{i,j}$ denotes the probability of the ith task belonging to the jth subgraph. Steps 5 to 10 encode the edge features of streaming applications by continuously aggregating the neighboring

Algorithm 2: Subgraph partitioning algorithm.

Input: Task features B, edge features E, number of subgraphs m;
Output: Probability matrix Y ;
1 Initialize the maximum number of iteration defined by user, noted as $count$;
2 **while** $count > 0$ **do**
3 Initialize the depth of the graph encoding, noted as d ;
4 **for** $i = 1$ *to* d **do**
5 **for** *each* $e_{l,b}$ *in* E **do**
6 Get task neighbors ϑ of edge $e_{l,b}$;
7 Get the edge set $\varepsilon_{i,j}$ of each task ϑ_i in ϑ ;
8 Calculate the embedding of edge $e_{l,b}$ based on ϑ and ε information ;
9 $e_{l,b} = \sum_{v \in \vartheta} v \cdot \sum_{e \in \varepsilon_{v,j}} e$;
10 **end**
11 **for** *each* $R_{o_{i,k}}$ *in* B **do**
12 Get task neighbors ϑ of task $o_{i,k}$;
13 Get edges ε between each task in ϑ and $o_{i,k}$;
14 Calculate the embedding of task $o_{i,k}$ based on ϑ and ε information;
15 $R_{o_{i,k}} = \sum_{v \in \vartheta, e \in \varepsilon} v \cdot e$;
16 **end**
17 **end**
18 $H^{(0)} \leftarrow \mathrm{CONCAT}(B,E)$;
19 $H^{(2)} \leftarrow \mathrm{Linear}(\mathrm{Linear}(H^{(0)}))$;
20 $Y \leftarrow \mathrm{softmax}(H^{(2)})$;
21 Calculate the loss function and backpropagate the gradient;
22 $count = count - 1$;
23 **end**
24 **return** Y

edge features. Steps 11 to 16 can encode the task characteristics of a streaming application by continuously aggregating neighbor feature information. Step 18 concatenates the encoded edge features and task features. In step 19, this integrated dataset is then fed into a fully connected neural network for processing. Step 20 outputs the probability matrix Y through the softmax function. Step 21 calculates the loss function, followed by the execution of gradient backpropagation for updating the network's parameters.

In Algorithm 2, by repeatedly inputting the network topology and updating the network parameters, the algorithm not only optimizes the communication load between subgraphs, but also ensures a dynamic balance in resource allocation among subgraphs.

6.4. Task deployment algorithm

Based on Algorithm 2, when a streaming application is successfully partitioned into multiple subgraphs, each subgraph can have several nodes that satisfy the deployment criterion. This criterion states that the computational resources required by the subgraph are less than those available on the node. In cases where subgraphs are assigned to nodes in an inefficient manner, this could lead to the unnecessary activation of a significant number of compute nodes within the data center. Such a scenario not only escalates energy consumption substantially but also results in a marked underutilization of available resources, as many nodes may remain idle or be employed below their capacity. The details of this process are described in Algorithm 3.

The input of Algorithm 3 includes the resource load R_g of task groups, the resource load R_{an} of activated nodes, and inactive nodes In. The output of this algorithm is the mapping result Mr between task groups and nodes. Steps 2 to 5 involve sorting R_{an} and R_g according

Algorithm 3: Task deployment algorithm.

Input: Resource load $R_g = \{R_{g_1}, R_{g_2}, ..., R_{g_k}\}$ of task groups, Resource load $R_{an} = \{R_{an_1}, R_{an_2}, ..., R_{an_u}\}$ of activated nodes, Inactive nodes $In = \{in_1, in_2, \dots, in_v\}$;
Output: Mapping result Mr between task groups and nodes ;
1 Initialize an empty set of Mr ;
2 **if** R_{an} *is not empty* **then**
3 Sort R_{an} in ascending order ;
4 **end**
5 Sort R_g in descending order ;
6 **if** *size of* R_g *is greater than* R_{an} **then**
7 $\phi = size(R_g) - size(R_{an})$;
8 Get ϕ elements from In to add to the header in R_{an} and mark the resource load of the elements as 0 ;
9 **end**
10 **while** R_g *is not empty* **do**
11 Get the first element R_{g_1} in R_g;
12 Get the first element R_{an_1} in R_{an};
13 **if** $R_{g_1} + R_{an_1} \leqslant \xi$ **then**
14 $Mr \leftarrow \langle g_1, an_1 \rangle$;
15 remove(R_{g_1},R_{an_1});
16 **else**
17 $Mr \leftarrow \langle g_1, in_1 \rangle$;
18 remove(R_{g_1},in_1);
19 **end**
20 **end**
21 **return** Mr

to different rules. Steps 6 to 9 involve assessing whether R_{an} meets the criteria for deploying R_g. If it does not, we add some inactive nodes to R_{an} to meet the minimum node requirement for deploying R_g. Steps 10 to 20 involve mapping the task groups to suitable nodes, which mainly consists of two parts: (1) If deploying the task group to a node does not cause overload, we map the task group to that node. (2) Otherwise, we deploy the task to an inactive node. Since sets R_{an} and R_g are ordered, we only operate on their head elements.

In Algorithm 3, activated nodes are given priority in usage to reduce the energy consumption of the cluster. When matching task groups to nodes, two objectives can be met: (1) Efforts are made to ensure load balancing among the nodes where the task groups are deployed. (2) The system's resource load is also balanced as much as possible to improve resource utilization efficiency.

7. Performance evaluation

In this section, we evaluate the performance of Ms-Stream system. The experimental environment and parameter settings are first discussed, followed by an analysis of the impact of inter-task and inter-node load imbalance on the system, as well as the system performance.

7.1. Experimental setup

The Ms-Stream system is built on Storm [36] and deployed on Ubuntu 20.04. The system's cluster comprises 20 machines, each 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. Moreover, two machines host Storm Nimbus as master nodes, and three are designated for running ZooKeeper. Machines assigned both Nimbus and ZooKeeper also function as supervisor nodes, while the remaining 15 exclusively serve as supervisor nodes.

(1) Datasets. In the experiment, we utilize a real-world data set [37] from Alibaba Cloud and synthetic datasets following the Zipf

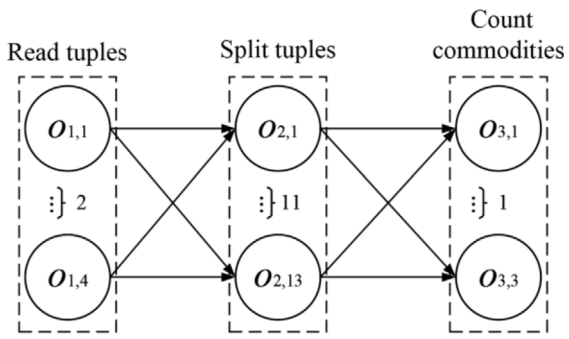

Fig. 5. Logical graph of COMMCount.

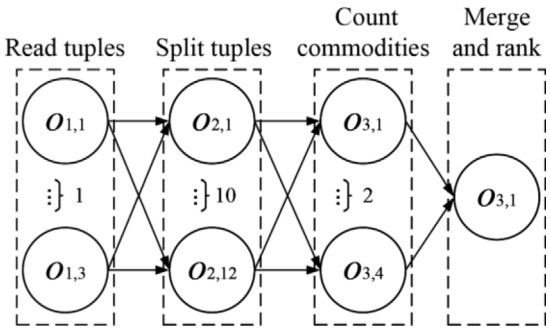

Fig. 6. Logical graph of Top_N.

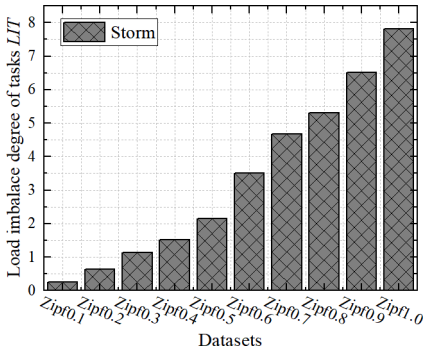

Fig. 7. Load imbalance degree of stateful tasks on different datasets.

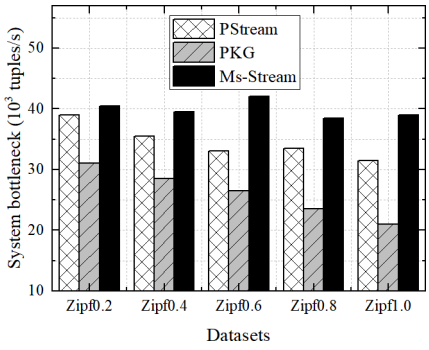

Fig. 8. System bottleneck of COMMCount with different load imbalance degrees between tasks.

distribution to test the system performance. The real-world dataset encompasses the activities of approximately one million random Taobao users who were active between November 25th and December 3rd, 2017. These activities include clicks, purchases, additions to cart, and likes. Each row of the dataset represents a tuple, consisting of user ID <Long type>, product ID <Long type>, product category ID <Long type>, type of activity <String type>, and timestamp <Long type>.

The Zipf distribution is a probability distribution used to describe the relationship between the frequency of elements in a dataset and their rank [38]. Its coefficient determines the skewness of the element frequency distribution. A larger coefficient corresponds to a greater degree of skewness. Therefore, we use the Zipf coefficient to adjust the skewness of the synthetic dataset. We set different coefficients to construct the dataset. For example, we set this coefficient to 0.2, 0.4, 0.6, 0.8, and 1.0, respectively, and denote the constructed datasets as Zipf0.2, Zipf0.4, Zipf0.6, Zipf0.8, and Zipf1.0. The generation of these synthetic datasets is similar to the methods described in [38,39].

(2) Streaming applications. We evaluate Ms-Stream with two different streaming applications: COMMCount and Top_N. COMMCount counts the number of browsing commodities, and Top_N identifies the products with the highest purchasing power. The logic graphs of COMMCount and Top_N are shown in Fig. 5 and Fig. 6, respectively.

(3) Baseline schemes and metrics. In the experiment, we compare the performance of Ms-Stream with those of state-of-the-art designs, including the popularity-aware key Grouping (PStream) [9], partial key grouping (PKG) [40], resource-aware scheduling (R-Storm) [41], and Storm [36]. We mainly evaluate the load imbalance degree, system bottleneck, and latency. High bottleneck (i.e. maximum throughput) and low latency are two critical metrics of system performance. We define system bottleneck as the maximum number of data tuples successfully processed per second by the system, and latency as the average processing time for each tuple.

7.2. Inter-task load imbalance

In this experiment, the impact of different LIT values on system performance is evaluated using various synthetic datasets. We set the trigger load balancing factor χ to 1.0. This setting triggers the balancing load strategy when the imbalance degree between tasks exceeds 1.0.

We simulate changing workloads under different scenarios by adjusting the data stream skewness. As shown in Fig. 7, we use the synthetic datasets as input to the stream computing system to simulate the load variations of operator tasks. The experimental results show that the degree of load imbalance increases with the rise of the Zipf skew coefficient. This is because, as the Zipf coefficient increases, the unevenness of data distribution grows, leading to a minority of tuples occupying a larger proportion. In stream computing systems, this may cause some tasks to process significantly more data than others. To address data skew, efficient load balancing strategies are to be adopted to balance tasks' workloads.

Given an increasing data stream rate and an increment of 500 tuples/s, the system bottleneck (i.e., maximum throughput) of different approaches is evaluated using different synthetic datasets. As shown in Figs. 8 and 9, compared to PStream and PKG, Ms-Stream exhibits a higher system bottleneck when running different streaming applications. In the COMMCount experiment, Ms-Stream achieves a 15.2% improvement over PStream. In the Top_N experiment, Ms-Stream shows a 22.1% enhancement compared to PStream. The significant boost in Ms-Stream can be attributed to its lightweight routing strategy and effective state data management. From these two experiments, it can be observed that PKG's system bottleneck decreases with increasing data skew. This is because PKG sends a data tuple to the lower-load task among two tasks, which can alleviate load imbalance but not eliminate the issue, bringing additional communication and memory overhead.

Given a stable input rate of 10,000 tuples/s, the system latency of different works is evaluate using different synthetic datasets. Compared to PStream and PKG, Ms-Stream has a lower response time when running different streaming applications. As shown in Fig. 10, Ms-Stream and PStream exhibit stable system latency as the skewness of the data stream increases. The average system latency for Ms-Stream

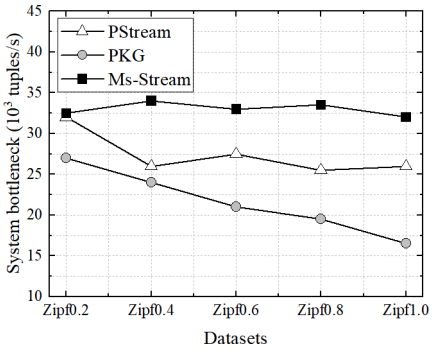

Fig. 9. System bottleneck of Top_N with different load imbalance degrees between tasks.

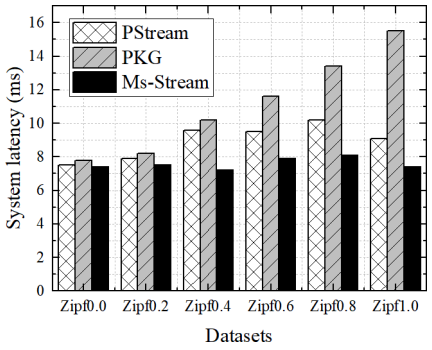

Fig. 10. System latency of COMMCount with different load imbalance degrees between tasks.

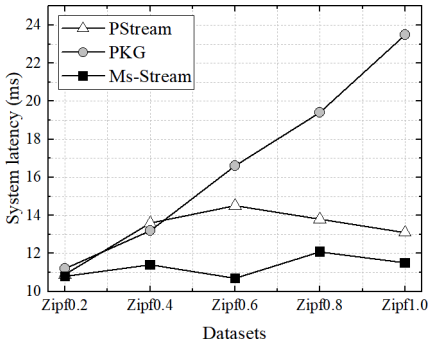

Fig. 11. System latency of Top_N with different load imbalance degrees between tasks.

and PStream is 9.6 ms and 7.6 ms, respectively. Ms-Stream reduces system latency by 20.3% compared to PStream. When the data stream skewness is set to Zipf 0.0, the latencies for Zipf 0.0 and Zipf 0.2 are very similar. This is because the degree of resource imbalance at Zipf 0.2 is insufficient to have a significant influence on system latency.

Similarly, in the Top_N experiment shown in Fig. 11, Ms-Stream reduces system latency by 17.8% compared to PStream. It can be found from experiments of the two streaming applications that Ms-Stream exhibits latency similar to PStream and PKG on Zipf0.2 because the load imbalance degree remains below the threshold, preventing Ms-Stream and PStream from triggering the load balancing strategy. The experiments demonstrate that Ms-Stream is more effective in balancing the workload among tasks within operators.

7.3. Inter-node load imbalance

In this experiment, we assess the effect of imbalanced resource loads LIN across nodes on system performance by employing a range of

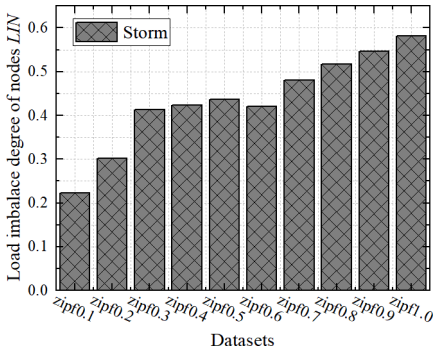

Fig. 12. Load imbalance degree of nodes on different datasets.

synthetic datasets. We consider the resource utilization of compute nodes under extreme scenarios. When the average resource utilization of compute nodes in the cluster reaches 55%–60%, the resource utilization of the node with the highest resource consumption should not exceed 80%. This metric ensures stable system operation under high load conditions, preventing the overload of any single node from causing performance bottlenecks or system crashes. Therefore, the load balancing threshold, denoted as η, for the Ms-Stream system is configured at 0.4. This parameter activates the load balancing mechanism whenever the imbalance degree of resource allocation exceeds 0.4.

Firstly, we construct the experimental environment with varying degrees of load imbalance by feeding diverse synthetic datasets into the stream computing system. The varying degrees of load imbalance can simulate the resource utilization of the cluster at different times. As shown in Fig. 12, as the coefficient of Zipf increases, the load imbalance in the system generally exhibits an upward trend.

Given an increasing data stream rate and an increment of 500 tuples/s, the system bottleneck (i.e., maximum throughput) of different approaches is evaluated with different load imbalance degrees between nodes. As shown in Figs. 13 and 14, Ms-Stream effectively maintains a stable system bottleneck even as data skew increases. This is achieved through its advanced load balancing mechanisms and intelligent task scheduling, which effectively distribute the workload across the cluster while ensuring optimal resource utilization. In contrast, R-Storm and Storm exhibit a gradual decline in their system bottlenecks as data skew increases. This is attributed to their less effective load balancing strategies, which struggle to handle the uneven workload distribution, leading to performance degradation.

At Zip0.2, the system bottleneck of Ms-Stream is lower than that of R-Storm due to the relatively lower data skew, which falls below the threshold required to trigger load balancing policies within Ms-Stream. However, in scenarios of highly skewed data streams, Ms-Stream's system bottleneck significantly surpasses that of both R-Storm and Storm. The experimental results indicate that as the skewness of the data stream increases, the improvement in system performance achieved by Ms-Stream compared to R-Storm and Storm becomes more evident.

Given a stable input rate of 5000 tuples/s, the system latency of different works is evaluated with different load imbalance degree between nodes. As shown in Figs. 15 and 16, the system latencies of Ms-Stream, R-Storm, and Storm increase significantly with the degree of data skew. However, Ms-Stream can maintain a lower system latency than R-Storm and Storm. While Ms-Stream effectively balances the resource load of the cluster, its system latency has a slight increase as the degree of data skew increases, because the system latency is affected by various factors, including the imbalance of workload between nodes and tasks. When the data stream becomes more skewed, a large number of data tuples are accumulated in fewer tasks, resulting in longer queuing time for tasks, which in turn affects the system latency. Therefore, it is crucial to implement a multi-level load balancing strategy to ensure optimal performance.

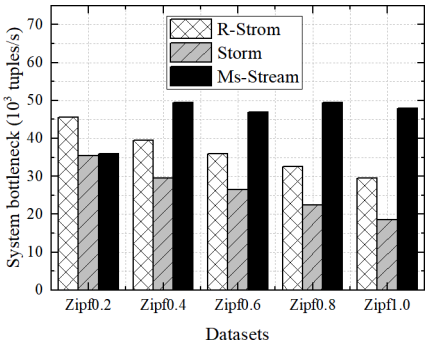

Fig. 13. System bottleneck of COMMCount with different load imbalance degrees between nodes.

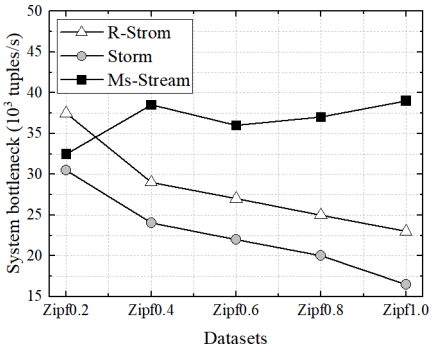

Fig. 14. System bottleneck of Top_N with different load imbalance degrees between nodes.

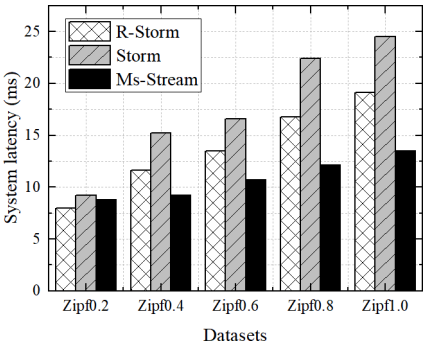

Fig. 15. System latency of COMMCount with different load imbalance degrees between nodes.

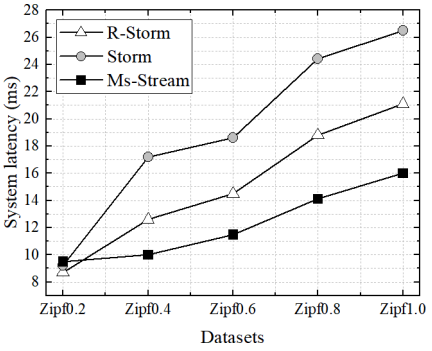

Fig. 16. System latency of Top_N with different load imbalance degrees between nodes.

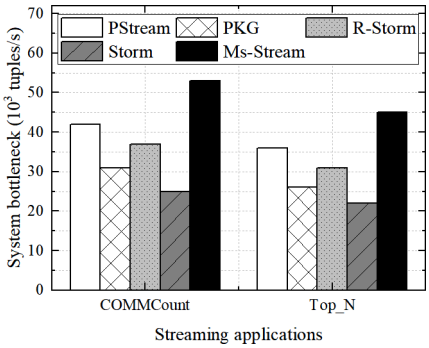

Fig. 17. System bottleneck of two streaming applications on real-word datasets.

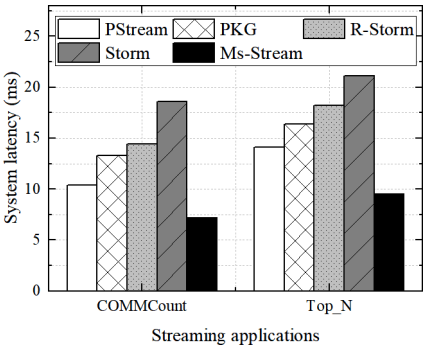

Fig. 18. System latency of two streaming applications on real-word datasets.

7.4. Overall performance improvement

In these experiments, we evaluate the overall performance of Ms-Stream using a real-world dataset provided by Alibaba Cloud, focusing on two metrics: system bottleneck and system latency.

Given an increasing data stream rate and an increment of 500 tuples/s, Ms-Stream exhibits significant improvements in system bottleneck compared to state-of-the-art works across different streaming applications. As shown in Fig. 17, the average system bottleneck for two streaming applications are 39,000 tuples/s, 28,500 tuples/s, 34,000 tuples/s, 23,500 tuples/s, and 49,000 tuples/s for PStream, PKG, R-Storm, Storm, and Ms-Stream respectively when the system stabilizes. Compared to the most advanced PStream, Ms-Stream enhances the system bottleneck by 25.6%. It is evident that the average bottleneck of Ms-Stream surpasses that of other works when the input rate is stable.

Given a stable input rate of 8000 tuples/s, Ms-Stream significantly reduces system latency in different streaming applications compared to state-of-the-art works. As shown in Fig. 18, in the COMMCount experiment, the average system latency is 10.4 ms, 13.3 ms, 14.4 ms, 18.6 ms, and 7.2 ms for PStream, PKG, R-Storm, Storm, and Ms-Stream respectively when the system stabilizes. In the Top_N experiment, the average system latency is 14.1 ms, 16.4 ms, 18.2 ms, 21.1 ms, and 9.5 ms for PStream, PKG, R-Storm, Storm, and Ms-Stream respectively when the system stabilizes. The maximum system latency of Ms-Stream is reduced by 61.2%, while the minimum is reduced by 30.7%. The reason why Ms-Stream has lower latency is that Ms-Stream implements a multi-layer load balancing strategy to optimize the system's performance in the face of skewed data stream.

Given a stable input rate of 2000 tuples/s, Ms-Stream has lower system latency across various numbers of streaming applications compared to state-of-the-art approaches. As shown in Fig. 19, for a single streaming application, Ms-Stream exhibits the lowest latency at 6.1 ms, while Storm demonstrates the highest at 11.2 ms. As the number of streaming applications increases, all systems experience an increase

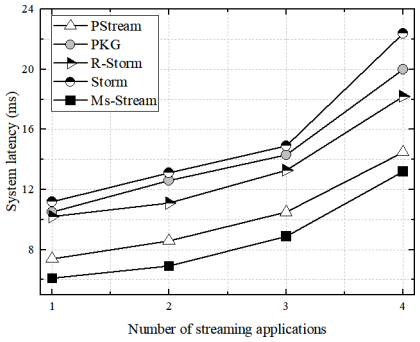

Fig. 19. Average system latency under varying numbers of applications.

in latency, with Ms-Stream consistently maintaining lower values. Notably, the rate of latency increase for Ms-Stream is significantly lower than those of the other systems as the number of streaming applications grows. This performance improvement can be attributed to Ms-Stream's optimized resource scheduling and data stream management mechanisms.

8. Conclusions and future work

In this paper, we introduced Ms-Stream, a hierarchical scheduling strategy designed to tackle skewed data distribution challenges in stream computing systems. These challenges cause untimely task execution and hinder system efficiency by generating task and node stragglers. The primary objective of Ms-Stream is to balance workloads among tasks and keep the workload differences among nodes within acceptable limits. It achieves this by establishing a lightweight, two-level grouping method that facilitates dynamic workload assignment for stateful tasks and offloads resources from stragglers. Furthermore, it ensures the equitable distribution of computation-intensive tasks across groups and deploys task groups to nodes of varying capacity using a descending maximum padding priority rule. We implemented the proposed strategy on the Apache Storm platform. Experimental results demonstrated remarkable performance enhancements across various skewness levels, whether synthetic or real-world datasets. This approach offers a significant advantage over existing solutions, markedly improving both system throughput and latency.

In our future work, we will integrate the auto-scaling operator parallelism mechanism into Ms-Stream to further reduce data processing latency and consider the energy consumption of cluster to improve energy efficiency.

CRediT authorship contribution statement

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

Declaration of competing interest

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

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 62372419; the Fundamental Research Funds for the Central Universities of China, China under Grant No. 265QZ2021001.

Data availability

Data will be made available on request.

References

[1] Tiangang Li, Shi Ying, Yishi Zhao, Jianga Shang, Batch jobs load balancing scheduling in cloud computing using distributional reinforcement learning, IEEE Trans. Parallel Distrib. Syst. 35 (1) (2024) 169–185.
[2] Yancan Mao, Zhanghao Chen, Yifan Zhang, Meng Wang, Yong Fang, Guanghui Zhang, Rui Shi, Richard T.B. Ma, StreamOps: Cloud-native runtime management for streaming services in ByteDance, Proc. VLDB Endow. 16 (12) (2023) 3501–3514.
[3] Xiangqiang Gao, Rongke Liu, Aryan Kaushik, Hierarchical multi-agent optimization for resource allocation in cloud computing, IEEE Trans. Parallel Distrib. Syst. 32 (3) (2021) 692–707.
[4] Shuhao Zhang, Jiong He, Amelie Chi Zhou, Bingsheng He, BriskStream: Scaling data stream processing on shared-memory multicore architectures, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 705–722.
[5] Dawei Sun, Minghui Wu, Zhihong Yang, Atul Sajjanhar, Rajkumar Buyya, A two-tier coordinated load balancing strategy over skewed data streams, J. Supercomput. 79 (18) (2023) 1–28.
[6] Hongjian Li, Jianglin Xia, Wei Luo, Hai Fang, Cost-efficient scheduling of streaming applications in apache flink on cloud, IEEE Trans. Big Data 9 (4) (2023) 1086–1101.
[7] Hai Jin, Fei Chen, Song Wu, Yin Yao, Zhiyi Liu, Lin Gu, Yongluan Zhou, Towards low-latency batched stream processing by pre-scheduling, IEEE Trans. Parallel Distrib. Syst. 30 (3) (2019) 710–722.
[8] Avadh Kishor, Rajdeep Niyogi, Bharadwaj Veeravalli, Fairness-aware mechanism for load balancing in distributed systems, IEEE Trans. Serv. Comput. 15 (4) (2022) 2275–2288.
[9] Hanhua Chen, Fan Zhang, Hai Jin, Pstream: A popularity-aware differentiated distributed stream processing system, IEEE Trans. Comput. 70 (10) (2021) 1582–1597.
[10] Xi Huang, Ziyu Shao, Yang Yang, POTUS: Predictive online tuple scheduling for data stream processing systems, IEEE Trans. Cloud Comput. 10 (4) (2022) 2863–2875.
[11] Shaoshuai Ding, Lei Yang, Jiannong Cao, Wei Cai, Mingkui Tan, Zhenyu Wang, Partitioning stateful data stream applications in dynamic edge cloud environments, IEEE Trans. Serv. Comput. 15 (4) (2022) 2368–2381.
[12] Lei Xu, Qimin Xu, Jingzheng Tu, Jinglong Zhang, Yanzhou Zhang, Cailian Chen, Xinping Guan, Learning-based scalable scheduling and routing co-design with stream similarity partitioning for time-sensitive networking, IEEE Internet Things J. 9 (15) (2022) 13353–13363.
[13] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, Efficient operator placement for distributed data stream processing applications, IEEE Trans. Parallel Distrib. Syst. 30 (8) (2019) 1753–1767.
[14] Muhammad Asif, Muhammad Aleem, BAN-storm: A bandwidth-aware scheduling mechanism for stream jobs, J. Grid Comput. 19 (3) (2021) 1–16.
[15] Mohammadreza Farrokh, Hamid Hadian, Mohsen Sharifi, Ali Jafari, SP-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters, Expert Syst. Appl. 191 (2022) 1–11.
[16] Zichuan Xu, Guangyuan Xu, Hao Wang, Weifa Liang, Qiufen Xia, Shangguang Wang, Enabling streaming analytics in satellite edge computing via timely evaluation of big data queries, IEEE Trans. Parallel Distrib. Syst. 35 (1) (2024) 105–122.
[17] Tiejun Wang, Xudong Mou, Juntao Hu, Rui Wang, Tianyu Wo, Two-stage scheduling of stream computing for industrial cloud-edge collaboration, in: 2022 IEEE International Conference on Joint Cloud Computing, JCC, 2022, pp. 57–64.
[18] Muhammad Mudassar Qureshi, Hanhua Chen, Fan Zhang, Hai Jin, IPC: Resource and network cost-aware distributed stream scheduling on skewed streams, Adv. Eng. Inform. 46 (2020) 1–10.
[19] Wenxin Li, Duowen Liu, Kai Chen, Keqiu Li, Heng Qi, Hone: Mitigating stragglers in distributed stream processing with tuple scheduling, IEEE Trans. Parallel Distrib. Syst. 32 (8) (2021) 2021–2034.
[20] Eleni Zapridou, Ioannis Mytilinis, Anastasia Ailamaki, Dalton: Learned partitioning for distributed data streams, Proc. VLDB Endow. 16 (3) (2022) 491–504.
[21] Shun Wang, Guosun Zeng, Two-stage scheduling for a fluctuant big data stream on heterogeneous servers with multicores in a data center, Cluster Comput. 66 (2023) 1–17.
[22] Xi Huang, Ziyu Shao, Yang Yang, Dynamic tuple scheduling with prediction for data stream processing systems, in: 2019 IEEE Global Communications Conference, GLOBECOM, 2019, pp. 1–6.
[23] Junhua Fang, Rong Zhang, Tom Z.J. Fu, Zhenjie Zhang, Aoying Zhou, Xiaofang Zhou, Distributed stream rebalance for stateful operator under workload variance, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2223–2240.

[24] Alexander Brown, Saurabh Garg, James Montgomery, Ujjwal K.C., Resource scheduling and provisioning for processing of dynamic stream workflows under latency constraints, Future Gener. Comput. Syst. 131 (2022) 166–182.
[25] Muhammad Mudassar Qureshi, Hanhua Chen, Hai Jin, Modeling distributed stream processing systems under heavy workload, in: 2019 International Conference on Cyberworlds, CW, 2019, pp. 93–100.
[26] Leila Eskandari, Jason Mair, Zhiyi Huang, David Eyers, I-Scheduler: Iterative scheduling for distributed stream processing systems, Future Gener. Comput. Syst. 117 (2021) 219–233.
[27] A. Muhammad, M.A. Qadir, MF-storm: a maximum flow-based job scheduler for stream processing engines on computational clusters to increase throughput, PeerJ Comput. Sci. 8 (2022) 1–23.
[28] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, Karthik Ramasamy, Dhalion: self-regulating stream processing in heron, Proc. VLDB Endow. 10 (12) (2017) 1825–1836.
[29] Nikos R. Katsipoulakis, Alexandros Labrinidis, Panos K. Chrysanthis, A holistic view of stream partitioning costs, Proc. VLDB Endow. 10 (11) (2017) 1286–1297.
[30] Xiao Huang, Yu Jiang, Hao Fan, Huayun Tang, Yiping Wang, Jin Jin, Hai Wan, Xibin Zhao, TATA: Throughput-aware task placement in heterogeneous stream processing with deep reinforcement learning, in: 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, 2021, pp. 44–54.
[31] Giselle van Dongen, Dirk Van den Poel, Evaluation of stream processing frameworks, IEEE Trans. Parallel Distrib. Syst. 31 (8) (2020) 1845–1858.
[32] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, Timothy Roscoe, Megaphone: Latency-conscious state migration for distributed streaming dataflows, Proc. VLDB Endow. 12 (9) (2019) 1002–1015.
[33] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, Azalia Mirhoseini, GAP: Generalizable approximate graph partitioning framework, 2019, pp. 1–13, arXiv: 1903.00614.
[34] Zhihao Wu, Xincan Lin, Zhenghong Lin, Zhaoliang Chen, Yang Bai, Shiping Wang, Interpretable graph convolutional network for multi-view semi-supervised learning, IEEE Trans. Multimed. 25 (2023) 8593–8606.
[35] Ying Cui, Chao Shao, Li Luo, Liguo Wang, Shan Gao, Liwei Chen, Center weighted convolution and graphsage cooperative network for hyperspectral image classification, IEEE Trans. Geosci. Remote Sens. 61 (2023) 1–16.
[36] Apache storm, 2021, https://storm.apache.org/2021/10/11/storm124-released.html.
[37] Aliyun, 2021, https://tianchi.aliyun.com/dataset/.
[38] Qihang Wang, Decheng Zuo, Zhan Zhang, Siyuan Chen, Tianming Liu, An adaptive non-migrating load-balanced distributed stream window join system, J. Supercomput. 79 (2023) 8236–8264.
[39] Shunjie Zhou, Fan Zhang, Hanhua Chen, Hai Jin, Bing Bing Zhou, FastJoin: A skewness-aware distributed stream join system, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 1042–1052.
[40] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garca-Soriano, Nicolas Kourtellis, Marco Serafini, The power of both choices: Practical load balancing for distributed stream processing engines, in: 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 137–148.
[41] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, Roy Campbell, R-storm: Resource-aware scheduling in storm, in: Proceedings of the 16th Annual Middleware Conference, 2015, pp. 149–161.

Minghui Wu is a Ph.D. 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.

Dawei Sun is a Professor in the School of Information Engineering, China University of Geosciences, Beijing, P.R. China. He received his Ph.D. degree in computer science from Northeastern University, China in 2012, and conducted the Postdoctoral research in the department of computer science and technology at Tsinghua University, China in 2015. His current research interests include big data computing, cloud computing and distributed systems. In these areas, he has authored over 90 journal and conference papers.

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

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored over 750 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 169 with 152,400+ 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.