Hindawi
Mobile Information Systems
Volume 2020, Article ID 5351824, 17 pages
https://doi.org/10.1155/2020/5351824

Research Article

Flink-ER: An Elastic Resource-Scheduling Strategy for Processing Fluctuating Mobile Stream Data on Flink

Ziyang Li,[1] Jiong Yu,[1] Chen Bian ,[2] Yonglin Pu,[1] Yuefei Wang,[1] Yitian Zhang,[3] and Binglei Guo[1]

[1]*School of Information Science and Engineering, Xinjiang University, Urumqi 830046, China*
[2]*College of Internet Finance and Information Engineering, Guangdong University of Finance, Guangzhou 510521, China*
[3]*School of Software, Xinjiang University, Urumqi 830008, China*

Correspondence should be addressed to Chen Bian; bianchen0720@126.com

Received 1 December 2019; Accepted 1 May 2020; Published 20 May 2020

Academic Editor: Raul Montoliu

As real-time and immediate feedback becomes increasingly important in tasks related to mobile information, big data stream processing systems are increasingly applied to process massive amounts of mobile data. However, when processing a drastically fluctuating mobile data stream, the lack of an elastic resource-scheduling strategy limits the elasticity and scalability of data stream processing systems. To address this problem, this paper builds a flow-network model, a resource allocation model, and a data redistribution model as the foundation for proposing Flink with an elastic resource-scheduling strategy (Flink-ER), which consists of a capacity detection algorithm, an elastic resource reallocation algorithm, and a data redistribution algorithm. The strategy improves the performance of the platform by dynamically rescaling the cluster and increasing the parallelism of operators based on the processing load. The experimental results show that the throughput of a cluster was promoted under the premise of meeting latency constraints, which verifies the efficiency of the strategy.

1. Introduction

With the explosive development of mobile computing devices, the Internet of Things, and virtual reality (VR) [1], a large amount of data that requires immediate processing with extremely low latency has been generated. Meanwhile, big data stream processing systems (DSPSs) provide exact real-time data processing services. The statistical data from Seagate show that the total amount of data will reach 163 ZB before 2025 [2], a quarter of which will be generated by mobile information devices and require real-time processing [3]. Therefore, to handle real-time mobile streaming data that exhibit volatility, burstiness, disorder, and infiniteness [4], DSPSs have been widely applied in multiple scenarios of big data stream computing, especially for mobile data processing. In addition, due to advantages such as low latency, high-throughput performance, efficient fault tolerance [5, 6], and elegant backpressure mechanisms, Apache Flink [7, 8] has been widely used in industry and has drawn increasing attention in academia, making it superior to similar software programs and one of the most popular DSPSs in the mobile data-processing domain.

However, under stream computing fluctuations, Flink has disadvantages in terms of elasticity and scalability. The input load of stream processing usually fluctuates drastically over time, but the Flink cluster cannot be promptly rescaled. Therefore, a performance decline always occurs under peak load due to the insufficiency of computing resources, while energy overuse always occurs during times of low load. Since Flink does not provide an elastic resource-rescheduling strategy, performance decline and energy overuse have become serious challenges that have drawn much attention from the open-source community.

To address these problems, an elastic rescale methodology suitable for the Apache Flink architecture is proposed, and Flink with an elastic resource-scheduling strategy (Flink-ER) is developed. The specific contributions of this paper are as follows:

(1) We propose abstracting the stream computing topology as the model of a flow network. By taking it as the basis for building the resource allocation model and data redistribution model, we provide a theoretical foundation for developing Flink-ER.

(2) We propose the capacity detection algorithm based on the flow-network model. The algorithm calculates the initial capacity based on the network transmission performance and adjusts it by periodic feedback to determine an appropriate capacity for each edge, thus laying the foundation for the elastic resource reallocation algorithm.

(3) We propose the elastic resource reallocation algorithm based on the resource allocation model. This algorithm optimally dispatches the increasing load and creates the resource-rescheduling plan by identifying the performance bottleneck, thus improving the performance by effectively utilizing computing resources.

(4) We propose the data redistribution algorithm based on the data redistribution model. This algorithm reduces the overhead required to execute the resource-rescheduling plan by improving the efficiency of the checkpoint and restore process in Flink.

The rest of this paper is organized as follows. Section 2 presents models built based on the flow network as the theoretical foundation of Flink-ER. Section 3 details the three algorithms of the elastic rescaling strategy and its implementation. Section 4 evaluates the performance improvement and overhead of the strategy by comparing it with strategies from closely related studies. Section 5 introduces the strengths and weaknesses of related works. Section 6 briefly concludes the paper and discusses future work.

2. Model Topology

In this section, the basic paradigm of big data stream computing is first presented and modeled as a flow network by analyzing the capacity and flow of every edge in the DAG. Second, the resource allocation model is built based on a flow network to identify the bottleneck of a cluster and create the resource-rescheduling plan. Finally, the data redistribution model is developed by considering the features of stateful data in stream computing as a theoretical foundation for the data redistribution algorithm.

2.1. Stream Computing Paradigm. In the big data stream computing platform, data are directly processed in the memory of the computing nodes, whereas Figure 1 shows the user-defined function is used as an operator and the unprocessed data are sent by the source, sequentially processed by the operators and finally written into storage by a sink. Generally, the data source reads data from a message queue such as Kafka, while a data sink writes results into the Hadoop distributed file system (HDFS) or Redis et al. according to the practical requirements.

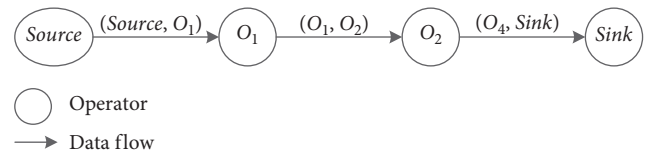

Figure 1: JobGraph of a streaming job.

However, in parallel stream processing engines such as Flink, the corresponding logic of each operator is deployed and executed in multiple nodes called vertices. Since Apache Flink adopts a master-slave architecture, the streaming job is represented as a JobGraph, as shown in Figure 1. However, in the parallel streaming processing platform, the ExecutionGraph of a job, as presented in Figure 2, serves as a guide for the JobManager to deploy the job and schedule tasks. Figure 2 illustrates that the topology structure of a streaming job is a DAG in which s represents the source, v_i represents vertices for processing the data, t_i represents the sinks used to output results, and (v_i, v_j) represents links used to transmit data between vertices. Generally, given that the processing logic of the operator O_i is deployed to vertices $v_1, v_2, \ldots, v_m$, these vertices are called the instances of the operator, defined as $v_i \in O_i$. In addition, given that the data transition between operators O_i and O_j is mapped by links (v_a, v_b), $(v_c, v_d), \ldots, (v_n, v_m)$, these links are called the instances of transition (O_i, O_j), defined as $(v_n, v_m) \in (O_i, O_j)$. Therefore, the relationships between the logical operator and physical vertex and between the logical transition and physical links are clarified and then used in the following descriptions.

2.2. Model of Flow Network. The stream computing paradigm shows that the data transition performance of edges and the processing capacity of vertices are crucial for achieving good cluster performance. The data that are not processed on time are stored in the buffer of the nodes and lead to data accumulation in Kafka. In other words, a lack of processing capability and redundancy of the input load lead to a latency increase. Therefore, the source corresponding to the most accumulated partition in Kafka formulates the flow-network paradigm [9]. For example, suppose that s_2 in the topology of Figure 2 leads to the most accumulations; then, the corresponding flow network is as presented in Figure 3.

As Figure 3 shows, the topology $G = (V, E)$ is a DAG with a single source. Set $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices in which $s \in S$ is the source and $t_i \in T$ are sinks. Set $E = \{(v_1, v_2), (v_3, v_4), \ldots, (v_n, v_m)\}$ is composed of data links between vertices. As shown in Figure 3, each edge $(v_i, v_j) \in E$ has a nonnegative capacity $c(v_i, v_j) \geq 0$, which represents the maximum value of the data transition on the link, and a nonnegative flow $f(v_i, v_j)$ represents the rate of data transition on the link at that moment. Furthermore, the constraint of $0 \leq f(v_i, v_j) \leq c(v_i, v_j)$ is strictly followed in flow networks, and the unit is tuple/s. The data processing capability $c(v_i, v_j)$ and data transition rate $f(v_i, v_j)$ are shown in Figure 3. In addition, since each vertex represents

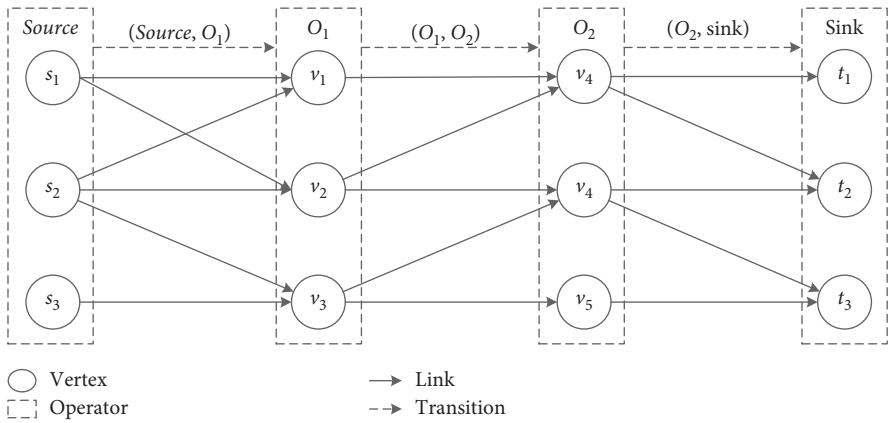

Figure 2: ExecutionGraph of a streaming job.

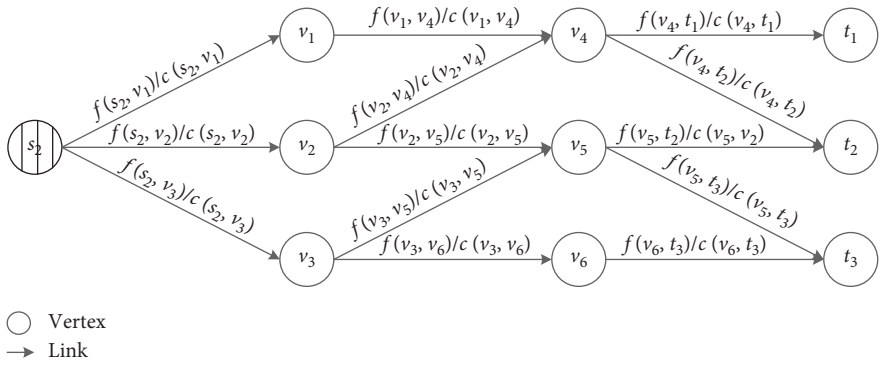

Figure 3: Diagram of the flow network.

instances of different operators and executes different instructions, the capacity on each edge is based on the processing capability of the corresponding vertices. The capacity detection algorithm (Algorithm 1) calculates the capacity and acquires different values for each edge. Meanwhile, the flow on each edge is based on the data sending rates from the upstream vertices. The experimental results show that the capacity and flow are different. Therefore, the flow network illustrates the relationships between the data processing capability and the data transition speed of the topology at that moment. Moreover, the stream of the network f is the sum of the data sending rates in the source, and its value is defined as

$$|f| = \sum_{v_i \in V} f(s, v_i). \tag{1}$$

According to the flow-network model, $c(v_i, v_j)$ is the peak value of the data transition rate, which represents the transition capability of the link and the processing capability of the corresponding vertex. It provides the foundation of the model for the elastic resource allocation algorithm, as presented in Section 4.2, to identify the performance bottleneck of the cluster and to reschedule computing resources. Accurate detection of the capacity of each edge is very important; thus, a capacity detection algorithm is proposed in Section 4.1 to calculate the capacity of each edge, construct the model of the flow network, and lay a foundation for creating the resource-rescheduling plan.

As shown in Figure 4, given the flow network $G = (V, E)$ with stream f, the corresponding improving network of G on f is $G_f = (V_f, E_f)$, which describes the optimization space for the throughput of the cluster. Moreover, $V_f = V$ is the set of vertices and E_f is the set of edges in the topology. For each edge $(v_i, v_j) \in E$ in the improving network, there is an improve space and a reduce space for the edge. The improve space $p(v_i, v_j) = c(v_i, v_j) - f(v_i, v_j)$ represents the space within which the data transition speed of the corresponding edge can be increased. The reduce space $r(v_i, v_j) = f(v_i, v_j)$ represents the space within which the data transition speed of the corresponding edge can be decreased.

Specifically, in the original flow network, the flow $f(v_i, v_j)$ represents the data transition speed on the corresponding edge, and $c(v_i, v_j)$ is the maximum data transition speed. Accordingly, the data transition speed of the corresponding edge can be no larger than the improve space $p(v_i, v_j)$, while the data transition rate can be no smaller than the reduce space. In conclusion, the improve and reduce spaces represent the ranges within which the data transition rate of the corresponding edge can be increased or reduced, respectively. Therefore, the processing load distribution can be optimized and the data processing efficiency can be improved by making full use of existing computing resources.

Input:
Topology of the job: $T = \{V, E\}$
Output:
Flow network of the job: $G = \{V, E\}$
Begin
Initialize the flow network from the topology.
foreach $(v_i, v_j) \in G.E$ do
  Calculate the initial capacity of the edge according to equation (10).
end foreach
if $S.f > 0$ then
/*Feedback regulation executes when a processing load exists*/
  foreach $v_j \in G.V$ do
    if avg($latency$) $> \theta$ and $f(v_i, v_j) \leq c(v_i, v_j)$ then
      $c(v_i, v_j) \longleftarrow c(v_i, v_j) - \eta$ /*Reduce the capacity*/
    else if avg($latency$) $<< \theta$ && $f(v_i, v_j) \approx c(v_i, v_j)$ then
      $c(v_i, v_j) \longleftarrow c(v_i, v_j) + \eta$ /*Enlarge the capacity*/
    end if
  end foreach
end if
return G
End

Algorithm 1: Capacity detection algorithm.

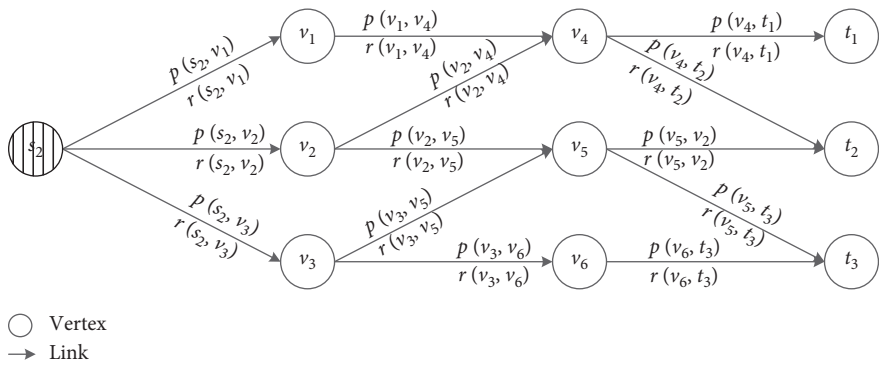

Figure 4: Diagram of the improving network and optimization path.

Meanwhile, as shown in Figure 4, one of the optimization paths P: $s \longrightarrow v_i \longrightarrow t_i$ is an acyclic path from the source to one of the sinks. Therefore, the promotion capacity of the optimization path P is the lowest promotion capacity of an edge on P:

$$\left|f_p\right| = c_p(P) = \min\left\{c_p\left(v_i, v_j\right) \mid \left(v_i, v_j\right) \in P\right\}, \quad (2)$$

where $c_p(v_i, v_j)$ is the promotion capacity of the corresponding edge in the improving network. Therefore, by promoting the flow of the original network along the optimization path, the promotion stream is acquired and denoted as $f' = f \uparrow f_p$, while the value of the promoted flow on each edge is

$$f'\left(v_i, v_j\right) = \left(f \uparrow f_p\right)\left(v_i, v_j\right) = \begin{cases} f\left(v_i, v_j\right) + \left|f_p\right|, & \left(v_i, v_j\right) \in P, \\ f\left(v_i, v_j\right) - \left|f_p\right|, & \left(v_j, v_i\right) \in P, \\ f\left(v_i, v_j\right), & \text{others}, \end{cases} \quad (3)$$

where $f(v_i, v_j)$ is the flow on the edges of the original network.

The promotion capacity in the same direction as the original network is the difference between the capacity and the flow of each edge, which represents the space for improving the capacity of the original network. By contrast, the promotion capacity in the opposite direction of the original network represents the space for reducing the transition load of the corresponding edges. Therefore, we can optimize the load distribution among nodes according to the promotion capacity of every edge. Meanwhile, the optimization path with the minimum promotion capacity of the edges on the path provides a scheme for promoting the throughput from the source to one of the sinks.

Supposed that $G = (V, E)$ is a flow network with flow f and the corresponding improving network $G_f = (V_f, E_f)$ with optimization path P. Thus, the promoted flow $f \uparrow f_p$ is also one of the streams of the original network, and the promoted value is $|f'| = |f \uparrow f_p| = |f| + |f_p| > |f|$, which means that the promoted flow is always larger than the

original flow. Therefore, since we can find a greater flow of the topology by the optimization path, a scheme to promote the value of the stream in the original flow network and to dispatch data more effectively is achieved. Thus, the latency to dispatch stored data in the direction of the optimization path should be reduced when data accumulation is formulated at the source. Therefore, the core idea of Flink-ER is to find the optimization path, dispatch accumulated data, and identify the performance bottleneck.

2.3. Model of Resource Allocation. The mathematical relationships between the processing capability of nodes and the input load in practice are quantified by proposing a flow-network model. However, when the cluster performance declines due to a lack of computing resources, optimization of the data distribution strategy is unnecessary; thus, it is crucial to develop the resource allocation model as a basis of an elastic resource reallocation algorithm.

Moreover, given the flow network $G = (V, E)$, in which $s \in S$ is the source and $t_i \in T$ are sinks, partition $D = (X, Y)$ divides the set of vertices in the flow network into two parts, namely, X and $Y = V - X$, where $s \in X$ and $v_i \in Y$ are strictly followed. The divided sets are satisfied with the equations $X \cap Y = \varnothing$ and $X \cup Y = V$. Moreover, for vertices $v_i, v_j \in O_i$, $v_i, v_j \in X$, or $v_i, v_j \in Y$ must be workable, which means that instances of the same operator are always in the same set of any partition. Therefore, the corresponding capacity of the partition $D = (X, Y)$ is the sum of the capacity values of edges across the partition:

$$c(D) = c(X, Y) = \sum_{v_i \in X} \sum_{v_j \in Y} c\left(v_i, v_j\right). \tag{4}$$

Furthermore, the flow of the partition is the sum of the values of edges across the partition:

$$f(D) = f(X, Y) = \sum_{v_i \in X} \sum_{v_j \in Y} f\left(v_i, v_j\right) - \sum_{v_i \in X} \sum_{v_j \in Y} f\left(v_j, v_i\right). \tag{5}$$

In particular, the partition with the minimum capacity is the minimum partition of the flow network.

According to the description above, a partition divides the flow network into two parts, for which the source and sinks are always on different sides and the instances of the same operators are always in the same set. Therefore, each partition of the network represents a potential performance bottleneck, where the processing load is beyond the transition capability across the partition. Consequently, partitioning is the most effective way to identify the bottleneck and supply the computing resources necessary to break it accurately.

Let $G = (V, E)$ be a flow network and $G_f = (V_f, E_f)$ be the improving network on stream f. If it does not exit any optimization path in G_f, then there is not any room to improve the throughput; all the computing nodes are fully occupied with data processing tasks, and the cluster meets the conditions of the performance bottleneck. There must be an operator whose instances are not capable of processing incoming data on time due to the lack of computing resources, and f must be the stream with the maximum flow of the network. As a result, there is at least one partition $D_{\min} = (X, Y)$ as the minimum partition of G that makes $|f| = f(X, Y) = c(X, Y)$ true [10], and this $D_{\min}$ indicates that the operator lacks computing resources. In other words, the minimum partition represents the actual performance bottleneck of the cluster that is not capable of processing arrival data on time. Therefore, Flink-ER identifies the performance bottleneck through the minimum partition and supplies computing resources to address it.

2.4. Model for Data Redistribution. To address the bottleneck, the job should be rescheduled, and tasks should be migrated among nodes to provide enough computing resources. However, the intermediate results of computing are retained in every node as the stateful data in stateful stream processing, which is the major challenge to propose an elastic resource-scheduling strategy.

During the execution of the resource-rescheduling plan, stateful data of each node are uploaded to the HDFS by the checkpoint and redistributed by the restored mechanism. However, in the Original Flink, stateful data are organized in *bulk*, which is not appropriate for data redistribution with the low granularity of data management. For example, suppose one of the processed data points of operator O_n is $tuple_i = (key, value)$; its corresponding *bulk* in the Original Flink would be

$$\text{bulk}\left(\text{tuple}_i\right) = \text{hash}\left(\text{key}\right), \tag{6}$$

which means that the *key* to tuples is mapped by the hash function, and tuples with the same hash value are mapped to the same *bulk*. However, the bulk-management strategy leads to computing nodes frequently accessing the HDFS, which directly results in a performance decline of the data transition at the checkpoint. However, in the optimized stateful data management strategy, suppose one of the processed data points for operator O_n is $tuple_i = (key, value)$ and that k buckets exist in the operator. The $tuple_j$ is first mapped to the corresponding bulk according to equation (6). Then, the tuple is mapped to the bucket as

$$\text{bucket}\left(\text{tuple}_i\right) = \text{bulk}\left(\text{tuple}_i\right)\%k. \tag{7}$$

The nodes that respond to process the tuple are

$$v_i = \frac{bucket\left(tupe_i\right) \times \left|P\left(O_n\right)\right|}{k_n}, \tag{8}$$

where $k_n \geq |O_n|$ and v_i is one of the instances of operator O_n. Thus, the mapping from the data to the computing nodes is defined by equations (7) and (8), while the bucket is the fundamental unit in stateful data management.

With the improved data redistribution model, the scattered stateful data are integrated as several *buckets* in nodes, which significantly improves the efficiency of the checkpoint and data restore process. Consequently, the data redistribution algorithm is designed based on the model and used to execute the elastic resource-rescheduling plan by

adding computing resources and effectively migrating stateful data among nodes.

3. Elastic Resource Reallocation Strategy

Based on the models developed in Section 3, the three core algorithms of Flink-ER are presented in this section. First, the capacity detection algorithm is designed to calculate the capacity of each edge in the flow network by feedback regulation. Second, the elastic resource reallocation algorithm is proposed based on the resource allocation model to optimize the data dispatching strategy, identify the performance bottleneck, and create the resource-rescheduling plan. Finally, the data redistribution algorithm is presented to invoke the resource-rescheduling plan by effectively migrating data to the nodes. Furthermore, the deployment of the strategy and the structure of Flink-ER are presented.

3.1. Capacity Detection Algorithm for Flow Network. According to the definition of the flow network, the capacity value $c(v_i, v_j)$ represents the processing and transition capability of corresponding vertices and edges, which is crucial to evaluate the performance of a cluster. As a result, an accurate value of the capacity must be acquired to lay the foundation for efficient elastic resource scheduling. Practical experience and the experimental results show that the CPU, RAM, and network bandwidth impact the performance of computing nodes, with the network bandwidth being the determinant factor because the transition overhead contributes the most latency to data processing. Therefore, the algorithm takes network performance as a basis for defining the initial value of the capacity, and it takes the remaining factors into consideration by feedback regulation.

Generally, a cluster is connected by a hundred-megabyte network, in which the highest data transition rate is $R_{v_i}^B = 100\,\text{Mb/s} \approx 12.5\,\text{MB/s}$. Suppose that, in a pure network environment where all the irrelevant processes are turned off in every node, i.e., no unnecessary transition overhead exists between nodes, the available resources for data transition are

$$\begin{aligned} N_{v_i}(\text{data}) = R_{v_i}^B - N_{v_i}(\text{system}) - N_{v_i}(\text{heartbeat}) \\ - N_{v_i}(\text{CK}) - N_{v_i}(\text{others}), \end{aligned} \tag{9}$$

where $N_{v_i}(\text{system})$ is the transition overhead contributed by the operating system, $N_{v_i}(\text{heartbeat})$ is that contributed by the heartbeat communication for Flink processes, $N_{v_i}(\text{CK})$ is that contributed by the checkpoint for the data transition between TaskManager and HDFS or Zookeeper, and $N_{v_i}(\text{others})$ is the dynamic unpredictable data transition overhead in the cluster, which is extremely small. Therefore, the remaining resources $N_{v_i}(\text{data})$ are available for actual data transition, and the unit of measurement is MB/s. Consequently, for a certain streaming jobs, the capacity of edge (v_i, v_j) is

$$c(v_i, v_j) = \frac{N_{v_j}(\text{data})}{|E_{ij}| \times \sum_{k=0}^{n-1} \text{size}(\text{tuple}.f_k)}, \tag{10}$$

where $|E_{ij}|$ is the number of input links of vertex v_j, *size* $(tuple.f_k)$ is the size of the data to be processed, and the unit of measurement is byte.

Based on the analysis above, the specific process of the capacity detection algorithm is as follows.

First, as shown in Algorithm 1, the flow network of the topology, which consists of all vertices and edges in the DAG topology, is initialized. Second, the initial capacity $c(v_i, v_j)$ of every edge is calculated according to equation (10). However, a deviation occurs in the representation of the transition capability of the corresponding edges. Therefore, feedback regulation is executed repeatedly on every edge. If the average processing latency exceeds the user-defined threshold, i.e., the capacity is beyond the exact capability of the vertex, then the capacity is reduced. In contrast, if the average processing latency is far below the user-defined threshold, i.e., resources still exist in the computing nodes, then the capacity is increased. Finally, the capacity of each edge is converted to a value within an ideal range by initial value calculation and continuous feedback regulation.

Therefore, the DAG topology of the job is transformed into a flow network by detecting the capacity of every edge, which provides a modeling foundation for the elastic resource reallocation algorithm to identify the performance bottleneck and create the resource-rescheduling plan. Moreover, η is an important parameter representing the step size of feedback regulation; we will discuss how to acquire the value of η in Section 4.4.

3.2. Elastic Resource Reallocation Algorithm. When data accumulation occurs in the source, the elastic resource reallocation algorithm is proposed to dispatch the accumulated data, identify the performance bottleneck, and execute the resource-rescheduling plan. According to Section 2.3, as the actual bottleneck of the cluster, the minimum partition limits the cluster performance, and the parallelism of the operator in set Y should be increased. However, this bottleneck might not be the only one. Suppose that the partition $D = (X, Y)$; every partition that meets the criterion

$$f(X, Y) \geq \lambda c(X, Y), \tag{11}$$

might be identified as the bottleneck of the cluster, where $0.85 \leq \lambda \leq 1$. Thus, once the flow of a partition reaches 85% of its capacity, the partition can become the potential bottleneck of the cluster. Therefore, the parallelism of the first operator in set Y should be increased by providing more computing resources.

According to the model of resource allocation and the analysis given above, the elastic resource reallocation algorithm is as follows.

First, as shown in Algorithm 2, the improving network is formulated according to the definition, and the optimization path is sought based on the flow network. Second, the accumulated data are dispatched along the optimization path when there is a path in the improving network to improve the throughput of the cluster. In contrast, when no optimization path exists in the improving network, which means that the unprocessed data arrival rate exceeds the highest

throughput of the cluster, then the bottleneck of the cluster is identified by the minimum partition, and extra resources are supplied to improve the processing capability and throughput of the cluster. Finally, the data redistribution algorithm is used to migrate stateful data and reschedule tasks.

In summary, resource utilization is maximized, and the performance bottleneck of the cluster is identified by using an elastic resource-rescheduling algorithm proposed based on the resource allocation model. Meanwhile, the resource and task scheduling plan is formulated by determining which operators should be scaled. Therefore, we propose the data redistribution algorithm for migrating stateful data and carrying out the task scheduling plan.

3.3. Data Redistribution Algorithm. As described above, it is important to migrate the stateful data when carrying out the resource and task scheduling strategy. However, traditional stateful data management is not appropriate for migration, which involves too much communication with too much scattering and low data management granularity. To address this problem, the data redistribution algorithm is proposed based on the data redistribution model to reduce the communication overhead during the stateful data and task migration.

As shown in Figure 5, suppose that the parallelism of an operator is increased from 3 to 4. The stateful data managed by 3 instances are first pushed to the HDFS in *bulk*. Then, the mapping from the data to the corresponding instances is modified according to the new parallelism. Finally, the stateful data are pulled from the HDFS according to the new mapping. Based on the description, the detailed process of the data redistribution algorithm is as follows.

Algorithm 3 shows that, for each operator to be scaled, the stateful data managed by instances are first pushed to the HDFS, and the handler, which records the location and size of the data, is stored in Zookeeper. Second, the supplementary nodes are requested and added to the operator, while the stateful data mapping is revised according to equations (7) and (8), as shown in Figure 5. Finally, each instance requires the data handler to obtain the location of stateful data and pull the data from the HDFS. Moreover, Algorithm 1 is used to regulate the capacity of each edge based on the new structure of the flow network.

By proposing the data redistribution algorithm, the scattered distribution of the data and frequent HDFS access are resolved. Therefore, the overhead of data migration is reduced, and the efficiency of Flink-ER is improved.

3.4. Discussion of Selected Parameter Values. In the capacity detection algorithm (Algorithm 1), parameter η represents the step size of capacity regulation. A value of η that is too high will lead to capacity fluctuation due to overregulation. In contrast, a value of η that is too low will lead to a decrease in the efficiency of the algorithm due to too many rounds of regulation before convergence. Therefore, an accurate and dynamic value of η is the key to achieving efficiency.

Suppose that the latency constraint of the job is l_c, which means the response time of the system should be limited to l_c. However, in practice, the actual average processing latency of the vertex v_i could be monitored through a latency tracking mechanism as

$$l_{v_i} = \frac{\sum_{i=1}^{n} (\text{tuple}_i.\text{latency})}{n}. \tag{12}$$

Moreover, according to the definition of capacity in Section 3, the capacity is the data transition rate per second. In other words, for every 1000 ms, the tuple transition rate of the vertex v_i would be

$$c(v_i, v_j) = \frac{1000}{l_{v_i}} = \frac{1000n}{\sum_{i=1}^{n} (\text{tuple}_i.\text{latency})}. \tag{13}$$

According to the latency constraint l_c, the deviation between the current capacity and the value, which is the feedback regulation step size η, would be

$$\eta = \left| \frac{1000n}{\sum_{i=1}^{n} (\text{tuple}_i.\text{latency})} - \frac{1000}{l_c} \right|. \tag{14}$$

In conclusion, parameter η can be calculated dynamically by continuously tracking the latency of the system response time and calculating the deviation between the current and idea value. Therefore, we can obtain the maximum possible capacity if the latency constraint is met.

3.5. Implementation and Deployment of the Strategy. To implement Flink-ER, the basic architecture of the platform should be improved. The architecture of Flink-ER is developed based on the Original Apache Flink.

As shown in Figure 6, the major improvement of Flink-ER compared to the Original Flink is the addition of the following four components or threads:

(1) Zookeeper: Zookeeper saves the architecture of the flow network and the metadata of the stateful data at the checkpoint and coordinates among TaskManagers.

(2) Capacity detector: this detector, which is a traditional Java thread using Algorithm 2, is responsible for detecting the capacity of the flow network and developing the flow-network model.

(3) Rescheduler: the Rescheduler is a traditional Java thread invoking Algorithm 1 that is responsible for optimally dispatching the accumulated data and creating the resource-rescheduling plan.

(4) Data migrator: this component, which is a traditional Java thread executed by each TaskManager and uses Algorithm 3, is responsible for pulling the stateful data from the HDFS according to the metadata stored in Zookeeper.

Regarding the data transition, since the controlling information is transformed by Akka [11] between TaskManager and JobManager, TaskManager sends the data handler to JobManager by the Akka actor named AcknowledgeCheckpoint. To implement Flink-ER, the actor named

Input:
Flow network: $G = \{V, E\}$
Current flow of the network: f
Accumulation of every partition in source: $lags[]$
Output:
Rescaled flow network: G'
Begin
Build the improving network from the flow network.
$P \leftarrow \text{BFS}(G_f, G_f.s, G_f.T)$/* Search for the optimization path*/
while $\text{Max}(lags[]) \geq \varphi$ AND $P\ != \varnothing$ do
  $|f_p| \leftarrow \min\{c_f(v_i, v_j)|(v_i, v_j) \in P\}$
  $G \leftarrow \text{enlargeFlow}(G.E, P, |f_p|)$
  $P \leftarrow \text{BFS}(G_f, G_f.s, G_f.T)$
end while
if $\text{Max}(lags[]) \geq \varphi$ then
  Enlarge the parallelism of each operator that meets the criterion of equation (11)
end if
$G' = \text{taskMigration}(operator[])$
return G'
End

Algorithm 2: Elastic resource reallocation algorithm.

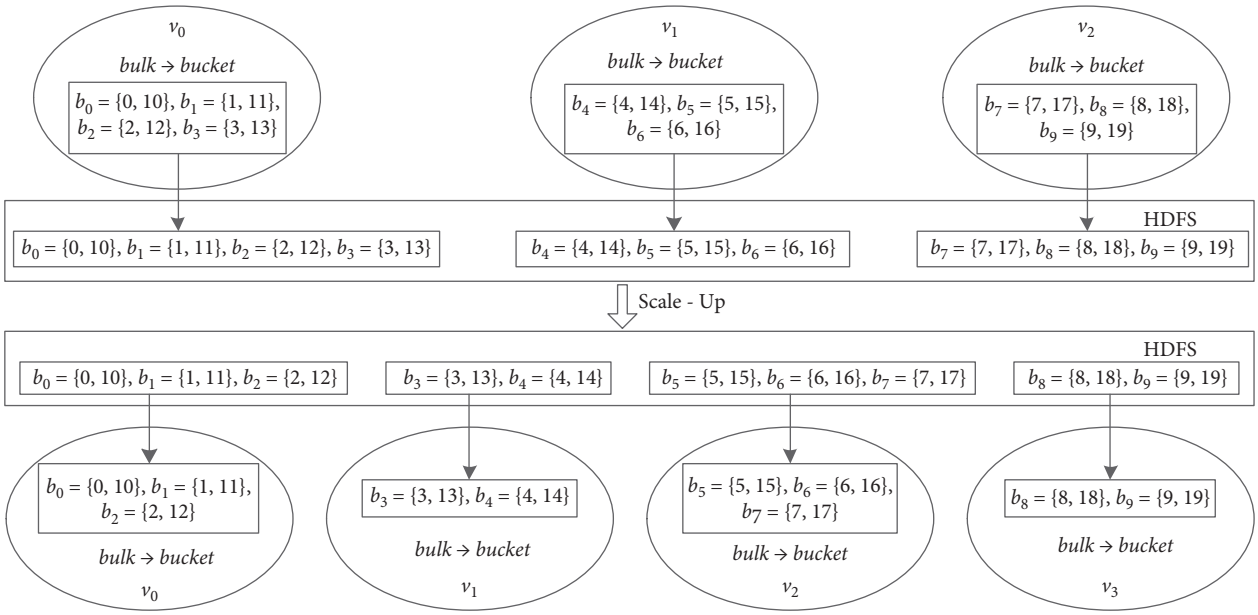

Figure 5: Diagram of data migration.

ReportNetwork is developed for the TaskManager to report the current capacity and flow to the JobManager. Moreover, the Rescheduler writes the scheduling plan and other metadata to Zookeeper, while the data migrator reads from it. Therefore, Zookeeper is a crucial coordinator for realizing Flink-ER.

4. Experimental Results and Analysis

To verify the performance improvement of Flink-ER on various kinds of jobs, a practical experiment was set up, and different benchmarks were executed on different platforms. For comparison with related state-of-the-art methods, we realized the core ideology of Sra-Stream [12] on Flink and ran the same benchmarks with Flink-ER for the same experimental setup. The experimental results verify the superiority and evaluate the overhead of Flink-ER compared with those of the Original Flink and Sra-Stream.

4.1. Experimental Setup. To simulate the practical use case, the cluster consisted of 21 personal computers, and the deployment architecture is as shown in Figure 6, in which one node was deployed as JobManager, 6 nodes were

Input:
Operator to be rescaled: *operator*[]
Resource pool: *pool*
Output:
Rescaled flow network: G'
Begin
foreach $O \in operator[]$ do
  foreach $v \in O$ do
    Send the stateful data of vertex v to the HDFS and the data handler to Zookeeper.
    end foreach
    $v \leftarrow pool$.getNode()
    O.add(v)/*Enlarge the parallelism of the Operator*/
    bucket[]←JobManager.remapping(*bucket*[],|O|)
    /*Remapping stateful data as shown in Figure 5*/
    foreach $v_i \in O$ do
      Require stateful data from Zookeeper and restore the corresponding stateful data from the HDFS
    end foreach
end foreach
flowNetwork_selfLearning(G')
return G'
End

Algorithm 3: Data redistribution algorithm.

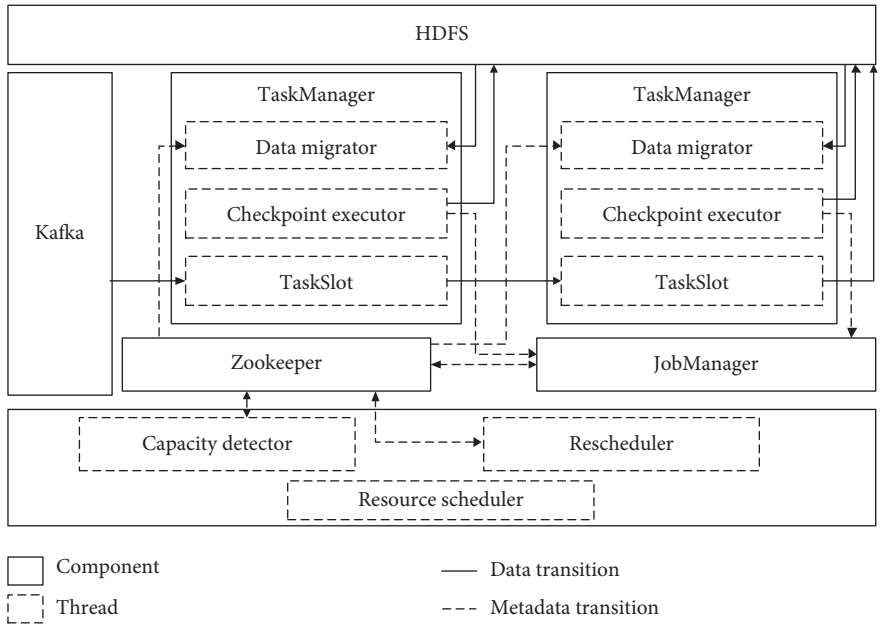

Figure 6: Architecture of Flink-ER.

deployed as TaskManager, and 4 nodes were deployed in the resource pool as the standby TaskManager for elastic supplementation. Furthermore, 3 nodes were deployed as Hadoop, 3 nodes were deployed as Zookeeper, and 1 node was deployed as the Resource Scheduler. The hardware and software setups are shown in Tables 1 and 2, respectively.

Sra-Stream, which is an elastic scheduling strategy for stateful stream processing, is the most closely related contribution to that in this paper. We realized its core ideology and plugged it into Flink as the resource and task scheduling strategy for comparison with Flink-ER. The experimental results and analysis are presented below.

4.2. Experimental Results and Analysis. To compare the performance of Flink-ER with those of Sra-Stream and the Original Flink under different practical application scenarios, four benchmarks, i.e., WordCount, TwitterSentiment, IncrementalLearning, and Streaming-Benchmarks, which come from the source code of Flink (v1.6.0) [13] and a contribution from Intel on GitHub [14], were executed on the three platforms, as they are representative Flink use cases.

In the experiment, the processing latency and data accumulation were gathered through latency tracking in the metrics by periodically sending an HTTP request to the

Table 1: Hardware setup.

Item	Value
CPU	Intel (R) Core (TM) i7-4790 CPU @ 3.60 GHz
RAM	4 GB DDR3 1600 MHz
Disk	1 TB, 7200 RPM
Network	100 Mb/s

Table 2: Software setup.

Item	Value
OS	CentOS 6.5
JDK	jdk1.8.0–181-Linux-x64
Apache Flink	1.6.0
Apache Hadoop	2.7.4
Apache Kafka	2.10–0.8.2.0
Apache Zookeeper	3.4.10
Redis	5.0.2

REST API, which is the web UI of Flink, with a 60 s interval. These two indicators directly reflect the performance of the cluster, and the experimental results are as follows.

As a CPU-intensive job, WordCount is used to count the frequency of each word that appears in an article. We set the input data as a continuous data flow. The experimental results are presented in Figure 7. The processing latency increased as the data accumulated, which verifies that data accumulation leads to processing latency. Furthermore, without any elastic scheduling strategy, the latency of the Original Flink continued to grow, exceeding 900 ms, which is unacceptable in practice. In contrast, the latency of Flink-ER increased first and then decreased to an acceptable range because Algorithm 2 dispatched accumulated data while supplementing computing resources for 5-6 min. As Figures 7(c) and 7(d) show, during the elastic resource-scheduling process (~300–480 s), both strategies reduced the processing latency by resource and task rescheduling, but Sra-Stream took longer to schedule resources and was involved in the performance fluctuation. In contrast, Flink-ER took approximately 60 s to schedule tasks and effectively reduced the latency because the data redistribution algorithm reduced the communication overhead during the stateful data migration. After that, both strategies limited the latency within an acceptable range.

As a memory-intensive job, TwitterSentiment is a standard benchmark used to analyze user sentiment through tweets, reflecting the data accumulation by memory usage. Therefore, the continuous data flow of user tweets on Twitter in JSON format is set as the input data source. Since each vertex retains complicated stateful data during the execution of TwitterSentiment, the overhead of data migration by Algorithm 3 is verified. Therefore, Figure 8(a) illustrates that, with the increase in input load, the throughput of the cluster decreases due to a lack of computing resources, while both elastic resource scheduling strategies promoted throughput by optimizing the load distribution and supplementing resources, which contributed a significant improvement. In addition, Figures 8(c) and 8(d) illustrate that memory utilization is reduced in both strategies. However, Sra-Stream required more time for stateful data migration and exhibited scheduling fluctuation. Flink-ER required less time for scheduling but had higher memory utilization for data accumulation during the stateful data restore process from the HDFS. In general, the memory overhead for task scheduling in Flink-ER is acceptable in practice.

As a CPU-intensive job, IncrementalLearning is a complicated iterative computation on a large-scale matrix in which the CPU is the typical scarce resource during execution. Figure 9(a) illustrates that the bottleneck of the Original Flink limited less than 80000 tuples/s, while both strategies provided significant improvement with acceptable overhead. Figures 9(b)–9(d) show the CPU utilization of one TaskManager in the cluster. With the increase in processing load, the CPU utilization approached 100%, which means that the lack of CPU resources limited the throughput and performance of the cluster. In contrast, both scheduling strategies decreased CPU utilization by supplying more computing resources, enlarging the parallelism of CPU-intensive operators (iterative computation in this job) and distributing the processing load. However, the imperfections of the two strategies are different. Sra-Stream showed fluctuations in the system response latency and resource utilization, while Flink-ER had a short processing break (approximately 3 s) for stateful data migration; the job was then restored to normal execution with better performance. Future research will focus on shortening or reducing breaks. Moreover, since all the systems executed the same learning benchmark during the experiment, their learning accuracies are exactly the same, but their executing efficiencies and performance results are different.

Streaming-Benchmarks is used for advertisement analysis and was contributed by Yahoo on GitHub [15]; it is a typical use case in data analysis for mobile devices. As Figure 10 illustrates, because of the high input load, complicated computation, and large-scale stateful data, the execution of the benchmark occupied both the CPU and memory resources of the TaskManager nodes. However, the Original Flink is not capable of processing data regarding the arrival rate on time, which results in data accumulation and latency increase. Therefore, the cluster executed both resource scheduling strategies over 600 s and 780 s while improving the performance by supplementing resources. The decrease in data accumulation and processing latency verified that resource rescheduling effectively improves the performance of a cluster by supplementing the resource and rescheduling tasks. In addition, before rescheduling, Flink-ER provided a higher throughput by optimally dispatching accumulated data. During rescheduling, Flink-ER reduced the overhead through efficient stateful data management and migration. After rescheduling, the cluster was likely to stabilize faster. Therefore, compared with Original Flink and Sra-Stream, Flink-ER is more suitable for practical application scenarios with complicated stateful data and is more applicable to data analysis in mobile information systems with complicated computing.

In conclusion, by dynamically rescheduling resources and efficiently migrating stateful data, Flink-ER promotes

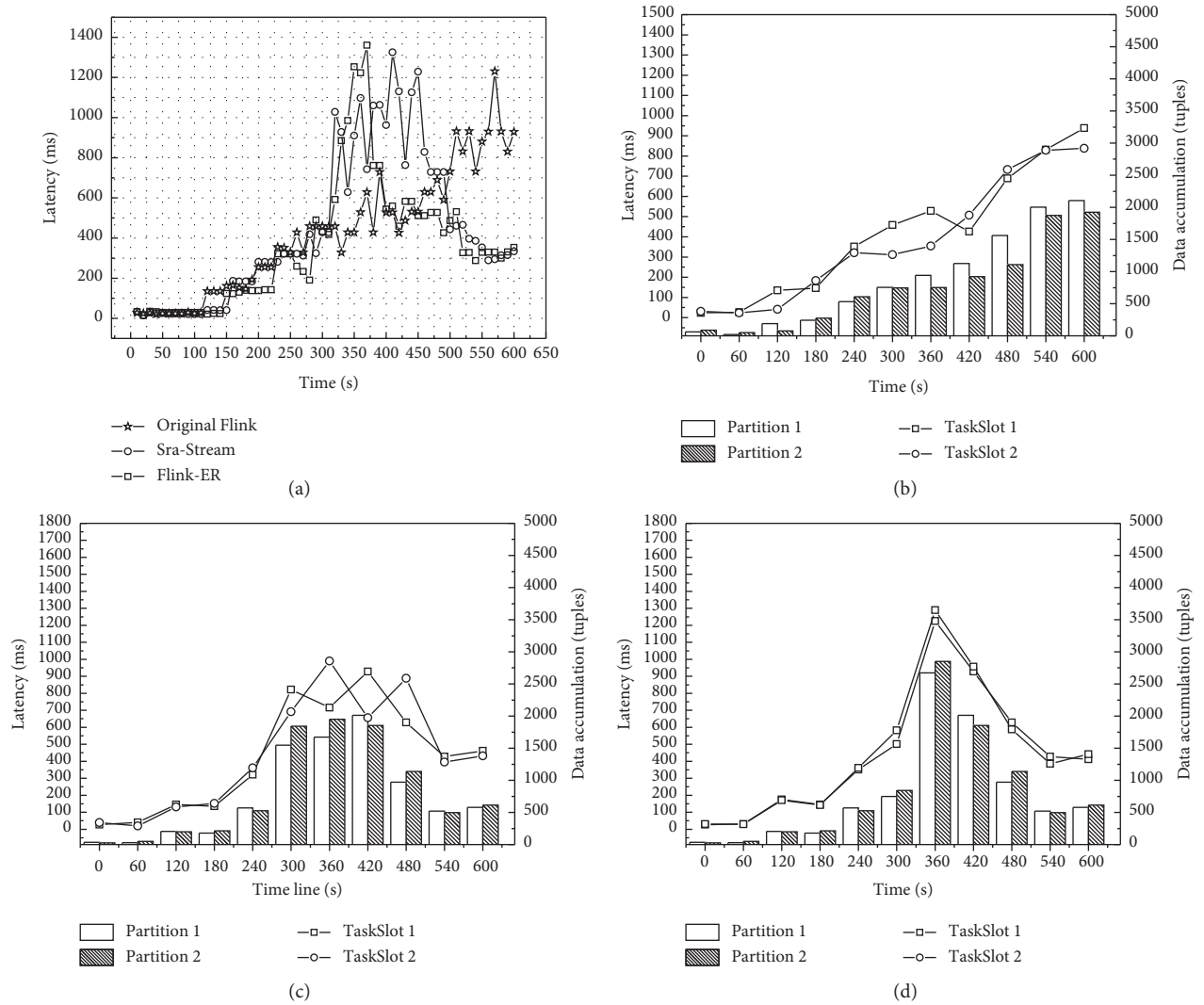

Figure 7: Performance comparison on WordCount: (a) latency comparison; (b) Original Flink; (c) Sra-Stream; (d) Flink-ER.

cluster throughput via latency constraints. Compared with Original Flink, Flink-ER improves the job execution efficiency by proposing an elastic resource-rescheduling strategy. Compared with Sra-Stream, Flink-ER reduces the overhead of rescheduling by proposing an efficient stateful data management and migrating strategy. The specific conclusions of the experiment are presented in Table 3.

4.3. Overhead Evaluation. Since the migration of stateful data involves transition overhead that might affect the performance of the cluster, the data transmission during the checkpoint and restore process was evaluated. StreamingBenchmarks and TwitterSentiment are streaming jobs with complicated stateful data; hence, they were used to evaluate the transition overhead, while the network transition monitoring operations for Linux, i.e., iftop and tcpdump, were used to monitor transmission among the nodes. The captured data are presented in Figure 11.

Figure 11 clearly shows that the checkpoint is involved in a small periodic transition because of the execution of the incremental checkpoint. Regarding the restoration process, the quantity of restored data was the same in both platforms, but Flink-ER took less time to restore data because stateful data are stored in the HDFS centralized in the data management system by a bucket. In addition, the frequency of HDFS access is significantly reduced by the data redistribution algorithm. In conclusion, Flink-ER efficiently reduced the overhead of data transition compared to original Flink.

4.4. Analysis of the Task Break Out. According to the experimental results, since existing work on elastic resource scheduling suffers from the problem of instantaneous break out, Flink-ER effectively shortens the break by about half due to the load migration process. The execution of the data processing thread will modify the stateful data. However, it is important that the stateful data remain consistent during migration among nodes, as consistency is crucial for ensuring the accuracy. Therefore, the data processing thread in each node should be paused during stateful data and task

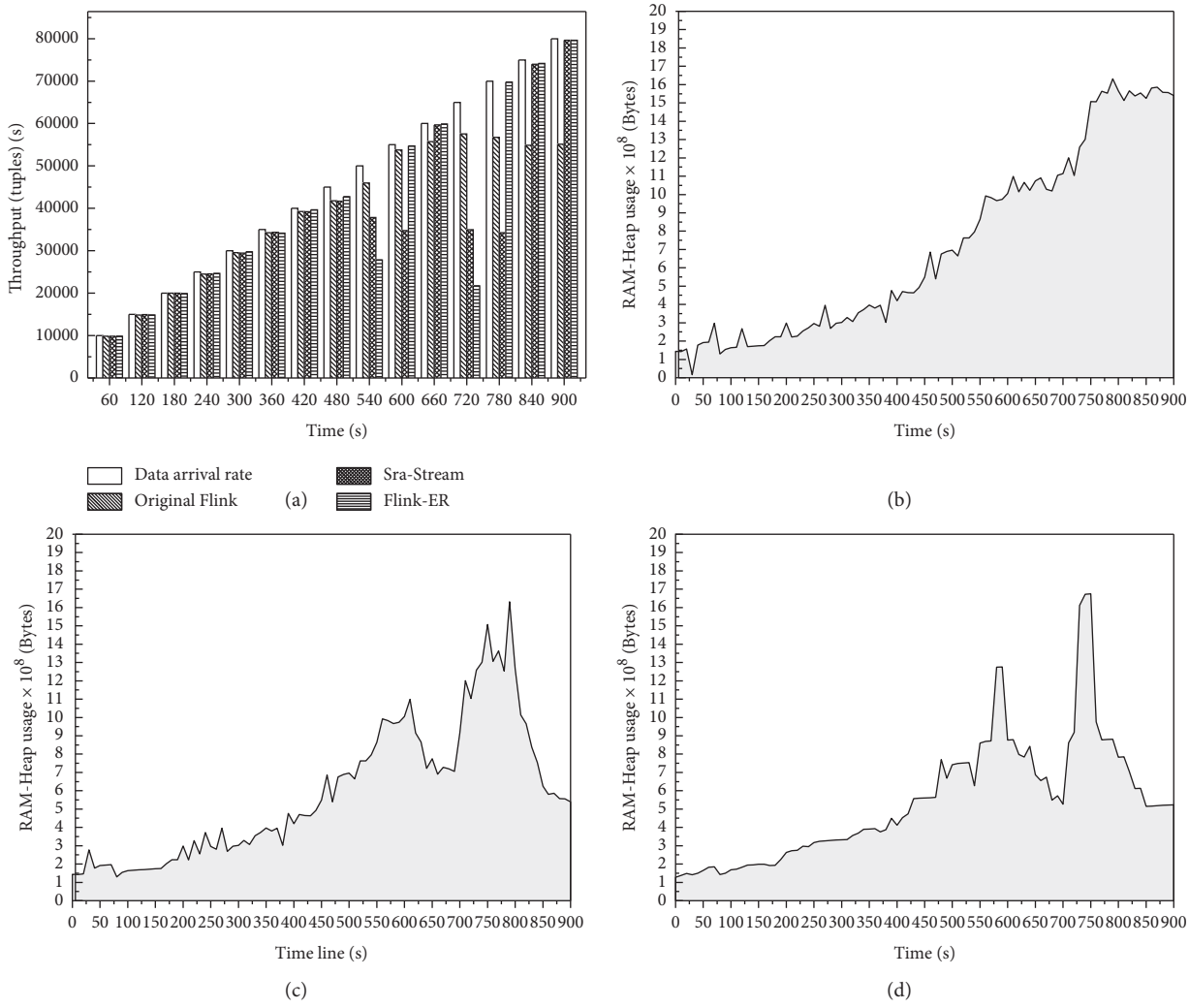

Figure 8: Performance comparison on TwitterSentiment: (a) throughput comparison; (b) heap memory usage on Original Flink; (c) heap memory usage on Sra-Stream; (d) heap memory usage on Flink-ER.

migration to maintain the consistency of stateful data. However, the task break out still affects the performance of the continuous mobile stream data processing within an acceptable range.

Therefore, completely eliminating the break out remains a goal for future work. To realize online elastic resource scheduling without any break out, the stateful data of each node should be migrated when the data processing thread is running while changes in stateful data should be merged. By considering the characteristics of the checkpoint mechanism in Flink, future work will focus on implementing both task scheduling and data processing and merging stateful data. By proposing an online stateful data and task migration algorithm, the problem of task break out will be resolved, and the performance of elastic resource scheduling will be improved.

5. Related Work

To address the performance decline DSPSs face with fluctuating data streams with different arrival rates, great efforts have been made by researchers to improve the performances of various stream processing engines. Three main approaches are used to address the problem: (1) improve the performance of clusters by optimizing the task scheduling strategy; (2) improve the job execution efficiency by reducing the communication overhead among computing nodes; and (3) improve the scalability of clusters by proposing an elastic resource-rescheduling strategy. The strengths and weaknesses of these methodologies as well as representative contributions are summarized below.

First, optimizing the task scheduling strategy is the most representative traditional technique for improving the performance of DSPSs while guaranteeing the latency constraint. Sun et al. [16, 17] propose different task scheduling strategies on a directed acyclic graph (DAG) of the topology, which achieve outstanding performance in terms of reducing both processing latency and energy consumption. Additionally, previous studies [18–21] formulate a variety of task-scheduling methods by analyzing various characteristics of the topology. In terms of latency,

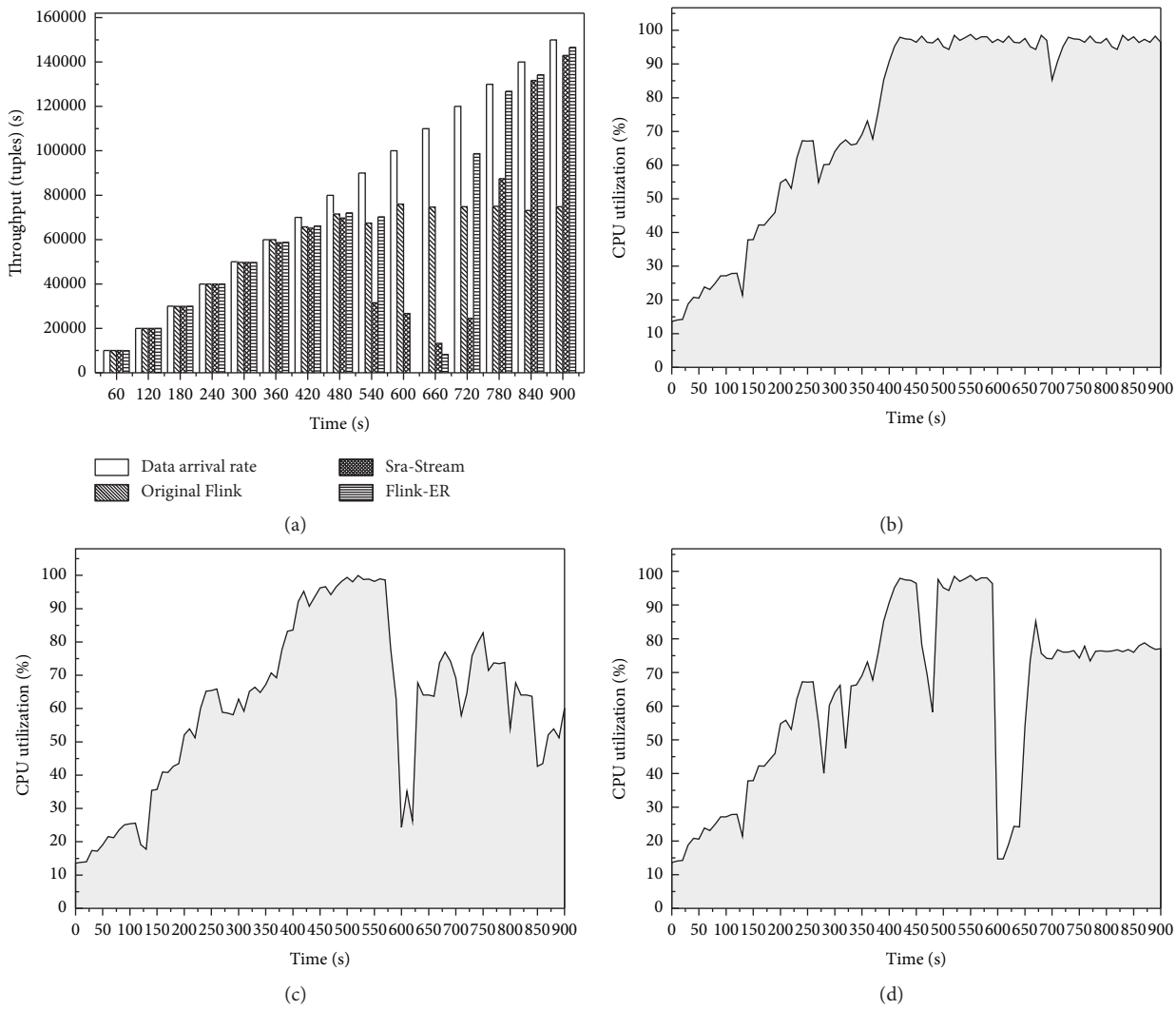

Figure 9: Performance comparison of IncrementalLearning: (a) throughput comparison; (b) CPU utilization on Original Flink; (c) CPU utilization on Sra-Stream; (d) heap memory usage on Flink-ER.

throughput, and resource utilization, they create a task rescheduling plan and improve the topology structure of the job, which is an effective methodology for improving the performance and efficiency of the data stream processing system. However, the task scheduling strategy fails to identify the performance bottleneck or resolve the performance decline when the data arrival rate fluctuates over time. Second, reducing the communication overhead is a popular task scheduling strategy in which tasks are scheduled according to communication performance metrics. Eskandari et al. [22] implement T3-Scheduler for Storm, which can efficiently identify the tasks that communicate with each other and assign them to the same node. Silva et al. [23] propose a methodology for monitoring communication metrics on the edges of DAGs as the training set of reinforcement learning to reconfigure the topology. Moreover, previous studies [24–26] propose different task scheduling strategies for minimizing communication overhead and reducing processing latency. Since data transition contributes the most overhead to the cluster, they reschedule tasks with the most data transitions to the same computing node to reduce communication overhead across nodes, which is an effective method for improving the efficiency of data transition and processing. However, the communication overhead may change drastically with changes in the topology structure and processing load. Therefore, scheduling strategies based on communication metrics are usually not able to respond on time.

In comparison with the two methodologies above, the elastic resource rescheduling strategy is an effective way to overcome the performance bottleneck when handing a continuously fluctuating data stream [27]. Jonathan et al. [28] discuss the trade-off between the reconfiguration of query executions and the adaptation of existing reconfiguration techniques. They find that the best adaptation technique depends on the network conditions, type of query, and optimization metrics. Hidalgo et al. [29] propose a methodology for measuring the elasticity and scalability of distributed stream processing systems, which is very useful for benchmarking the performance of DSPSs when the data

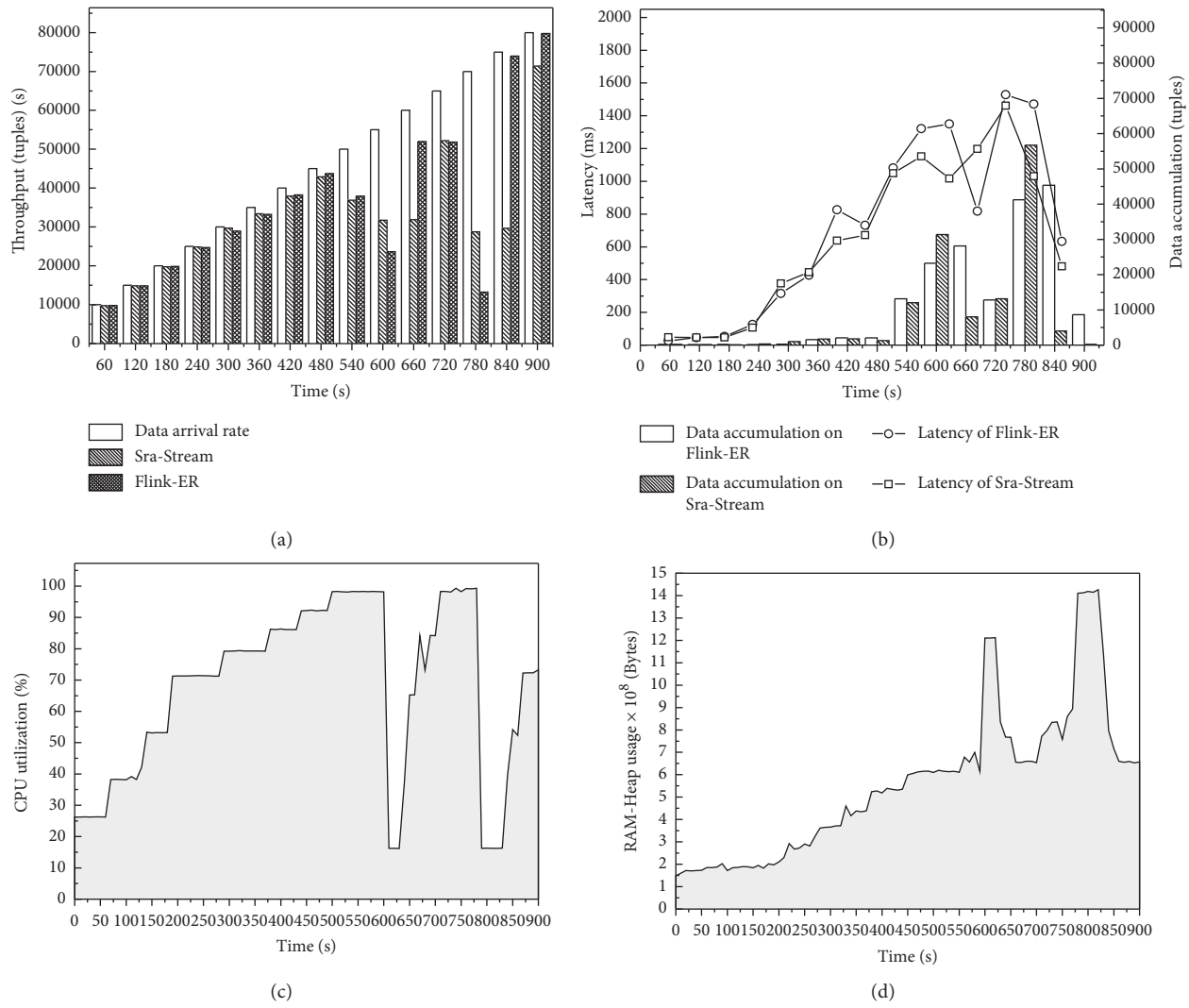

Figure 10: Performance comparison on Streaming-Benchmarks: (a) throughput comparison; (b) latency and data accumulation comparison; (c) CPU utilization on Flink-ER; (d) memory usage on Flink-ER.

arrival rate fluctuates over time. Russo et al. [30, 31] propose different elastic data stream scheduling strategies with a reinforcement learning approach that autorescales the DSPSs by using self-learning algorithms and the performance metrics of the cluster. The integration of machine learning in elastic resource scheduling strategies in DSPSs is promising but still not applicable to industry. Meanwhile, previous studies [32–37] propose a variety of elastic resource dynamic scheduling strategies from the perspectives of self-adaptation, resource constraints, and topology features, which are suitable for different application scenarios. By taking performance metrics and architectural characteristics of specific platforms into consideration, they propose different elastic resource scheduling strategies for disrupting the performance bottleneck. However, some of the above strategies may involve scheduling overhead, and others cannot be applied to Flink.

In addition, Sun et al. [12] propose Sra-Stream, a state-, and runtime-aware scheduling strategy for elastic stream computing. Sra-Stream considers both stateful data migration and the mathematical relationship between the response time and data arrival rate in the task scheduling strategy and achieves good performance on stateful data stream processing. Cardellini et al. [38] realize a decentralized self-adaptation solution for elastic data stream processing that is based on three distributed self-adaptation policies, including a threshold-based approach and two reinforcement learning algorithms. Liu et al. [39] propose an adaptive resource enforcement online scheduling scheme integrated into Storm. The core idea is to isolate the stateful and stateless instance and mitigate the resource contention in a node. However, most existing elastic resource scheduling strategies face the following problems [40]:

(1) To execute elastic resource scheduling, tasks and stateful data should be migrated to computing nodes, which requires significant communication overhead, and the long instantaneous break out may exceed the acceptable range [41]

Table 3: Conclusions of the experiment.

DSPS	Scheduling strategy	Advantage	Disadvantage	Applicable scenario
Flink	Default scheduling strategy	Exactly once processing	Performance bottleneck	Constant and invariable processing load
Sra-stream [12]	First-fit and latency-sensitive runtime-aware-based scheduling	Stateful data scheduling	Scheduling performance fluctuation	Little stateful data
Flink-ER	Flow-network-based dynamic scheduling	Accurate breaking of bottleneck with lower overhead	Instantaneous break out	Large-scale stateful data

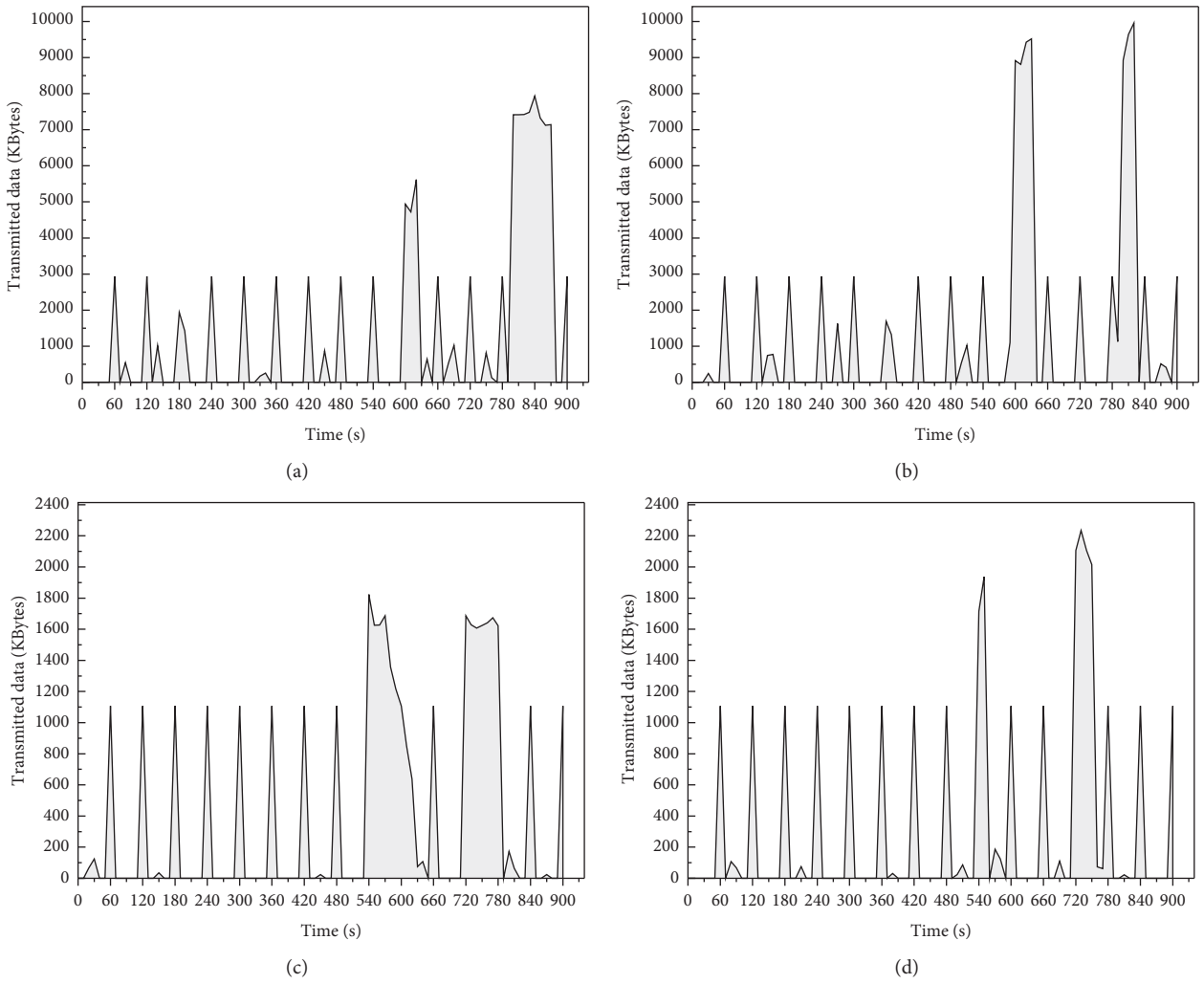

Figure 11: Communication overhead for stateful data migration: (a) TwitterSentiment on Original Flink; (b) TwitterSentiment on Flink-ER; (c) Streaming-Benchmarks on Original Flink; (d) Streaming-Benchmarks on Flink-ER.

(2) Existing online data migrating and task scheduling strategies may produce results with performance fluctuations and declining cluster stability, which are not appropriate for industry application

(3) As a resource optimizing method, the existing scheduling strategy requires excessive computing resources, which affect the performance of data processing and is not acceptable in resource-intensive application scenarios

To address the above problems, Flink with elastic resource scheduling (Flink-ER) is proposed. The fundamental differences between this paper and existing studies are summarized as follows:

(1) Existing studies change the scale and operator parallelism without considering the utilization of existing resources. This paper makes full use of existing computing resources by improving the throughput before executing elastic resource scheduling.

(2) Existing studies do not take the stateful data [36] into consideration during task scheduling. This paper proposed a stateful data management model and stateful data migration algorithm that efficiently reduced communication overhead during task scheduling.

(3) In general, the existing studies propose resource scheduling strategies based on resource utilization (including CPU, RAM, and network bandwidth). This paper comprehensively considers the processing latency and throughput of the cluster to directly and effectively disrupt the performance bottleneck.

6. Conclusions and Future Work

As the most popular DSPS, Apache Flink has been widely applied for processing the mobile information data stream. However, the lack of a resource-rescheduling strategy results in a decline in performance and limits the further development of the platform. Therefore, Flink with an elastic resource scheduling strategy (Flink-ER) is proposed in this paper. The experimental results show that the strategy efficiently promotes the throughput of the cluster and reduces the communication overhead of stateful data migration under latency constraints by optimizing the data distribution strategy, elastically rescheduling the resources and effectively reducing the data migration overhead.

In future work, the problem of instantaneous break out faced by existing studies will be addressed, and we will propose an online elastic resource scheduling strategy without any break out for the data processing thread. Therefore, our possible future work can be summarized as follows:

(1) To solve the problem of instantaneous break out, an online stateful data and task migration algorithm should be designed and implemented

(2) To reduce dependence on outside components such as Zookeeper, the Flink master should be improved to independently carry out elastic resource scheduling

(3) To widely apply a data stream processing system in the mobile information processing area, more relevant problems should be resolved in more application scenarios

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the National Natural Science Foundation of China (Grant nos. 61862060, 61462079, and 61562086), Natural Science Foundation of Xinjiang Uygur Autonomous Region of China (2017D01A20), Educational Research Program of Xinjiang Uygur Autonomous Region of China (XJEDU2016S106), and Doctoral Innovation Program of Xinjiang University (Grant no. XJUBSCX-201902).

References

[1] X. Hu, W. Quan, T. Guo, Yu Liu, and L. Zhang, "Mobile edge assisted live streaming system for omnidirectional video," *Mobile Information Systems*, vol. 2019, Article ID 8487372, 15 pages, 2019.

[2] Seagate, Data age 2025. 2019, https://www.seagate.com/files/www-content/our-story/trends/files/data-age-2025-white-paper-simplified-chinese.pdf.

[3] K. Sasaki and T. Inoue, "Coordinating real-time serial cooperative work by cuing the order in the case of theatrical performance practice," *Mobile Information Systems*, vol. 2019, Article ID 4545917, 10 pages, 2019.

[4] D. Sun, G. Zhang, and W. Zheng, "Big data stream computing: technologies and instances," *Journal of Software*, vol. 25, no. 4, pp. 839–862, 2014.

[5] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache flink," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

[6] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," 2015, https://arxiv.org/abs/1506.08603.

[7] A. Alexandrov, R. Bergmann, S. Ewen et al., "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[9] H. Mueller-Merbach, "An improved starting algorithm for the ford-fulkerson approach to the transportation problem," *Management Science*, vol. 13, no. 1, pp. 97–104, 1966.

[10] H. Thomas, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Massachusetts: Institute of Technology, Cambridge, MA, USA, 2009.

[11] A. Rosà, L. Y. Chen, and B.. Walter, "Profiling actor utilization and communication in Akka," in *Proceedings of the 15th International Workshop on Erlang*, ACM, Nara, Japan, September 2016.

[12] D. Sun, S. Gao, X. Liu, F. Li, X. Zheng, and R. Buyya, "State and runtime-aware scheduling in elastic stream computing systems," *Future Generation Computer Systems*, vol. 97, pp. 194–209, 2019.

[13] GitHub, Apache Flink, 2019, https://github.com/apache/flink.

[14] Fabian Hueske, Incubator-Flink, 2019, https://github.com/physikerwelt/incubator-flink.

[15] GitHub, DataArtisans: Yahoo Streaming-Benchmarks. 2019, https://github.com/dataArtisans/yahoo-streaming-benchmark/.

[16] D. Sun, G. Fu, X. Liu, and H. Zhang, "Optimizing data stream graph for big data stream computing in cloud datacenter environments," *International Journal of Advancements in Computing Technology*, vol. 6, no. 5, pp. 53–65, 2014.

[17] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, and K. Li, "Re-Stream: real-time and energy-efficient resource scheduling in big data stream computing environments," *Information Sciences*, vol. 319, no. 13, pp. 92–112, 2015.

[18] C. Zhang, X. Chen, Z. Li et al., "An On-The-Fly Scheduling Strategy for Distributed Stream Processing Platform," in *Proceedings of the 2018 IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, IEEE, Guangzhou, China, pp. 773–780, December 2018.

[19] A. Shukla and Y. Simmhan, "Model-driven scheduling for distributed stream processing systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 98–114, 2018.

[20] V. Cardellini, G. Mencagli, D. Talia, and M. Torquati, "New landscapes of the data stream processing in the era of fog computing," *Future Generation Computer Systems*, vol. 99, pp. 646–650, 2019.

[21] N. Tantalaki, S. Souravlas, M. Roumeliotis et al., "Linear scheduling of big data streams on multiprocessor sets in the cloud," in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pp. 107–115, ACM, Thessaloniki, Greece, October 2019.

[22] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "T3-Scheduler: a topology and Traffic aware two-level Scheduler for stream processing systems in a heterogeneous cluster," *Future Generation Computer Systems*, vol. 89, pp. 617–632, 2018.

[23] A. Silva Veith, F. R. de Souza, M. D. de Assunção et al., "Multi-objective reinforcement learning for reconfiguring data stream analytics on edge computing," in *Proceedings of the 48th International Conference on Parallel Processing*, vol. 106, ACM, New York; NY, USA, August 2019.

[24] T. Loukopoulos, N. Tziritas, M. Koziri et al., "A pareto-efficient algorithm for data stream processing at network edges," in *Proceedings of the 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 159–162, IEEE, Nicosia, Cyprus, December 2018.

[25] A. Pagliari, F. Huet, and G. Urvoy-Keller, "On the cost of acking in data stream processing systems," in *Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud, and Grid*, Larnaca, Cyprus, May 2019.

[26] S. Zhou, F. Zhang, H. Chen et al., "FastJoin: a skewness-aware distributed stream join system," in *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, Rio de Janeiro, Brazil, pp. 1042–1052, May 2019.

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

[28] A. Jonathan, A. Chandra, and J. Weissman, "Rethinking adaptability in wide-area stream processing systems," in *Proceedings of the 10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, USA, July 2018.

[29] N. Hidalgo, E. Rosas, C. Vasquez, and D. Wladdimiro, "Measuring stream processing systems adaptability under dynamic workloads," *Future Generation Computer Systems*, vol. 88, pp. 413–423, 2018.

[30] G. R. Russo, V. Cardellini, and F. L. Presti, "Reinforcement learning based policies for elastic stream processing on heterogeneous resources," in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, ACM, Darmstadt, Germany, pp. 31–42, June 2019.

[31] G. Russo Russo, M. Nardelli, V. Cardellini et al., "Multi-Level elasticity for wide-area data streaming systems: a reinforcement learning approach," *Algorithms*, vol. 11, no. 9, 2018.

[32] V. Cardellini, F. L. Presti, M. Nardelli et al., *Towards Hierarchical Autonomous Control for Elastic Data Stream Processing in the Fog, European Conference on Parallel Processing*, Springer, Cham, Switzerland, 2017.

[33] M. Belkhiria and C. Tedeschi, "A fully decentralized autoscaling algorithm for stream processing applications," in *Proceedings of the International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing*, Göttingen, Germany, 2019.

[34] G. R. Russo, "Self-adaptive data stream processing in Geo-distributed computing environments," in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, ACM, Darmstadt, Germany, pp. 276–279, June 2019.

[35] G. Mencagli, M. Torquati, and M. Danelutto, "Elastic-PPQ: a two-level autonomic system for spatial preference query processing over dynamic data streams," *Future Generation Computer Systems*, vol. 79, pp. 862–877, 2018.

[36] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems and Software*, vol. 127, pp. 205–216, 2017.

[37] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems*, IEEE, Columbus, OH, USA, pp. 399–410, June 2015.

[38] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018.

[39] S. Liu, J. Weng, J. H. Wang et al., "An adaptive online scheme for scheduling and resource enforcement in Storm," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, 2019.

[40] A. Floratou and A. Agrawal, "Self-regulating streaming systems: challenges and opportunities, proceedings of the international Workshop on real-time Business intelligence and analytics," *ACM*, vol. 1, 2017.

[41] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *The VLDB Journal*, vol. 27, no. 6, pp. 847–872, 2018.