A Prediction-Driven Collaborative Scheduling Strategy for Distributed Stream Computing Systems

Minghui Wu[1], Dawei Sun[1]*, Xiaoxian Wang[1], Shang Gao[2], Rajkumar Buyya[3]
[1]School of Artificial Intelligence, China University of Geosciences, Beijing, 100083, China
wuminghui@email.cugb.edu.cn; sundaweicn@cugb.edu.cn; wangxiaoxian@email.cugb.edu.cn
[2]School of Information Technology, Deakin University, Waurn Ponds, Victoria, 3216, Australia
shang.gao@deakin.edu.au
[3]Quantum Cloud Computing and Distributed Systems (qCLOUDS) Lab, School of Computing and Information Systems,
The University of Melbourne, Grattan Street, Parkville, Victoria, 3010, Australia
rbuyya@unimelb.edu.au

2025 IEEE 31th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3315-4901-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICPADS67057.2025.11323167

Abstract—**Multi-objective collaborative optimization is essential for improving performance in stream computing systems. However, existing approaches often neglect the interdependencies among communication overhead, load balancing, and energy consumption, and lack predictive capabilities, resulting in delayed scheduling decisions that degrade system latency and throughput. To overcome these limitations, we propose a prediction-driven collaborative framework, named Pc-Stream, which proactively identifies overloaded compute nodes and triggers task migrations in advance. This paper presents this strategy through two key components: (1) A temperature-driven neighborhood adjustment method for task topology partitioning. This method dynamically adjusts the number of migrated tasks based on a predefined temperature. Tasks with high communication volume are batch-migrated to nodes with lower utilization rates during the high-temperature phase, and migrated individually during the low-temperature phase. (2) A sliding window mechanism that generates multiple sub-sequences for training multiple predictive models. These models enable the system to monitor load trends and proactively migrate tasks from overloaded nodes to those with sufficient resources, thereby reducing communication costs and improving load balance. Experimental results demonstrate that, under dynamic and fluctuating data stream conditions, Pc-Stream significantly enhances overall system performance: reducing average system latency by 49.9%, and increasing average throughput by 16.9%.**

Index Terms—**Stream computing systems, Task migration, Topology partitioning, Predictive models.**

I. Introduction

Amidst the ongoing digital transformation, enterprises are increasingly demanding real-time data analysis and millisecond-level decision-making capabilities [1], driving the gradual shift from traditional batch processing models to streaming processing architectures [2]. These architectures are capable of continuously processing dynamically generated, unbounded data streams and delivering analytical results in near real time [3]. To address the real-time challenges posed by high-velocity data streams, stream computing systems employ low-latency processing and high fault-tolerance mechanisms to unlock the full value of dynamic data streams [4].

Effective scheduling in stream computing systems is crucial for system latency and throughput [5]. Some researchers [6]–[8] have attempted to optimize scheduling strategies from a single-objective perspective, primarily aiming to reduce cross-node data transmission by co-locating communication-intensive tasks on the same compute nodes. Although this can effectively reduce communication overhead, it often leads to significant workload imbalances across the cluster. Some nodes with ample resources may remain under-loaded due to insufficient task allocation, while others become overloaded, resulting in slower task processing or even downtime [9].

To address this challenge, prior studies [8], [10], [11] have explored dynamic scheduling algorithms that optimize resource utilization through task migration. These methods, however, typically employ a reactive trigger-response mechanism [5]. That is, task migrations are only initiated after a resource bottleneck is detected, for example, when a node's utilization exceeds a preset threshold or when a migration reduces inter-node communication overhead [12]. Unfortunately, such reactive strategies often lag behind sudden workload spikes [13], [14], as task reallocation only occurs after overload conditions have materialized. In contrast, if node loads can be accurately predicted, proactive resource allocation and task deployment can be implemented in advance to prevent performance degradation and maintain system stability.

Motivated by these observations, we propose a prediction-driven collaborative framework, Pc-Stream, which aims to continuously optimize system performance by anticipating future changes in node workloads. Pc-Stream employs proactive resource prediction techniques to generate task allocation plans in advance based on load trends observed within varying time windows. By jointly considering communication overhead, load balancing, and resource efficiency, it strives to co-locate communication-intensive tasks while reducing cross-node communication and promoting balanced and efficient resource utilization.

To continuously monitor task-level data flow changes in real time, Pc-Stream adopts a multi-model weighted XGBoost load prediction model to forecast the load conditions of each node. Based on these predictions, Pc-Stream proatively adjusts task deployments to optimize resource utilization and scheduling responsiveness. Through this prediction-driven collaboration, Pc-Stream maintains a prolonged online operation and effectively handles fluctuating data streams, while supporting high throughput and fast response time.

The key contributions of this paper are as follows:

(1) We construct a stream application model to identify task-level communication dependencies, a resource model to match task demands with available node resources, and a prediction model to analyze load variations across compute nodes.
(2) We design a heuristic algorithm that jointly considers communication overhead, load balancing, and resource utilization. This algorithm co-locate tasks with intensive communication on the same nodes, balances workloads across the cluster, and aligns task demands with available resources.
(3) We develop a multi-model weighted XGBoost prediction model to forecast future node load variations. This model uses multiple time windows to train a set of XGBoost models, capturing both short-term load fluctuations and long-term load trends. Task deployments are dynamically adjusted based on these predictions.
(4) We implement Pc-Stream on the Apache Storm platform and evaluate system performance using metrics such as system latency and throughput. Experimental results demonstrate Pc-Stream's superior performance in optimizing system throughput and reducing latency.

The rest of the paper is organized as follows: Section II reviews related work on streaming application scheduling and load prediction. Section III introduces the Pc-Stream system model, including models for stream application, resources, and load prediction. Section IV explains Pc-Stream and its core algorithms. Section V evaluates system performance using system throughput and latency metrics. Section VI concludes the paper and outlines future directions.

II. Related Work

Distributed stream computing systems require continuous data processing and have high demands for resource reliability, making the development and implementation of effective task scheduling algorithms particularly challenging [15]. Task allocation in such systems is an NP-hard problem [16], which means that obtaining globally optimal solutions within reasonable time constraints is computationally infeasible. To address these challenges, researchers have proposed various scheduling strategies, mainly including topology-aware scheduling, resource-aware scheduling, and proactive scheduling [17].

Topology-aware scheduling employs graph partitioning algorithms to allocate tasks in streaming applications by minimizing cut edges and balancing the load. Tc-Storm [18] integrates graph partitioning with the characteristics of data flow within the topology to construct an adaptive resource allocation model under multiple constraints and generates a suboptimal deployment solution. However, this method lacks sufficient responsiveness to dynamic changes in data streams, which makes its performance degrade under significant load fluctuations. SP-ant [8] proposes a scheduling strategy based on the ant colony algorithm, which dynamically allocates resources in heterogeneous clusters by combining global search and local optimization search. However, the algorithm stops evolving once it reaches convergence, making it difficult to handle sudden load surges. Moreover, the cost function does not fully consider the centralized deployment of adjacent operators, potentially leading to increased communication overhead between nodes.

Resource-aware scheduling intelligently deploys tasks to compute nodes based on the workload of nodes and the resource requirements of tasks. R-Storm [6] proposes a resource-aware scheduler that categorizes resource constraints into soft and hard constraints and optimizes task allocation based on resource state vectors and Euclidean distance. However, this algorithm is not suitable for heterogeneous environments and neglects the communication overhead between tasks. D-Storm [7] makes adaptive adjustments based on runtime workload variations. It employs the First-Fit Decreasing bin-packing heuristic algorithm, taking into account the communication between tasks and the load impact on compute nodes to minimize cross-node data transmission and optimize resource utilization. However, the overhead of task reallocation is significant, which can increase scheduling delays in high-load environments.

Proactive scheduling predicts system states and workload variations to make task allocation and resource adjustment decisions in advance. To meet the latency requirement with the minimal energy cost, Pec [19] proposes a proactive elastic resource scheduling strategy for computation-intensive and communication-intensive applications. It uses a collaborative workload prediction model to accurately forecast the upcoming workload. An energy-efficient resource pre-allocation method is designed to adjust the CPU frequency and reconfigure the resource in the cluster. Similarly, to ensure continuous data processing without interruptions, PRM [13] predicts load variations and resource scarcity in clusters. It allows for customizable service-specific metrics, which are used by the load predictor to anticipate resource consumption peaks and proactively allocate resources. However, these efforts do not take into account the load relationships among tasks in stream applications, limiting their effectiveness in improving overall system performance.

In summary, proactive workload prediction for optimizing resource allocation in clusters has been explored. However, many existing approaches either suffer from low prediction accuracy, high computational overhead, or focus on single-objective optimization. To address these limitations, we leverage XGBoost [20], a gradient boosting framework known for its strong capability in nonlinear modeling, high prediction accuracy, and efficient parallel training, which are essential for capturing complex patterns in workload behaviors. To address the short-term load volatility in dynamic cluster environments, we introduce a sliding window mechanism to enable the model to focus on recent trends and adapt quickly to changes, thereby improving workload prediction under fluctuating conditions.

A comparison of four key aspects (Communication-aware,

TABLE I
RELATED WORK COMPARISON

Related work	Aspects			
	Communication	Resource	Adaptive	Predictive
Tc-Storm [18]	✓	✓	✓	×
SP-ant [8]	✓	✓	✓	×
R-Storm [6]	×	✓	✓	×
D-Storm [7]	✓	✓	✓	×
I-Scheduler [10]	✓	✓	✓	×
PRM [13]	×	✓	✓	✓
Our work	✓	✓	✓	✓

Resource-aware, Adaptive, and Predictive) between Pc-Stream (our work) and relevant research is summarized in Table I.

III. SYSTEM MODEL

Before introducing the Pc-Stream framework and its related algorithms, we first explain the stream application model, resource model, and prediction model in distributed stream computing environments.

A. Stream Application Model

The logical topology of a stream application can be modeled as a directed acyclic graph (DAG), denoted as $G = \{V, E\}$ [21], where the vertex set $V = \{v_1, v_2, ..., v_k\}$ represents the operators in the topology and the directed edge set $E = \{e_{v_{i,m},v_{j,b}} | v_i, v_j \in V\}$ represents the data transmission relationships between task $v_{i,m}$ in the operator v_i and task $v_{j,b}$ in the operator v_j. Each operator v_i has a set of tasks denoted as $v_i = \{v_{i,1}, v_{i,2}, ..., v_{i,m}, ..., v_{i,h}\}$, and the number of tasks defines the parallelism of the operator. These tasks are executed in parallel across different workers of compute nodes.

For a cluster with N compute nodes, data transmission between nodes can incur significant communication overhead. We define the communication cost $Comm_{cost}$ of node n_t ($n_t \in N$) to measure the quality of task deployment, which can be calculated by Eq. (1).

$$Comm_{cost}(n_t) = \sum_{v_{i,m} \vee v_{j,b} \in Task(n_t)} e_{v_{i,m},v_{j,b}} \cdot x_{v_{i,m},v_{j,b}}(n_t), \tag{1}$$

where $e_{v_{i,m},v_{j,b}}$ denotes the data transmission volume between task $v_{i,m}$ and task $v_{j,b}$ in the topology. $Task(n_t)$ represents the set of tasks that are deployed on compute node n_t. $x_{v_{i,m},v_{j,b}}(n_t)$ is a binary decision variable used to indicate whether tasks $v_{i,m}$ and $v_{j,b}$ are assigned to compute node n_t. If the two tasks are assigned to the same node (i.e., n_t), then $x_{v_{i,m},v_{j,b}}(n_t) = 0$; otherwise, $x_{v_{i,m},v_{j,b}}(n_t) = 1$.

When task $v_{j,b}$ migrates from its current compute node to the target node n_t, the communication overhead between these two nodes changes. We define candidate nodes as those that remain non-overloaded after accepting task $v_{j,b}$. The communication cost change $\Delta(v_{j,b}, n_t)$ of each candidate node n_t can be calculated using Eq. (2).

$$\Delta(v_{j,b}, n_t) = Comm_{cost}(n_t) - Comm'_{cost}(n_t), \tag{2}$$

where $Comm_{cost}(n_t)$ and $Comm'_{cost}(n_t)$ denote the communication cost of compute node n_t before and after the migration, and can be calculated by Eq. (1). If $\Delta(v_{j,b}, n_t) \leq 0$, migrating task $v_{j,b}$ to node n_t increases the communication cost, and such a migration is therefore not recommended. If $\Delta(v_{j,b}, n_t) > 0$, the migration reduces communication overhead, indicating that it improves system communication efficiency. Based on this criterion, we identify the set of tasks eligible for migration, denoted as CT.

During scheduling, all tasks in the set CT with planned deployment changes are migrated concurrently. If $\sum_{v_{j,b} \in CT, n_t \in N} \Delta(v_{j,b}, n_t) > 0$, this scheduling action is beneficial for reducing overall system communication overhead.

To avoid a large span in communication costs between different nodes, we normalize $\Delta(v_{j,b}, n_t)$, which can be calculated using Eq. (3).

$$Comm_{\text{score}}(v_{j,b}, n_t) = \begin{cases} \frac{\Delta(v_{j,b},n_t) - \Delta_{\min}}{\Delta_{\max} - \Delta_{\min}}, & \Delta(v_{j,b}, n_t) > 0 \\ 0, & \Delta(v_{j,b}, n_t) \leq 0 \end{cases} \tag{3}$$

where the communication score $Comm_{\text{score}}(v_{j,b}, n_t)$ falls within the interval $[0, 1]$. $\Delta_{\max}$ and $\Delta_{\min}$ represent the minimum and maximum values of communication change in all migrated tasks, respectively.

B. Resource Model

In a distributed stream computing system, task execution relies on the computational resources of nodes, particularly CPU and memory [22]. Given $Task(n_t)$ represents the set of tasks deployed on node n_t, the CPU requirements $Cpu(n_t)$ and memory requirements $Mem(n_t)$ of $Task(n_t)$ can be calculated using Eq. (4).

$$\begin{cases} Cpu(n_t) & = \sum_{v_{j,b} \in Task(n_t)} Cpu_{v_{j,b}} \\ Mem(n_t) & = \sum_{v_{j,b} \in Task(n_t)} Mem_{v_{j,b}} \end{cases} \tag{4}$$

where $Cpu_{v_{j,b}}$ denotes the CPU consumption of task $v_{j,b}$ in $Task(n_t)$. $Mem_{v_{j,b}}$ denotes the memory consumption of task $v_{j,b}$ in $Task(n_t)$. To ensure the execution of tasks on node n_t, the resource requirements of $Task(n_t)$ don't exceed the node's available resources, and the resource allocation satisfies constraint Eq. (5).

$$\begin{cases} Cpu(n_t) & \leq Cpu_{avail}(n_t), \\ Mem(n_t) & \leq Mem_{avail}(n_t) \end{cases} \tag{5}$$

where $Cpu_{avail}(n_t)$ and $Mem_{avail}(n_t)$ represent the available CPU and memory resources of compute node n_t, respectively.

The scheduler needs to reasonably distribute tasks to different compute nodes based on the information to achieve efficient resource utilization. After the task assignment is completed, the remaining CPU $Cpu_{rem}(n_t)$ and memory $Mem_{rem}(n_t)$ resources of compute node n_t can be calculated using Eq. (6).

$$\begin{cases} Cpu_{rem}(n_t) & = Cpu_{avail}(n_t) - Cpu(n_t) \\ Mem_{rem}(n_t) & = Mem_{avail}(n_t) - Mem(n_t) \end{cases} \tag{6}$$

The remaining resources reflect the degree of resource utilization of compute node n_t under the current task allocation. To more comprehensively evaluate the availability of node resources, we define the resource score to quantify the utilization of compute nodes, thus providing a decision-making basis for task scheduling. It can be calculated using Eq. (7).

$$Res_{score}(n_t) = \alpha \cdot \frac{Mem_{rem}(n_t)}{Mem_{total}(n_t)} + (1-\alpha) \cdot \frac{Cpu_{rem}(n_t)}{Cpu_{total}(n_t)} \tag{7}$$

where $Cpu_{total}(n_t)$ and $Mem_{total}(n_t)$ respectively represent the total CPU resources and total memory resources of compute node n_t, and α denotes a user-defined weighting factor, $\alpha \in [0,1]$.

We use the load balancing degree $Load\_Balance\_Degree(n_t)$ to measure the load balance level of the target compute node n_t. If the load of n_t significantly deviates from the system average, it indicates an imbalance in its load state, suggesting that task migration may be necessary. The load balancing degree can be calculated using Eq. (8).

$$Load\_Balance\_Degree(n_t) = \frac{|Res_{score}(n_t) - \mu_{Res}|}{\mu_{Res}} \tag{8}$$

where μ_{Res} denotes the average system load across all compute nodes, and is calculated by Eq. (9).

$$\mu_{Res} = \frac{1}{N}\sum_{i=1}^{N} Res_{score}(n_i), \tag{9}$$

where N denotes the total number of compute nodes in the cluster.

The load on some nodes may be much higher than the average level, resulting in significant disparities among different nodes during the scoring process. Therefore, we normalize the load balancing degree $Load\_Balance\_Degree(n_i)$ to obtain a load balancing score $Balance_{score}(n_t)$, and it can be calculated using Eq. (10).

$$Balance_{score}(n_t) = \frac{1}{1 + Load\_Balance\_Degree(n_t)} \tag{10}$$

A larger $Balance_{score}(n_t)$ indicates a higher degree of load balance for the node.

C. Prediction Model

The load of a compute node is influenced by multiple factors, such as task load, node processing capability, and data transmission rate. These factors interact in complex ways, making it difficult to predict using data from a single time point [19]. To address this, we segment the data using a sliding window and perform time series modeling within each window. This enables the prediction model to accurately learn the historical load variation patterns.

Our objective is to learn a prediction function f that maps the historical data X within the time range [0, T] to the future value $X(T+1)$.

$$[X(1), X(2), ...X(k), ..., X(T)]^{\mathbf{T}} \xrightarrow{f} X(T+1), \tag{11}$$

where $X(k)$ is a feature vector, defined as $X(k) = [x_{k1}, x_{k2}]$, and x_{k1} and x_{k2} denote CPU utilization and memory utilization, respectively.

IV. Pc-Stream: Architecture and Algorithms

Based on the models constructed above, we propose Pc-Stream, a prediction-driven collaborative scheduling framework. This section introduces its architecture and the algorithms for subgraph partitioning and proactive scheduling.

A. System Architecture

Built on the Apache Storm platform, Pc-Stream consists of three core components: Nimbus, ZooKeeper, and Supervisor. As shown in Fig. 1, Nimbus receives topology tasks submitted by users and assigns the tasks to compute nodes in the cluster via its scheduler. ZooKeeper serves as the coordination and state management component, which maintains communication between Nimbus and worker nodes, ensuring coordination and consistency among different components in the system. Supervisor manages the workers on each compute node, ensuring that each worker can stably execute computational tasks.

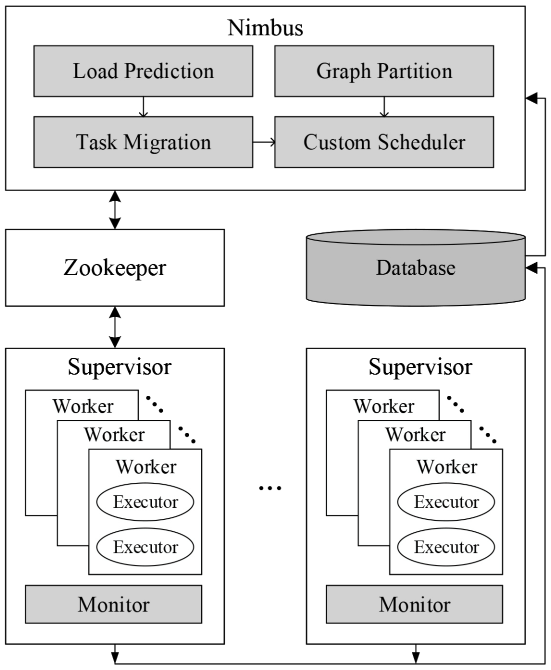

Fig. 1. Architecture of Pc-Stream.

Pc-Stream extends Apache Storm by introducing four additional modules: a Monitor module, a Graph Partition module, a Load Prediction module, and a Task Migration module.

The **Monitor** module collects real-time CPU and memory usage data of each node and stores it in the database for subsequent optimized scheduling.

The **Graph Partition** module divides the topology graph into subgraphs during system runtime. By analyzing task-level communication and resource utilization, it optimizes task allocation by co-locating frequently communicating tasks and balancing the computational load across nodes.

The **Load Prediction** module forecasts the future load of compute nodes using historical resource data collected by the **Monitor** module. It proactively identifies compute nodes that are likely to become overloaded in the future, thereby triggering task migration in advance.

Based on the predicted resource load information, the **Task Migration** module proactively adjusts task deployment. It ranks these nodes based on their predicted load levels and prioritizes the most urgent cases. It then selects tasks from the most heavily loaded node using a communication cost score. Finally, the selected task is migrated to the node with the lowest current load to achieve the most efficient load balancing.

Through the collaborative operation of these modules, Pc-Stream effectively enhances system adaptability to dynamic workloads and reduces scheduling overhead. Prediction-based task migration reduces the need to frequently re-execute the costly graph partitioning module. However, repeated local migrations may gradually increase cross-node communication. Once the communication metric exceeds a user-defined threshold, the graph partitioning module is invoked again to globally re-optimize task placement. In this cooperative workflow, the graph partitioning module establishes a communication-efficient baseline, predictive scheduling performs lightweight, prediction-driven adjustments, and repartitioning is only re-triggered when the accumulated adjustments degrade communication beyond the acceptable limit.

B. Subgraph Partitioning Algorithm

We employ a simulated annealing algorithm in the **Graph Partition** module that adjusts the neighborhood size based on temperature to perform subgraph partitioning for stream applications in a heterogeneous cluster environment.

During the annealing process, the neighborhood size $Nbr_{migrate}$ is adjusted dynamically according to the current temperature $Temp$, and can be calculated by Eq. (12).

$$Nbr_{migrate} = \lfloor Nbr_{\min} + \left(\frac{Temp}{Temp_0}\right) \cdot (Nbr_{\max} - Nbr_{\min}) \rfloor, \tag{12}$$

where $Nbr_{\min}$ and $Nbr_{\max}$ represent the minimum and maximum neighborhood sizes at low and high temperatures, respectively. $Temp_0$ denotes the user-set initial temperature. In the high-temperature phase, a larger neighborhood enables batch migrations for global exploration, while in the low-temperature phase, a smaller neighborhood focuses on local fine-tuning.

An initial random deployment plan is generated as the starting solution, which is iteratively refined to search for the optimal solution using an objective function Z. This objective function comprehensively considers both communication overhead and resource load, and is calculated using Eq. (13).

$$\begin{aligned} Z = \beta \cdot \sum_{v_{j,b} \in V, n_t \in N} Comm_{score}(v_{j,b}, n_t) \\ + (1-\beta) \cdot \sum_{n_t \in N} Balance_{score}(n_t) \end{aligned} \tag{13}$$

where N denotes the set of compute nodes in the cluster, and $\beta \in [0, 1]$ is a user-defined weighting factor that balances the importance between communication overhead and load balancing.

Algorithm 1 outlines the process of dividing the DAG into subgraphs, with the aim of minimizing system communication delay while maintaining a relatively balanced workload across nodes.

The input of Algorithm 1 includes the given task graph $G = (V, E)$, initial temperature $Temp_0$, minimum temperature $Temp_{\min}$, cooling coefficient $\eta \in (0, 1)$, neighborhood range $[Nbr_{\min}, Nbr_{\max}]$, temperature-stage threshold δ, convergence threshold ε, and weighting factor β. The output is the optimal task assignment P_{best}.

Steps 2-4 initialize the temperature and randomly generate the initial solution. Step 5 initializes Δ_{avg} value. To ensure the algorithm enters the body of the `While` loop on its first execution, Δ_{avg} must be greater than ε. Steps 7-11 generate a new solution P' based on the current solution P and the neighborhood size. If $\Delta_{avg} > \delta$, the neighborhood size is set to Nbr_{max} to enable broad exploration of the solution space; otherwise, it is set to Nbr_{min} for fine-grained search. Step 12 calculates the objective function of the new solution P'.

In steps 13-21, if the objective function value Z' of the new solution P' improves upon that of the current solution P, the new solution is accepted directly as the current solution; otherwise, acceptance of the inferior solution is determined probabilistically.

Step 22 calculates the average change Δ_{avg} in the objective function values, while Step 23 updates the temperature $Temp$ according to the cooling schedule before proceeding to the next iteration.

C. Proactive Scheduling

Due to its static modeling mechanism, the traditional single XGBoost model [20] struggles to fully analyze and utilize the dynamic information within time-series data. When confronted with highly dynamic load data, it fails to effectively grasp the overall trend of the data. To enhance the model's adaptability to temporal load changes and improve its prediction robustness, we construct a prediction model in the **Load Prediction** module based on multi-model ensemble.

As shown in Fig. 2, for each time window, an independent XGBoost sub-model is trained, thereby building a set of sub-models with the capability to perceive temporal differences. To synthesize the predictive information contained in different time windows, we design a window-weighting mechanism. This mechanism dynamically adjusts the weights of each sub-model in the final prediction according to strategies such as

```
Algorithm 1: Graph Partition
Input: Task graph $G = (V, E)$; initial temperature
       $Temp_0$; minimum temperature $Temp_{\min}$;
       cooling coefficient $\eta \in (0, 1)$; neighborhood
       range $[Nbr_{\min}, Nbr_{\max}]$; temperature-stage
       threshold $\delta$; convergence threshold $\varepsilon$; weighting
       factor $\beta$
Output: Optimal task assignment $P_{\text{best}}$
1  Initialization:
2    Generate an initial solution $P$ randomly;
3    $Z \longleftarrow$ Compute objective function using Eq. (13);
4    $Temp \leftarrow Temp_0$, $P_{\text{best}} \leftarrow P$, $Z_{\text{best}} \leftarrow Z$, count = 0;
5    $\Delta_{\text{avg}} = 10$; /* Initial value of $\Delta_{\text{avg}}$ is
       greater than $\varepsilon$.                              */
6  while $Temp > Temp_{\min}$ and $\Delta_{avg} > \varepsilon$ do
7      if $\Delta_{avg} > \delta$ then
           /* GenerateNewSolution function
              controls the extent of change
              in the new solution $P'$ based
              on the current solution $P$     */
8          Generate new solution
            $P' \leftarrow$ GenerateNewSolution$(P, Nbr_{\max})$;
9      else
10         Generate new solution
            $P' \leftarrow$ GenerateNewSolution$(P, Nbr_{\min})$;
11     end
12     $Z' \longleftarrow$Compute objective function of $P'$ using Eq.
        (13);
13     if $Z' < Z$ then
14         $Z \leftarrow Z'$;
15         $P_{\text{best}} \leftarrow P'$;
16     else
           /* Probability of accepting $P'$
              */
17         Random $r$, $r \in [0, 1]$;
18         if $r < e^{\frac{Z'-Z}{Temp}}$ then
19             $P_{\text{best}} \leftarrow P'$;
20         end
21     end
22     count++; $\Delta_{\text{avg}} = \frac{1}{\text{counter}} \sum_i^{\text{counter}} |Z' - Z|$;
23     $Temp \leftarrow \eta \cdot Temp$;
24 end
25 return $P_{best}$
```

window position, model performance, or time-decay factors, aiming to achieve the optimal fusion of multi-model prediction results. Through this approach, the model can more accurately respond to load changes at different time scales and enhance its ability to perceive sudden load fluctuations and long-term trends.

Before training the model, we define the load impact coefficient LIC to measure the dynamic correlation between features and the load. It can be calculated using Eq. (14).

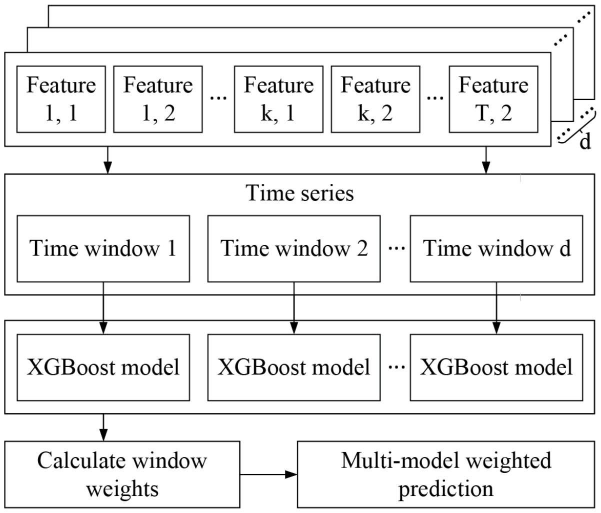

Fig. 2. Multi-model weighted XGBoost prediction model.

$$LIC_{X,Res_{score}(n_t)} = \frac{\sum_{k=1}^{T} \frac{e^{-\lambda(T-k)} \cdot \Delta X(k) \cdot \Delta Res_{score}(n_t,k)}{\max(\Delta X) \cdot \max(\Delta Res_{score}(n_t,k))}}{\sum_{k=1}^{T} e^{-\lambda(T-k)}} \tag{14}$$

where $\Delta X(k)$ denotes the rate of change in feature value at time k relative to the previous time point $k-1$, i.e., $\Delta X(k) = X(k) - X(k-1)$. $\Delta X(k)$ is used to measure feature fluctuations over time and capture the load variation trend. $\Delta Res_{score}(n_t, k)$ represents the rate of change in resource load of compute node n_t at time k, i.e., $\Delta Res_{score}(n_t, k) = Res_{score}(n_t, k) - Res_{score}(n_t, k-1)$, where $Res_{score}(n_t, k)$ denotes the resource load $Res_{score}(n_t)$ of node n_t at time k. λ is the decay coefficient, which controls the rate at which the weights decay.

Algorithm 2 outlines the process of predicting future loads on compute nodes for proactive scheduling. It enables dynamic migration and optimized allocation of tasks before system resources approach the bottleneck.

The input of Algorithm 2 is the feature data collected by the Monitor module and load impact threshold. The output is the load prediction value for the future time.

Step 2 initializes the feature set and the predicted load. Steps 3-8 calculate the load impact coefficient for each feature, assessing its importance for load prediction. Step 9 divides the time-series data into d subsets. In each time window, the training data size is determined by the ratio of the total number of elements in dataset X to the number of windows. Steps 10-13 independently train each XGBoost model on its corresponding subset. Steps 14-16 calculate the exponential decay weights for each XGBoost model. Step 17 normalizes the weights. Step 18 integrates the prediction results from all XGBoost models by feeding the weighted outputs into a feedforward neural network, which learns context-aware, non-linear interactions among the predictors and dynamically

Algorithm 2: Multi-model weighted XGBoost Prediction

```
Input: Feature data $X$ collected by the Monitor
       module; Load impact threshold $\psi$
Output: Predicted load at future time $T+1$: $X(T+1)$
1  Initialization:
2      $X \leftarrow \emptyset; Weights \leftarrow \emptyset; X(T+1) \leftarrow 0$;
3  for each $X(k)$ in $X$ do
4      $LIC \leftarrow$ Compute load-impact coefficient using Eq. (14);
5      if $LIC > \psi$ then
6          $X(k) \leftarrow X(k) \cup \{LIC\}$;
7      end
8  end
9  Divide $X$ into $d$ data subsets based on the time series;
10 for $q = 1$ to $d$ do
11     Train $XGB_q$ on $X[q]$ ;
12     Store $XGB_q$ in model set;
13 end
14 for $q = 1$ to $d$ do
       /* Calculate the fusion weight $W_q$ for each model. */
15     $W_q \leftarrow \exp(-\lambda(d-q))$;
16 end
17 Normalize $W_q$ so that $\sum_{q=1}^{d} W_q = 1$;
18 Input the weighted model outputs into a feedforward
     neural network;
19 The feedforward neural network outputs the prediction
     results $X(T+1)$;
20 return $X(T+1)$;
```

emphasizes the most reliable model under varying workload conditions. Step 19 outputs the prediction result $X(T+1)$.

The prediction model forecasts the load of compute nodes based on this historical data and sends the prediction results to the **Task Migration** module. Upon receiving the data, the **Task Migration** module analyzes the load information of compute nodes. If the load of certain nodes exceeds a predefined threshold ϑ, it is added to an overload set $N_{overload} = \{n_1, n_2, ..., n_k\}$, and the overload values of these compute nodes are sorted in descending order (from highest to lowest). The sorted list of overloaded nodes ensures that compute nodes with the highest loads are prioritized for task migration. Meanwhile, it is necessary to further select which tasks need to be migrated from overloaded nodes. The communication score for each task can be calculated using Eq. (3), which represents the volume of data exchanged between each task in the overloaded node and other tasks. Finally, the task with the highest score is migrated to the node with the lowest resource utilization. We adopt a stop-and-restart migration mechanism: the task is terminated on the source node and then re-instantiated on the target node. The dominant migration overhead is the restart latency, which we measured to be approximately 10 seconds per migration, from task termination on the source to activation on the target.

V. Performance Evaluation

In this section, we evaluate the performance of Pc-Stream. The experimental setup is first discussed, followed by an analysis of system performance and prediction model accuracy.

A. Experimental Setup

Pc-Stream is deployed on the CentOS 7 operating system using Apache Storm 2.4.0. The cluster consists of 12 compute nodes, with 11 nodes configured as Supervisors, and the remaining nodes hosting Nimbus and Zookeeper services. To establish a heterogeneous computing environment, nodes are configured with varying CPU and memory capacities: five nodes are equipped with 1-core CPUs and 1GB of memory, while the other six nodes have 2-core CPUs and 2GB of memory.

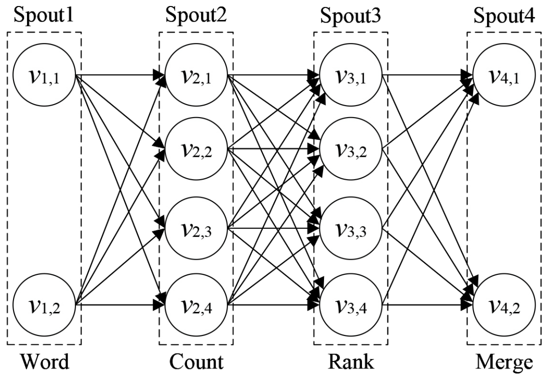

Fig. 3. Instance topology of TopN.

We use the Taobao user behavior dataset [23] provided by Alibaba Cloud, which contains 100 million records, to test system performance. A TopN streaming application is developed to identify best-selling products from this dataset. The topology of TopN is illustrated in Fig. 3. In the experiments, we compare the performance of Pc-Stream with two state-of-the-art schedulers: EvenScheduler and Tc-Storm [18]. EvenScheduler allocates tasks to compute nodes in a round-robin manner, while Tc-Storm co-locates communication-intensive tasks on the least-loaded compute nodes. Both are widely used as benchmark baselines for performance comparison.

B. System Performance

We evaluate the overall performance of Pc-Stream using two key metrics: system latency and system throughput. To simulate varying load conditions, we use different data stream rates: 5000 tuples/s, 1000 tuples/s, and 4000 tuples/s. We configure the predictor to produce a sequence of five forecasts at 60-second steps. This provides sufficient lead time for task migration before any node reaches its overload threshold.

Given a high input rate of 5000 tuples/s, Pc-Stream significantly outperforms both EvenScheduler and Tc-Storm. The Prediction module of Pc-Stream is capable of detecting overloaded nodes in advance and triggering proactive task

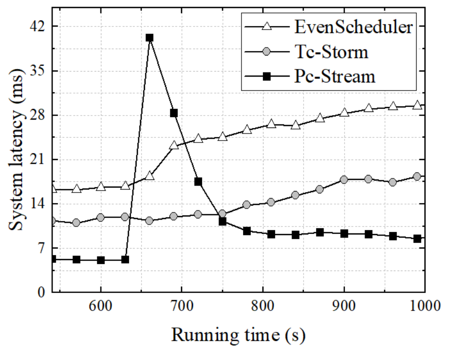

Fig. 4. System latency of TopN under high input rate (5000 tuples/s).

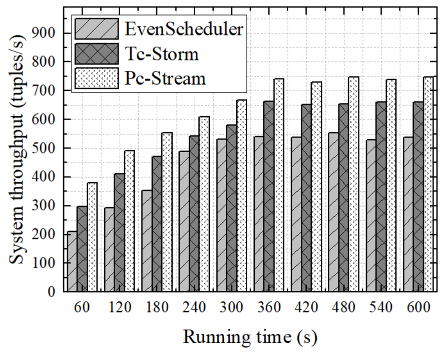

Fig. 5. Average throughput of TopN under low input rate (1000 tuples/s).

migrations. As shown in Fig. 4, at around 660 seconds, the system experiences a temporary increase in latency due to the triggered task rescheduling. During this process, tasks are paused while incoming data is buffered. When restarted, the tasks must process this backlog of data simultaneously, creating a workload burst that briefly spikes latency. However, once migration is completed, the latency rapidly drops and stabilizes at a low level of 9.8 ms. In contrast, EvenScheduler and Tc-Storm are unable to adjust their scheduling strategies promptly in response to the rising input rate, leading to escalating system load. The latency of EvenScheduler increases to 30.2 ms, while the latency of Tc-Storm rises from 11.3 ms to 18.1 ms.

Given a stable input rate of 1000 tuples/s, Pc-Stream achieves higher system throughput compared to EvenScheduler and Tc-Storm. As shown in Fig. 5, after the system stabilizes, EvenScheduler attains an average throughput of 538 tuples/s, and Tc-Storm reaches 657 tuples/s. In contrast, Pc-Stream achieves 741 tuples/s, representing a 12.8% improvement over Tc-Storm.

Similarly, given a higher input rate of 4000 tuples/s, Pc-Stream consistently outperforms the baselines. As shown in Fig. 6, after the system stabilizes, EvenScheduler achieves an average throughput of 2264 tuples/s, while Tc-Storm achieves 2762 tuples/s. Pc-Stream demonstrates the highest throughput of 3345 tuples/s, yielding a 21.1% improvement over Tc-Storm and a 47.7% improvement over EvenScheduler.

Table II presents an ablation study on the performance contribution of Pc-Stream's key components. The average

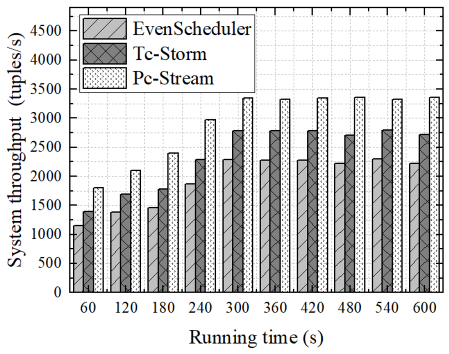

Fig. 6. Average throughput of TopN under high input rate (4000 tuples/s).

latency and throughput are evaluated under input rates of 5000 tuples/s and 1000 tuples/s, respectively. Pc-Stream integrates two core mechanisms: thermal-driven partitioning and predictive scheduling. The results show that thermal-driven partitioning alone already outperforms the baselines, reducing latency to 17.9 ms and increasing throughput to 655 tuples/s. The predictive scheduling component provides a further significant boost. When both components are combined, Pc-Stream achieves the best performance, reducing latency to 9.2 ms and increasing throughput to 741 tuples/s. These results demonstrate that while each mechanism individually contributes to performance, their integration yields the most significant overall gain.

TABLE II
CONTRIBUTION OF EACH COMPONENT TO SYSTEM PERFORMANCE

Components	Average latency	Average throughput
EvenScheduler	29.1 ms	538 tuples/s
Tc-Stream	18.3 ms	643 tuples/s
Thermal-driven partitioning	17.9 ms	655 tuples/s
Predictive scheduling	13.2 ms	701 tuples/s
Pc-Stream	9.2 ms	741 tuples/s

These experimental results demonstrate that Pc-Stream offers stronger scheduling responsiveness and resource adaptation efficiency, particularly under high-intensity data stream conditions.

C. Resource utilization

We measure resource utilization via the average CPU and memory consumption within the cluster, which indicates the algorithm's efficiency in managing resources. We evaluate energy consumption by the number of compute nodes required for the streaming application. The number of active nodes is closely correlated with the total energy consumption. A lower number of active nodes means lower overall energy consumption.

Given a high input rate of 5000 tuples/s, Pc-Stream is more resource efficient than both EvenScheduler and Tc-Storm. As shown in Fig. 7, Pc-Stream distributes the entire workload into 8 of the 10 nodes (Nodes 2-9). It uses these active nodes at a consistently high resource utilization of around 59%, while the other two nodes (10 and 11) remain completely idle (0%

usage). In comparison, Tc-Storm uses all 10 nodes but at a lower average resource utilization of 44%. EvenSchedule performs poorly, showing a major load imbalance where some nodes are overloaded at 70% while others are nearly unused at just 6%.

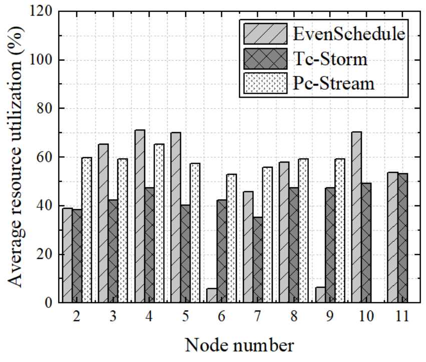

Fig. 7. Average resource utilization of compute node (5000 tuples/s)

Instead of distributing tasks across all machines, Pc-Stream puts streaming applications onto the smallest number of nodes required. This ensures that the active nodes are used to their full potential, avoiding the waste seen in the other methods. This approach has a clear benefit for energy consumption. Because Pc-Stream leaves some nodes completely idle, these machines can be put into a low-power sleep mode.

D. Prediction Accuracy

We select the data aggregation node that deploys the highest number of spout3 instances in Fig. 3 as the test subject, as its resource usage directly affects system performance. The experiment predicts the node's load under an increasing data input rate, comparing the predicted loads with the actual observed load values.

As illustrated in Fig. 8, the predicted values closely follow the actual values in terms of their overall trend. As the input rate increases, both the predicted and actual loads exhibit linear growth, indicating that the Pc-Stream prediction model effectively reflects the dynamic characteristics of load changes in response to varying data pressures. In the low-to-medium load range (2000-8000 tuples/s), the model accurately captures node load variations, with small prediction errors and relatively stable fluctuations. Even in the medium-to-high load range (above 8000 tuples/s), the model still maintains good prediction accuracy and stability, without significant deviations. This demonstrates its strong adaptability in modeling highly dynamic workloads.

Fig. 9 presents the prediction error analysis, using the Mean Absolute Percentage Error (MAPE) as the evaluation metric. The results indicate that the prediction error remains below 5% in most cases, with an average of 3.81%. These findings confirm the high prediction accuracy of the model and its ability to effectively track compute node load variations.

The model remains robust even under fluctuating input rates, as resource utilization is mainly influenced by the

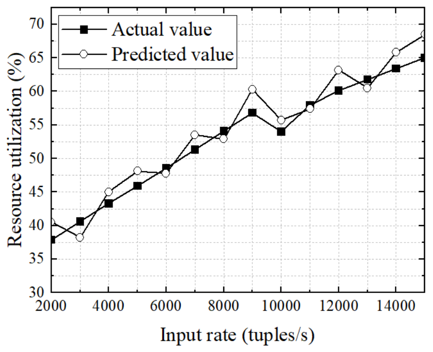

Fig. 8. Comparison between predicted and actual values.

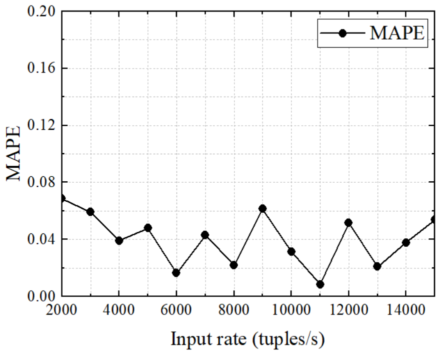

Fig. 9. MAPE between predicted and actual values.

instantaneous input rate rather than its long-term trend. The system responds to input changes smoothly, allowing the model to track variations effectively. Additionally, the training data includes a diverse range of input rates, which enhances the model's generalization capability.

VI. Conclusions and Future work

To address the trade-off between communication overhead and load balancing in distributed stream computing systems, we proposed a multi-objective collaborative scheduling strategy. By optimizing task allocation and enabling dynamic adjustments, this strategy reduces inter-task communication overhead while balancing resource utilization across compute nodes. By integrating load prediction into the scheduling process, the strategy can proactively identify potential computational bottlenecks and adjust task allocations in advance, allowing the system to better adapt to the dynamic nature of data streams.

As part of future work, we will explore the following two directions:

(1) Extending our implementation to other distributed stream computing platforms to further demonstrate the generality of Pc-Stream.
(2) Developing an efficient and reliable state management approach to further enhance the stability and adaptability of the scheduler in complex scenarios.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 62372419 and 62025203; and the Fundamental Research Funds for the Central Universities under Grant No. 265QZ2021001.

References

[1] Z. Zhang, T. Liu, Y. Shu, S. Chen, and X. Liu, "Dynamic adaptive checkpoint mechanism for streaming applications based on reinforcement learning," in *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, 2023, pp. 538–545.

[2] S. Chaturvedi, S. Tyagi, and Y. Simmhan, "Cost-effective sharing of streaming dataflows for iot applications," *IEEE Transactions on Cloud Computing*, vol. 9, no. 4, pp. 1391–1407, 2021.

[3] Q. Wang, D. Zuo, Z. Zhang, S. Chen, and T. Liu, "Sepjoin: A distributed stream join system with low latency and high throughput," in *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, 2023, pp. 633–640.

[4] J. Tan, Z. Tang, W. Cai, W. J. Tan, X. Xiao, J. Zhang, Y. Gao, and K. Li, "A cost-aware operator migration approach for distributed stream processing system," *IEEE Transactions on Cloud Computing*, vol. 13, no. 1, pp. 441–454, 2025.

[5] Z. Li, J. Cui, H. Song *et al.*, "Research on weighted adaptive particle swarm optimization algorithm based on flink stream processing framework," in *2024 3rd International Conference on Big Data, Information and Computer Network*, 2024, pp. 48–52.

[6] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, 2015, p. 149–161.

[7] X. Liu and R. Buyya, "D-storm: Dynamic resource-efficient scheduling of stream processing applications," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems*, 2017, pp. 485–492.

[8] M. Farrokh, H. Hadian, M. Sharifi, and A. Jafari, "Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters," *Expert Systems with Applications*, vol. 191, pp. 1–11, 2022.

[9] X. Liu and R. Buyya, "Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–40, 2020.

[10] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "I-scheduler: Iterative scheduling for distributed stream processing systems," *Future Generation Computer Systems*, vol. 117, pp. 219–233, 2021.

[11] M. Barika, S. Garg, A. Chan, and R. N. Calheiros, "Scheduling algorithms for efficient execution of stream workflow applications in multicloud environments," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 860–875, 2022.

[12] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Computing Surveys*, vol. 52, no. 2, 2019.

[13] G. Marques, C. Senna, S. Sargento, L. Carvalho, L. Pereira, and R. Matos, "Proactive resource management for cloud of services environments," *Future Generation Computer Systems*, vol. 150, pp. 90–102, 2024.

[14] S. M. Attallah, M. B. Fayek, S. M. Nassar, and E. E. Hemayed, "Proactive load balancing fault tolerance algorithm in cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 10, p. e6172, 2021.

[15] H. Peng, Y. Bai, G. Kang, Y. Li, and T. Chen, "Optimizing big data analytics architecture for edge computing using container technology," in *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*, 2024, pp. 254–261.

[16] M. Wu, D. Sun, Y. Cui, S. Gao, X. Liu, and R. Buyya, "A state lossless scheduling strategy in distributed stream computing systems," *Journal of Network and Computer Applications*, vol. 206, p. 103462, 2022.

[17] Y. Guo, H. Shan, S. Huang, K. Hwang, J. Fan, and Z. Yu, "Gml: Efficiently auto-tuning flink's configurations via guided machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2921–2935, 2021.

[18] J. Zhang, C. Li, L. Zhu *et al.*, "The real-time scheduling strategy based on traffic and load balancing in storm," in *2016 IEEE 18th International Conference on High Performance Computing and Communications*, 2016, pp. 372–379.

[19] X. Wei, L. Li, X. Li, X. Wang, S. Gao, and H. Li, "Pec: Proactive elastic collaborative resource scheduling in data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1628–1642, 2019.

[20] P. Mishra, A. Al Khatib, S. Yadav *et al.*, "Modeling and forecasting rainfall patterns in india: a time series analysis with xgboost algorithm," *Environmental Earth Sciences*, vol. 83, p. 163, 2024.

[21] H. Li, J. Xia, W. Luo, and H. Fang, "Cost-efficient scheduling of streaming applications in apache flink on cloud," *IEEE Transactions on Big Data*, vol. 9, no. 4, pp. 1086–1101, 2023.

[22] A. Muhammad, M. Aleem, and M. A. Islam, "Top-storm: A topology-based resource-aware scheduler for stream processing engine," *Cluster Computing*, vol. 24, pp. 417–431, 2021.

[23] "Tianchi," https://tianchi.aliyun.com/dataset/649?t=1679727494514, 2018.