Elastic Scaling of Stateful Operators Over Fluctuating Data Streams

Minghui Wu, Dawei Sun, Shang Gao, Member, IEEE,
Keqin Li, Follow, IEEE and Rajkumar Buyya, Follow, IEEE

***Abstract*—Elastic scaling of parallel operators has emerged as a powerful approach to reduce response time in stream applications with fluctuating inputs. Many state-of-the-art works focus on stateless operators and change the operator parallelism from one aspect. They often lack efficient management of operator states and overlook the costs associated with resource over-provisioning. To overcome these limitations, we introduce Es-Stream for elastic scaling of stateful operators over fluctuating data streams, which includes: (1) We observe that under-provisioning of operator parallelism leads to data pile-up, resulting in longer system latency, while over-provisioning of operator parallelism causes idle instances and additional resource consumption. (2) The Es-Stream system scales in two dimensions: the parallelism of operators and the number of resources. It dynamically adjusts operators to an optimal parallelism while scaling the resources used by the stream application. (3) When the parallelism of stateful operators changes, upstream operators backup downstream operators' state and cache the emitted data tuples at dynamic time intervals, ensuring the operator parallelism is adjusted in a low-overhead way. (4) Experimental results demonstrate that Es-Stream provides promising performance improvements, reducing the maximum system latency by 3x and saving the maximum state recovery time by 2x, compared to existing state-of-the-art works.**

***Index Terms*—Operator parallelism, Resource scaling, State management, Stateful operator, Distributed stream computing.**

I. Introduction

PROCESSING continuous data streams in a scalable and timely manner is becoming crucial for applications such as Internet of Things (IoT), traffic monitoring, telecommunications and health care [1], [2]. These stream-oriented applications need to quickly analyze large volumes of continuous data and produce predictable and actionable results in a high-performance computing environment [3], [4]. The performance of a stream computing system can be affected by multiple factors, including computing resource, operator parallelism, memory settings and buffer sizes [5]. Among them, the degree of operator parallelism plays an important role in exploiting the system performance [6]. Automatic adjustment to the operator parallelism for performance optimization has become a key challenge [7].

A real-time stream application running in a stream computing system can be modeled as a directed acyclic graph (DAG) which describes the dependency between its tasks [8]. The DAG is submitted to the cluster for deployment and each task in the DAG is scheduled to compute node in the cluster for execution. If there are no manual intervenes or system failures, the deployed applications will run forever [9]. In this case, the operator parallelism settings of DAGs are static and cannot adapt to the fluctuating data stream rates [10]. This brings two negative effects: (1) When the arrival rate of data stream exceeds the tuple processing bottleneck of the system, a large amount of data will pile up, leading to slow latency and even system crash. (2) When the arrival rate is consistently low, the computing resources occupied by the operators of DAG cannot be dynamically recycled, causing the system to generate idle resource consumption.

To ensure that data streams are processed with low latency and resource efficiency, an elastic scaling mechanism for operators is essential [11]. This mechanism is expected to dynamically adjust the degree of parallelism between the operators of DAG for low latency and effective resource allocation. However, many existing works [12], [13] don't provide a suitable method that supports operator scaling up/down and coordinates resources between operators of a DAG to dynamically allocate and release resources based on current data rate. For example, DRS [12] allocated resources along one dimension by gradually increasing the parallelism of the operator that benefits the most in the topology. However, when the resources used by the stream application are inadequate, simply changing the operator parallelism may not improve the system performance [6].

Recent research [10], [14] has been developed to incorporate multiple aspects for optimizing the parallelism of operators. Specifically, [14] presented a platform that supports approximate computing and scales the parallelism of operators when resources are sufficient. [10] focused on balancing the load across stateful operator instances while scaling the parallelism, offloading tasks from overloaded instances to new one. Despite their great efforts, both studies overlooked the fact that adjusting the operator parallelism using a greedy algorithm can lead to increased system overhead. Additionally, improper instances deployment during scaling can result in the over-provisioning of compute nodes and resource waste.

Changing the operator parallelism of streaming applications

This work is supported by the National Natural Science Foundation of China under Grant No. 62372419; the Fundamental Research Funds for the Central Universities under Grant No. 265QZ2021001. (Corresponding author: Dawei Sun.)

Minghui Wu and Dawei Sun are with the School of Information Engineering, China University of Geosciences, Beijing, 100083, China (e-mail: wuminghui@email.cugb.edu.cn; sundaweicn@cugb.edu.cn)

Shang Gao is with the School of Information Technology, Deakin University, Waurn Ponds, Victoria, 3216, Australia (e-mail: shang.gao@deakin.edu.an)

Keqin Li is with the Department of Computer Science, State University of New York,New Paltz, New York, 12561, USA (e-mail: lik@newpaltz.edu)

Rajkumar Buyya is with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Austral (e-mail: rbuyya@unimelb.edu.au)

can improve the system performance [13], however, it also introduces new challenges for the state management of tasks, further complicating the auto-scaling mechanism. Changes to the DAG's structure can cause the inconsistency of data dependency between the backups and the states of operators before and after auto-scaling [15]. If the backups are re-partitioned according to the scaled operators, additional overhead can be generated. Should there be a better state management method and an operator scaling up/down mechanism to dynamically adjust the parallelism of stream applications for fluctuating data streams, the system stability and performance may be improved. These ideas motivate our research on elastic scaling of parallel operators for fluctuation data streams.

To address the aforementioned issues, we scale parallel operators to reduces the system latency and ensure the state reliability of application operators. The following questions are to be resolved: (1) Scaling operator parallelism. In a streaming application, how to tune the ratio of parallelism degrees between operators so that the system latency is minimized when the application occupies fixed computing resources? (2) Scaling resources. How to dynamically allocate more resources to an application when its occupied resources are not sufficient? (3) Low-overhead state recovery. How to effectively guarantee consistency between the processed data tuples and the stored task states if the application topology is changed?

A. Contributions

In this paper, a reliable resource scaling in/out framework (Es-Stream) is proposed to reduce the response time of distributed stream computing systems. Our contributions are summarized as follows:

(1) We observe that under-provisioning of operator parallelism leads to data pile-up and resource constraints, while over-provisioning causes idle instances and extra resource usage. This finding reveals the inherent conflict between operator parallelism and resource dynamics.
(2) A data tuple queuing model based on the M/M/k system is built to optimize the operator parallelism and achieve a trade-off between the system latency and resource consumption. Resource scaling is performed using skewed distribution to evaluate the resource consumption.
(3) Upstream backup of operator states and data tuple caches in dynamic time intervals is achieved to reduce the state recovery cost. In addition, the backup interval can be dynamically changed based on the resource load of compute node to lower the system overhead.

Experimental results show that Es-Stream makes promising improvements in resource configuration and state management compared to existing works.

B. Paper organization

The rest of this paper is organized as follows. Section II discusses the effect of parallelism on system latency and the motivation for the research. Section III introduces the system models, including the stream application model, communication model, data model and resource constraint model; Section IV formalizes the resource allocation problem between the operators and the states before and after scaling; Section V introduces the Es-Stream system and its main algorithms; Section VI evaluates the performance of Es-Stream; Section VII presents related work and Section VIII concludes our work along with future directions.

II. Observation and motivation

A series of experiments are designed on the Storm 2.4.0 platform to identify the impact of different parallelisms on the system latency and resource utilization, thus leading to the motivation of our research.

We use the public data [16] from AliCloud to evaluate the system performance. The system' s cluster comprises 16 machines, each powered by an Intel(R) Xeon(R) X5650 CPU (dual-core, 2.4 GHz), equipped with 2 GB of RAM and a 100 Mbps Ethernet interface card. Among the 16 machines in the cluster, 3 run Nimbus as master nodes, and 13 deploy Supervisor nodes. 3 machines (3 multiplexed with the Nimbus nodes) deploy the Zookeeper cluster. We configure each compute node to deploy a maximum of two Workers, with each Worker running up to two operator instances. The average size of tuples emitted by the data source is 92 bytes. Two stream applications, COMMCount and Top_N, are submitted to the cluster using the EvenScheduler, which are commonly used for performance test and analysis of stream computing system. EvenScheduler employs a round-robin strategy to evenly distribute these instances across compute nodes. Once deployed, the operator instances run continuously unless there is a failure or manual intervention. COMMCount and Top_N count the number of browsing commodities and the most purchased commodities, respectively. The function and logic graph of each operator in their DAGs are similar to [17].

A. Observations

As shown in Fig. 1, we allocate different resources to the COMMCount topology by setting different instance numbers to the operators. It can be observed that the system latency under different resource configurations is different, and the optimal resource configuration changes with the data rate. When the input data rate is kept stable at 900 tuples/s, the optimal instance number for operators is (3,10,11). However, when the input rates are 1,800 tuples/s and 2,700 tuples/s, the optimal configurations for COMMCount topology are (3,13,8) and (8,15,6), respectively. This result shows that there exists an optimal configuration producing minimal system latency for a topology, and this configuration usually changes with the input data rate.

As shown in Fig. 2, different resources are configured to the Top_N topology to test the system latency. It can be observed that there exists an optimal configuration. When the input is 900 tuples/s, the optimal and worst configurations for Top_N topology are (3,10,6,1) and (3,9,7,1), respectively. The main reason behind is that the computing resources occupied by different numbers of parallel operators are different, and poor resource allocation between operators will cause longer

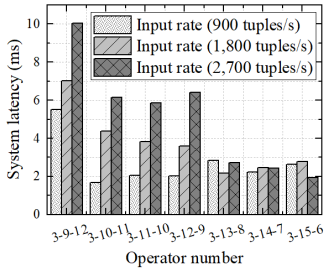


Fig. 1. System latency under different resource configurations for COMMCount topology.

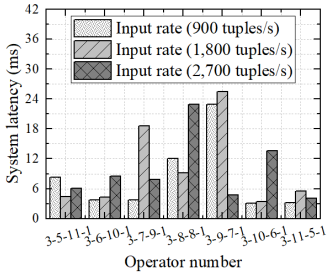


Fig. 2. System latency under different resource configurations for Top_N topology.

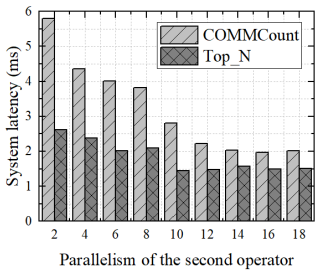


Fig. 3. System latency under different instance numbers of the second operator.

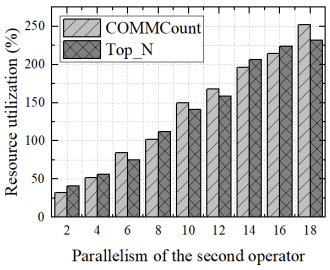


Fig. 4. Resource utilization under different instance numbers of the second operator.

queuing time for data processing. In this experiment, the system latency exhibits fluctuations because resource competition and data queue buildup alternately become the primary factor affecting the latency under different topology configurations. In addition, varying data stream rates lead to varying optimal resource allocations between operators in the topology. For example, at configuration (3,9,7,1), the input rate of 2,700 tuples/s has a lower latency than those of 900 tuples/s and 1,800 tuples/s, because the input rate of 2,700 tuples/s receives a better resource allocation between the operators. This observation highlights the importance of finding a dynamic scaling mechanism for varying numbers of operator instances to handle fluctuating data streams.

In Fig. 3, the input rate is stabilized at 1,800 tuples/s, and the instance number of the second operator gradually increases. When the instance number is less than 10, increasing the operator parallelism helps decrease the system latency. However, when the number is greater than 10, the system latency gradually becomes stable. The result shows that an excessive instance number no longer helps diminish the system latency. Therefore, it is important to set a proper instance number for each operator to lower the resource consumption.

Resource consumption of a topology increases with the number of instances. As shown in Fig. 4, with the input rate of data stream stabilized at 100 tuples/s, the resources consumed by all the instances of the second operator increase with the number of instances. The main reason is that a new operator instance takes up memory resources and consumes certain CPU resources, also incurs additional communication overhead. It is important to set a proper instance number for each operator to optimize the system latency and resource consumption.

B. Motivations

Based on the above observation and analysis, it can be seen that the system latency can be affected by different resource configurations to the topological operators. The optimal configuration for a given topology is dynamic and affected by the input rate. A poor resource allocation to operators will likely generate longer latency and consume more resources. It is wise to consider changing the operator parallelism at runtime for performance optimization. However, manual adjustment to the instance number of operator would incur expensive costs and make it more difficult to maintain the operator states. To optimize the performance and state recovery, an elastic scaling method for parallel operators may help. Our motivations can be summarized as follows:

(1) Given the input rate, output rate of instance and resource load, how to achieve the trade-off between the instance number of the operators and the resources consumed by the instances in a comprehensive way when targeting low system latency?

(2) Given the resource load of the cluster and resources consumed by topologies, at what time to scale the number of resources consumed by topologies when targeting high system throughput?

(3) How to effectively maintain the consistency between the processed data tuples and the stored task states when the parallelism of stateful operators changes?

III. System Model

Before formalizing the resource allocation and state management problem and introducing our proposal, we first establish models for stream application, communication load and resource constraint.

A. Stream application model

In a stream computing environment, user defines a topology [17] for a given stream application and then submits it to the computing cluster. The topology can be described as a directed acyclic graph $G = \{V(G), E(G)\}$, composed of a finite vertex set $V(G) = \{v_1, v_2, ..., v_i, ..., v_n\}$ and a finite directed edge set $E(G) = \{e_{i,j} | v_i, v_j \in V(G)\}$. A vertex v_i represents an operator in the stream application, which is implemented by a specific function $f(v_i)$ defined by the user. The function $f(v_i)$ of each vertex v_i is different, that is if $\forall v_i, v_j \in V(G)$, $v_i \neq v_j$, then $f(v_i) \neq f(v_j)$. Edge $e_{i,j}$ between vertex v_i and vertex v_j describes the dependency between the two vertices and can be used to represent their communication load.

When a stream application is committed to the data centre, the application is deployed by the scheduler. In this process, multiple instances of vertex v_i can be initialized with the same function based on the parallelism set by the user, where $\forall v_{i,k}, v_{i,m} \in v_i$, then $\exists f(v_{i,k}) = f(v_{i,m})$. The dependency between instances is described as a directed acyclic instance topology $IT = \{V(IT), E(IT)\}$, and $V(IT) \subseteq V(G)$, $E(IT) \subseteq E(G)$. This instance topology is allocated with computing resources by the scheduler.

B. Communication load model

In the mapped instance topology $IT = \{V(IT), E(IT)\}$, we use edges $E(IT)$ to describe the transmission rates between instances. E.g., $tr(v_{i,k}, v_{j,m})$ represents the data tuples' rate that instance $v_{j,m}$ receives from an upstream instance $v_{i,k}$. At time t, the number of data tuples received by instance $v_{j,m}$ from upstream can be calculated by Eq. (1).

$$w_{j,m} = \sum_{v_{i,k} \in pre(v_{j,m})} tr(v_{i,k}, v_{j,m}), \tag{1}$$

where $w_{j,m}$ denotes the number of tuples received by instance $v_{j,m}$ per time unit, and $pre(v_{j,m})$ is the set of immediate predecessor instances of instance $v_{j,m}$.

Since there may be transient fluctuations in the data arrival rate, we calculate the average of $w_{j,m}$ by Eq. (2) to ease the impact brought by sudden fluctuations.

$$E_{w_{j,m}} = \frac{\int_{t_s}^{t_e} w_{j,m} dt}{t_e - t_s}, \tag{2}$$

where $E_{w_{j,m}}$ represents the average input rate of instance $v_{j,m}$ in time interval $[t_s, t_e]$. t_s and t_e denote the start time and end time of a given short time period which can be set by users.

Then, the average input rate for operator v_j can be calculated by Eq. (3).

$$Ir_{v_j} = \sum_{w_{j,m} \in v_j} E_{w_{j,m}}, \tag{3}$$

where Ir_{v_j} is the sum of average input rates of all instances of operator v_j during $[t_e, t_s]$, which provides data support for the elastic scaling of operator parallelism degrees. The larger the Ir_{v_j}, the more instances that operator v_j needs to reduce the sojourn time of data tuples in v_j. If Ir_{v_j} is small and the parallelism of operator v_j is relatively large, operator v_j will consume extra resources. Therefore, making the operator's instance number adaptive to varying Ir_{v_j} is important.

C. Resource constraint model

The latency of the instance changes with the resource load of the compute node [18]. Resource overload can cause a node to experience downtime. It is necessary to model the constraints on node resources to ensure the cluster to work uninterrupted. The resource load of compute nodes can be measured in different dimensions, such as CPU, memory and I/O. We assume that each compute node in the cluster does not deploy other services except for stream applications. Given a cluster with s compute nodes $CN = \{cn_1, cn_2, ..., cn_s\}$, the CPU utilization, memory utilization and I/O utilization consumed by an instance of the stream application are denoted as $r^c_{v_{i,j}}, r^m_{v_{i,j}}$ and $r^i_{v_{i,j}}$, respectively. Multiple instances are deployed on a compute node. Therefore, the CPU, memory and I/O utilization of a compute node cn_k can be calculated by Eq. (4).

$$\begin{cases} R^c_{cn_k} = \sum\limits_{v_{i,j} \in cn_k} r^c_{v_{i,j}} + b^c_{cn_k}, \\ R^m_{cn_k} = \sum\limits_{v_{i,j} \in cn_k} r^m_{v_{i,j}} + b^m_{cn_k}, \\ R^i_{cn_k} = \sum\limits_{v_{i,j} \in cn_k} r^i_{v_{i,j}} + b^i_{cn_k}, \end{cases} \tag{4}$$

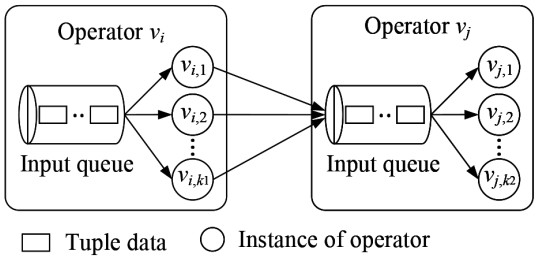


Fig. 5. Parallelism of operators

where $R^c_{cn_k}, R^m_{cn_k}$ and $R^i_{cn_k}$ denote the CPU, memory and I/O utilization of the compute node cn_k, respectively. $b^c_{cn_k}, b^m_{cn_k}$ and $b^i_{cn_k}$ denotes the static CPU, memory, and I/O utilization of the compute node cn_k without running other services.

Once a stream application is submitted to the cluster, data are continuously generated and processed. The instance deployment caused by the elastic scaling of operator parallelism degrees should not lead to resource overload on the compute nodes. The following condition (5) is to be satisfied.

$$R^c_{cn_k} \leq \alpha_c, R^m_{cn_k} \leq \alpha_m, R^i_{cn_k} \leq \alpha_i, \tag{5}$$

where α_c, α_m and α_i denote the maximum CPU, memory and I/O resources consumed by the compute node cn_k, respectively.

Based on this condition (5), it can be known that the system performance can be affected by any of the CPU, Memory and I/O [19]. Therefore, the resource load R_{cn_k} of compute node cn_k is represented by Eq. (6).

$$R_{cn_k} = \max\{R^c_{cn_k}, R^m_{cn_k}, R^i_{cn_k}\}. \tag{6}$$

IV. Problem Statement

In this section, we formalize the problem raised by the static configuration of application topology, which mainly includes the scalability of operator parallelism, state consistency and state recovery overhead.

A. Scalability of operators

Based on the above models (Section III), the resource allocation problem between operators can be described as follows. Poor resource allocation between operators can affect system performance or waste computing resources [13]. For example, in Fig. 5, there are two operators v_i and v_j, and the numbers of their instances are k_1 and k_2, respectively. The input rate Ir_{v_j} of the input queue for operator v_j can be calculated by the communication load model (Section III-B). Assume that the tuple processing rates of the instances in v_j are $\{\mu_{j,1}, \mu_{j,2}, ..., \mu_{j,k_2}\}$. Then, the tuple processing rate μ_{v_j} of operator v_j can be calculated by Eq. (7).

$$\mu_{v_j} = \sum_{m=1}^{k_2} \mu_{j,m}. \tag{7}$$

Obviously, operator v_j must have enough instances for tuple processing to keep up with the input rate. If $Ir_{v_j} > \mu_{v_j}$, the queue for operator v_j will keep getting longer, resulting

in the increase of tuple sojourn time of the operator, which further pushes the downtime risk of the instances. In addition, if $Ir_{v_j} << \mu_{v_j}$, some instances of operator v_j may experience idle time while waiting for input from the upstream. This is a waste of resources. Therefore, the number of instances for operator v_j is be adjusted to balance Ir_{v_j} and μ_{v_j} and minimize the sojourn time of data tuples in the operator.

The optimization problem for a operator can be described as Eq. (8).

$$z(k) = \beta \cdot k + (1-\beta) \cdot Q_{v_j}(k), \tag{8}$$

where β is the resources used by each instance of operator v_j, k is the number of instances of v_j. $Q_{v_j}(k)$ is the average queue length of k instances of operator v_j. Our objective now is to find the minimum value k^* in function $z(k)$ under the constraints of resource consumption and system latency.

Since k can only take integers, $z(k)$ is not a continuous function. We use marginal analysis to find the minimum value , which is described as condition (9).

$$\begin{cases} z(k^*) \leq z(k^*-1), \\ z(k^*) \leq z(k^*+1). \end{cases} \tag{9}$$

Taking Eq. (8) into condition (9), we get

$$\begin{cases} \beta \cdot k^* + (1-\beta) \cdot Q_{v_j}(k^*) \\ \quad \leq \beta \cdot (k^*-1) + (1-\beta) \cdot Q_{v_j}(k^*-1), \\ \beta \cdot k^* + (1-\beta) \cdot Q_{v_j}(k^*) \\ \quad \leq \beta \cdot (k^*+1) + (1-\beta) \cdot Q_{v_j}(k^*+1). \end{cases} \tag{10}$$

After simplifying condition (10), we get

$$\begin{aligned} Q_{v_j}(k^*) - Q_{v_j}(k^*+1) \leq \frac{\beta}{1-\beta} \leq \\ Q_{v_j}(k^*-1) - Q_{v_j}(k^*). \end{aligned} \tag{11}$$

Ultimately, our optimization problem is simplified to find an optimal number k^* which is the instance number of a given operator and satisfies the constraints of resource and system latency. If each operator satisfies the condition (11), the system will have less latency and less waste of resources.

B. State consistency

In a streaming application, intermediate results (i.e., states) are produced by an instance when processing data tuple dt. This state information is usually stored in memory at runtime. To improve the system reliability, the state information is usually backed up in remote storage by checkpoint mechanism. As shown in Fig. 6, instance $v_{i,1}$ emits four data tuples $dt1, dt2, dt3$ and $dt4$ to downstream instances $v_{j,1}$ and $v_{j,2}$ of operator v_j at time t_1. Instance $v_{j,1}$ receives data tuples $dt1$ and $dt2$, performs logical computation and produces states $(k1, v1)$ and $(k2, v2)$, where $k1$ is the key of tuple $dt1$ and $v1$ is the intermediate result of processing $dt1$ by any instance of operator v_j (as instances have the same processing logic). The process of data tuple dt_i is grouped by the Hash function, which can be described by Eq. (12).

$$v_j(m^{'}) = Hash(dt_i(k_i))\%m, \tag{12}$$

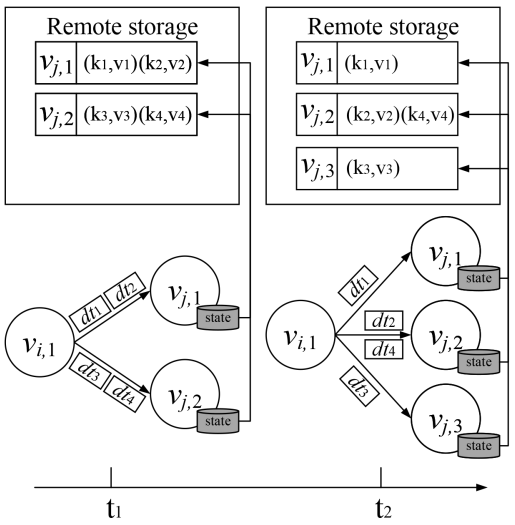


Fig. 6. States before and after the change to the number of operator instances.

where m denotes the instance number of operator v_j, and $v_j(m^{'})$ denotes that the data tuple dt_i will be emitted to the instance $v_{j,m'}$ for processing. In addition, the state of instance $v_{j,m'}$ is periodically backed up to the remote storage.

At time t_2, adding an instance to operator v_j allows instance $v_{i,1}$ to redirect the data stream, which is described by Eq. (13).

$$v_j^{'}(m^{'}) = Hash(dt_i(k_i))\%(m+\Delta), \tag{13}$$

where Δ denotes the variation in instance number. If $\Delta \neq 0$, then $v_j(m^{'}) \neq v_j^{'}(m^{'})$. As shown in Fig. 6, instance $v_{j,3}$ receives data tuple $dt3$ after adding an instance to operator v_j at time t_2, but the state of $dt3$ has been previously stored in instance $v_{j,2}$ at time t_1. Similarly, instance $v_{j,2}$ receives $dt2$ at time t_2, but the state of $dt2$ has been stored in instance $v_{j,1}$ at time t_1. As a result, the tuples processed by the instances are inconsistent with the previously stored states after the data stream is redirected.

C. State recovery overhead

In an environment with unbounded data stream, it is unavoidable to have failing tasks, which poses challenges in ensuring the reliability of data processing and resource scaling.

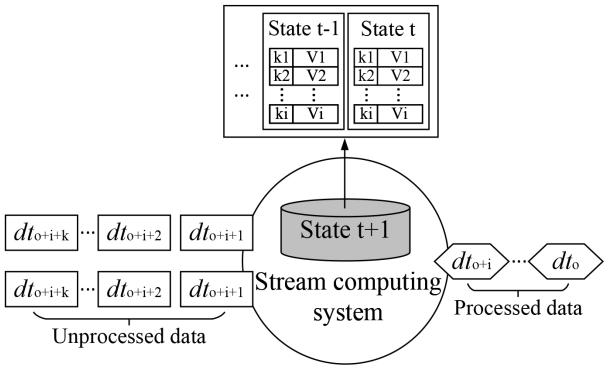


Fig. 7. State backup in Stream Computing System.

Checkpoint mechanism is usually used to cope with faulty tasks. However, it also introduces additional overhead, such as (1) Double computing. If a task fails, the system will lose part of the state data and perform a rollback to recalculate the lost state. As shown in Fig. 7, the latest state in the system is produced at $t+1$. When a task fails, the whole system needs to roll the task back to its state produced at t. In the process, the state at $t+1$ is lost and the data tuples are re-fed from tuple dt_o. (2) Global rollback. Once a task fails, the whole task topology needs to roll back to the prior state, discarding valid intermediate stream states.

V. Es-Stream: architecture and algorithms

Based on the above analysis, an elastic scaling method for operator parallelism, Es-Stream, is proposed. It aims to dynamically adjust the configuration of topology in a volatile data stream environment. This section provides an overview of Es-Stream, its architecture and the algorithms used for elastic scaling and adaptive state repair.

A. System architecture

As shown in Fig. 8, Es-Stream consists of the modules for monitor, topology analysis, resource analysis and state notification management.

The monitoring module is responsible for real-time information collection, including CPU, I/O and memory resource consumption of compute nodes and tasks in the cluster, the data transmission rates between tasks in the topology, and the running status of the stream system. The collected information is subsequently stored in the database.

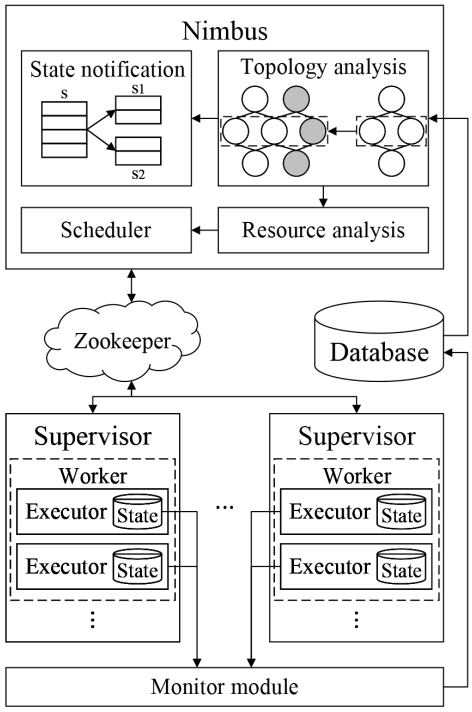


Fig. 8. Es-Stream architecture.

The topology analysis module focuses on optimizing the instance number for each operator in the topology. This instance number is a critical parameter during when topology initialization. A well-defined instance number for an operator can effectively improve the system throughput and reduce the response time. The topology analysis module retrieves the topology information stored in the database, analyzes and models these data to determine the optimal instance number for each operator in the topology.

The resource analysis module analyzes the resource load of the cluster and determines the appropriate nodes for deploying or recycling operator instances based on the results of topology analysis.

The state notification module notifies the tasks to repartition the backup state. Using the output from the topology analysis module, changes to the number of instances for stateful operators can be obtained. If the number of operator instances is scaled at runtime, the task's state should be repartitioned accordingly to ensure that the buffered data tuples are correctly mapped to the respective partitions.

Compared to traditional solutions, Es-Stream has the following advantages: (1) Traditional solutions configure the instance number for operators in a topology based on the stable rate of data stream, resulting in a static running topology in the cluster. Es-Stream, on the other hand, can dynamically perceive changes in the data stream rate and adjust the resource allocation weight between operators accordingly. In addition, Es-Stream can effectively reduce the resources used in low data stream rates scenarios. (2) Traditional solutions only adjust the number of instances for stateless operators, which limits their ability to improve system bottlenecks. Es-Stream can scale the number of instances for both the stateless and stateful operators for overall throughput improvement. Additionally, Es-Stream implements a flexible and low-overhead state recovery mechanism by backing up task states to upstream instances and caching data tuples within a checkpoint interval.

B. Elastic scaling

After operators of a streaming application are deployed to the compute nodes, there arises a need to automatically scale the instance number for operators to adapt to fluctuating data streams. Several factors need to be considered in this process. When the resources utilized by the streaming application are sufficient, scaling up/down the parallelism of operators within the stream application proves to be an effective way for reducing system latency. When the resources are inadequate, scaling out the resources becomes a viable solution to improving the system bottlenecks. This involves instantiating new instances of operators on additional compute nodes. Therefore, the elastic scaling mechanism primarily encompasses scaling operator parallelism and resource allocation.

(1) Scaling operator parallelism. We first focus on the instance number of operator v_i. Assume the average input rate of operator v_i in time interval $[t_e, t_s]$ is Ir_{v_i}, and each instance of operator v_i processes data tuples at the rate of $\{\mu_{i,1}, \mu_{i,2}, ..., \mu_{i,k}\}$, where k is the current instance number of

operator v_i, the average processing rate μ_{v_i} of v_i's instances can be calculated by Eq. (7).

If $Ir_{v_i} > \mu_{v_i} \cdot k$, operator v_i can not keep up with the incoming tuples, resulting in an increasing number of tuples in the operator queue over time and infinite queuing latency. In such cases, scaling up the parallelism for operator v_i becomes crucial for system performance optimization. Conversely, if $Ir_{v_i} < \mu_{v_i} \cdot k$, we model operator v_i as an $M/M/k$ queuing system [20] to further reduce the sojourn time of tuples processed by operator v_i.

Based on Erlang formula [12], it can be inferred:

$$p_n = P\{N = n\} = \begin{cases} \frac{\rho^n}{n!} \cdot p_0, & n = 1, ..., k, \\ \\ \frac{\rho^n}{k! \cdot k^{n-k}} \cdot p_0, & n \geq k, \end{cases} \tag{14}$$

where p_n denotes the probability distribution of queue length n for operator v_i in a steady environment, and p_0 and ρ can be calculated by Eq. (15).

$$p_0 = \left[\sum_{n=0}^{k-1} \frac{\rho^n}{n!} + \frac{\rho^k}{k! \cdot (1 - \frac{Ir_{v_i}}{k \cdot \mu_{v_i}})} \right]^{-1}, \rho = \frac{Ir_{v_i}}{\mu_{v_i}}. \tag{15}$$

The probability of n data tuples in operator v_i under steady conditions is given by Eq. (14). If $n \geq k$, the data tuple emitted by the upstream instance will be placed in the queue in operator v_i, awaiting processing. The waiting probability of data tuples form upstream can then be calculated by Eq. (16).

$$wp = \sum_{n=k}^{\infty} p_n = \frac{\rho^k}{k! \cdot (1 - \frac{Ir_{v_i}}{k \cdot \mu_{v_i}})} \cdot p_0. \tag{16}$$

Based on the waiting probability wp, the average queue length Ql can be calculated by Eq. (17).

$$\begin{aligned} Ql &= \sum_{n=k+1}^{\infty} (n-k) \cdot p_n \\ &= \frac{p_0 \cdot \rho^k}{k!} \cdot \sum_{n=k}^{\infty} (n-k) \cdot \left(\frac{Ir_{v_i}}{k \cdot \mu_{v_i}} \right)^{n-k} \\ &= \frac{wp \cdot \frac{Ir_{v_i}}{k \cdot \mu_{v_i}}}{1 - \frac{Ir_{v_i}}{k \cdot \mu_{v_i}}}. \end{aligned} \tag{17}$$

The number of data tuples in operator v_i mainly consists of the average queue length and the average number of tuples being processed. Hence, the average queue length Q_{v_i} for tuples in operator v_i can be calculated by Eq. (18).

$$\begin{aligned} Q_{v_i}(k) &= Ql + \rho \\ &= \frac{wp \cdot \frac{Ir_{v_i}}{k \cdot \mu_{v_i}}}{1 - \frac{Ir_{v_i}}{k \cdot \mu_{v_i}}} + \frac{Ir_{v_i}}{\mu_{v_i}} \\ &= \frac{\left(\frac{Ir_{v_i}}{\mu_{v_i}} \right)^k \cdot Ir_{v_i} \cdot p_0}{k! \cdot \left(1 - \frac{Ir_{v_i}}{k \cdot \mu_{v_i}} \right)^2 \cdot k \cdot \mu_{v_i}} + \frac{Ir_{v_i}}{\mu_{v_i}}, \end{aligned} \tag{18}$$

where $Q_{v_i}(k)$ denotes the average number of data tuples in operator v_i when the number of instances of operator v_i is k. If there exists a value k^* that makes $Q_{v_i}(k^*)$ satisfy condition (11), then k^* is considered the optimal number of instances for operator v_i.

From the stream application model, it is evident that a stream application can be regarded as a directed acyclic graph. Thus, the data tuples processed by the application follow a directed flow, with upstream instances emitting processed data tuples to downstream instances. When scaling up or down the number of instances of operator v_i, it does not affect the upstream operators' ability to process data tuples. However, it may result in downstream operator instances accumulating data tuples or generating idle resources. Therefore, adjusting the instance number of operators should be performed hierarchically, with priority given to scaling up/down the instance number of upstream operators. We utilize topological sorting to determine the sequence of scaling operators, as it can effectively analyze the dependencies between operators.

The algorithm for scaling up/down the instance number of operators for stream applications is described in Algorithm 1.

Algorithm 1: Scaling up/down the instance number of operators.

Input: Ir_{v_i} and μ_{v_i}.
Output: k^*.

1 Get the resource β used by the instance of operator v_i ;
2 Initialize the number k of instances for operator v_i, $k = \left\lfloor \frac{Ir_{v_i}}{\mu_{v_i}} \right\rfloor$;
`/* Search the optimal number of instances */`
3 **while** *true* **do**
4 Calculate the average queue length Q_k of k instances by (18);
5 Calculate the average queue length Q_{k+1} of $k+1$ instances by (18);
6 Calculate the average queue length Q_{k-1} of $k-1$ instances by (18);
7 **if** $Q_k - Q_{k+1} \leq \frac{\beta}{1-\beta} \leq Q_k - Q_{k-1}$ **then**
8 $k^* \leftarrow k$;
9 Break ;
10 **end**
11 $k++$;
12 **end**
13 **return** k^*

The input of Algorithm 1 includes the input rate Ir_{v_i} and processing rate μ_{v_i} of operator v_i. The output of the algorithm is the optimal number k^* of instances for operator v_i. Step 1 gets the average resource used by each instance of operator μ_{v_i}. Step 2 initializes the minimum number of operator instances, where the input rate equals the processing rate of the operator. Under the constraints of resource consumption and system latency, steps 4 to 13 search for the optimal number of operator instances. The time complexity of Algorithm 1 is $O(k^*)$, where k^* denotes the optimal number of instances.

(2) Scaling resources. Given a cluster $CN = \{cn_1, cn_2, ..., cn_s, ..., cn_{sc}\}$, where cn_s represents a compute node in the cluster of size sc, we assume the CPU utilization, memory

utilization and I/O utilization of a compute node cn_s are $R^c_{cn_s}, R^m_{cn_s}$ and $R^i_{cn_s}$, respectively. For a compute node cn_s, if its resource load satisfies condition (5), tasks deployed on that node can process data tuples efficiently. Considering all compute nodes in the cluster, when the data stream rate increases, to keep the resource load of all nodes satisfy the condition (5), we need to decide whether to deploy new instances on the fixed compute nodes used by the stream application or preempt new compute nodes. Similarly, when the data stream rate decreases, we need to determine whether to reduce the number of compute nodes used by the stream application. To address this problem, we construct a resource load model to evaluate the system's resource usage. It is described by Eq. (19).

$$SK = \frac{cs \cdot \sum_{i=1}^{cs} \left(R_{cn_i} - \overline{R}_{cn}\right)^3}{(cs-1)\cdot(cs-2)\cdot\sigma^3}, cs \leq sc, \tag{19}$$

where SK is the skewness coefficient of the resource load for the compute nodes used by the stream application, cs denotes the number of compute nodes used by the stream application, sc denotes the size of the cluster, $\overline{R}_{cn}$ is the average value of R_{cn_i}, and σ can be calculated by Eq. (20).

$$\sigma = \sqrt{\frac{\sum_{i=1}^{cs} \left(R_{cn_i} - \overline{R}_{cn}\right)^2}{cs}}, \tag{20}$$

where σ is the standard deviation of resource load for the compute nodes used by the stream application.

If $SK > 0$, it shows a right-skewed distribution of resource load among nodes, meaning that most nodes have a lower resource load compared to the average value $\overline{R}_{cn}$. If $SK < 0$, it implies a left-skewed distribution of resource load, where most nodes have a higher resource load than the average value $\overline{R}_{cn}$. In addition, a larger absolute value of SK indicates a more pronounced skewness in the load distribution among nodes.

Based on the above analysis, if $SK \leq \eta$ and $\overline{R}_{cn} \geq \varphi$, the system may approach a performance bottleneck. In this case, if the data stream rate continues to increase, the problem of data tuples piling up in the system can be addressed by adding some new instances of operators in the stream application. However, it is not advisable to deploy these new instances on the nodes already used by the topology, as the resource load of most nodes already exceeds φ. Instead, a better approach to mitigate the risk of introducing a bottleneck in the stream application is to deploy the new instances, generated through scaling up the topology, on separate compute nodes.

If $SK \geq \eta$ or $\overline{R}_{cn} \leq \varphi$, it indicates that the system has sufficient capacity to process more data tuples. In such cases, deploying new instances on the compute nodes used by the stream application can be explored. Consideration is first given to deploying the new instance on the nodes with the minimum resource load. If the nodes with the lowest resource load cannot meet the deployment conditions, an appropriate node will be selected from the entire cluster. In addition, the CPU resource r^c_{new} consumed by a new instance of operator v_i can be measured by Eq. (21).

$$r^c_{new} = \frac{1}{k} \cdot \sum_{v_{i,j} \in v_i} r^c_{v_{i,j}}, \tag{21}$$

where k denotes the optimal number of instances for operator v_i and $r^c_{v_{i,j}}$ denotes the CPU resource consumed by instance $v_{i,j}$ of operator v_i.

Similarly, the memory resource r^m_{new} and I/O resource r^i_{new} consumed by the new instance of the operator can be measured in the same way.

Based on the resource constraint model (Section III-C), deploying a new instance of an operator to the compute node cn_s needs to satisfy the conditions represented by Eq. (22).

$$\begin{cases} R^c_{cn_s} + r^c_{new} \leq \alpha_c, \\ R^m_{cn_s} + r^m_{new} \leq \alpha_m, \\ R^i_{cn_s} + r^i_{new} \leq \alpha_i. \end{cases} \tag{22}$$

If $SK \geq \chi$ and $\overline{R}_{cn} \leq \zeta$, it indicates that the system may have idle resources. In this case, if the data stream rate continuously decreases, reducing the number of instances can effectively prevent the stream application from consuming unnecessary resources. During this process, the compute nodes that were preempted by scaling out the resource for the stream application are prioritized for releasing resources to shrink the number of nodes used by the stream application.

C. State recovery

To support scaling up or down the parallelism for stateful operators, we design a data repartitioning mechanism that involves partitioning and merging data backups while auto-scaling the instances of operators at runtime. For each instance of a stateful operator, when the logical topology of instances is changed through elastic scaling, the backup for each instance needs to adjust the mapping relationship between the state of data tuples and the instance of the operator. This mapping adjustment process is described by (23).

$$S(v_{i,j}) \overset{Map(v_i)}{\longrightarrow} \{S(v_{i,1}, S_{i,1}), ..., S(v_{i,k^*}, S_{i,k^*})\}, \tag{23}$$

where $S(v_{i,j})$ denotes the backup state of instance $v_{i,j}$ in operator v_i, and S_{i,k^*} and k^* respectively denote the partial state of instance $v_{i,j}$ and the optimal number of instances after scaling the instances of operator v_i. The backup state is defined as the union of individual backup partitions, denoted as $S(v_{i,j}) = S_{i,1} \cup S_{i,2} \cup ... \cup S_{i,k^*}$, $S_{i,1} \cap S_{i,2} \cap ... \cap S_{i,k^*} = \emptyset$. Each partition $S_{i,m}$ corresponds to a specific instance of operator v_i. The repartitioned states $\{S_{i,1}, \ldots, S_{i,k^*}\}$ from the backup of instance $v_{i,j}$ are emitted to their corresponding instances. Upon receiving the states from the backups, the instance merges these state data, as described by Eq. (24).

$$S^{'}(v_{i,j}) = S_{1,j} \cup S_{2,j} \cup ... \cup S_{k^*,j}, \tag{24}$$

where $S^{'}(v_{i,j})$ denotes the merged state of instance $v_{i,j}$ and $S_{1,j} \cap S_{2,j} \cap ... \cap S_{k^*,j} = \emptyset$.

The overhead of this data repartitioning mechanism conforms to the following Theorem 1.

Theorem 1. *The repartitioning time RT of state S for operator v_i decreases as the number of scaled instances in operator v_i increases.*

Proof. When the parallelism of an operator changes, we need to repartition the operator's state to maintain state consistency. Let the state of operator v_i be S, its repartitioning time RT is the maximum time taken to repartition the state S across all instances of v_i. In this repartitioning process, each instance $v_{i,j}$ primarily incurs two time costs: the computation time for repartitioning the state and the transmission time for emitting the partitioned data to downstream instances. The repartitioning time RT of state S for operator v_i can be calculated by Eq. (25).

$$RT = \max\left(\frac{S}{k^* \cdot ce_j} + \frac{S}{k^* \cdot tr_j}\right), j = 1, 2, ..., k \quad (25)$$

where ce_j and tr_j respectively denote the computing efficiency and the transmission bandwidth of the instance backing up the state of $v_{i,j}$, and k^* denotes the optimal instance number for operator v_i.

From Eq. (25), it can be seen that RT is the repartitioning time of the worse-performing instance (i.e., with the lowest computing efficiency and bandwidth). When the optimal instance number k^* increases, the repartitioning time RT of the worse-performing instance can be significantly reduced.

To minimize overhead while achieving a state repair mechanism for scaling the parallelism of stateful operators, we also design a mechanism for the upstream operators to backup state and cache data tuples. This mechanism has two key aspects:

(1) State Backup: The state of instances in stateful operators is backed up to their upstream instances. Leveraging the existing communication connections in the logical ring formed by all operators in the stream application, each instance in a stateful operator manages its own state. If the resource load of a node is below a threshold value, denoted as α, the instance periodically synchronizes its state with the upstream. Otherwise, the sync interval is set to infinite.

(2) Output Result Backup: For a stateful operator, the output results of its upstream instances for processing data tuples are locally backed up within a specific time interval. When the parallelism of stateful operators changes, the upstream instances repartition the backup state and send the partitioned state to the corresponding downstream instances. Subsequently, the locally cached backup output results within the current time interval in the upstream instances are re-emitted to the downstream instance. This approach avoids the need for a global rollback of state across the entire topology and reduces system overhead.

The overhead of this state repair mechanism conforms to the following Theorem 2.

Theorem 2. *When operator v_i has failed instances, there is a positive correlation between the state repair time SR and the checkpoint interval time CI for operator v_i.*

Proof. When there are failed instances in operator v_i, merely pulling the state of operator v_i from the previous backup is insufficient because the state data processed during the current checkpoint interval has been lost. This state data has to be recovered. Let the average input rates of the f failed instances in operator v_i be $\{IR_{i,1}, IR_{i,2}, ..., IR_{i,f}\}$. In this state repairing process, upstream data emission and downstream data processing occur simultaneously, but they may have different completion time due to variations in workload. Therefore, the state repair time of each failed instance is the maximum between the upstream data emission time and the downstream data processing time. The state repair time SR for operator v_i is the maximum repair time across all the failed instances, which can be calculated by Eq. (26).

$$\begin{aligned} SR &= \max\left(\max\left(\frac{IR_{i,j} \cdot CI}{er_{i,j}}, \frac{IR_{i,j} \cdot CI}{\mu_{i,j}}\right)\right) \\ &= CI \cdot \max\left(\max\left(\frac{IR_{i,j}}{er_{i,j}}, \frac{IR_{i,j}}{\mu_{i,j}}\right)\right), j \in [1, f] \end{aligned} \quad (26)$$

where $er_{i,j}$ denotes the data receiving rate of the failed instance $v_{i,j}$ during the state repairing process, which is calculated by Eq. (3). $\mu_{i,j}$ denotes the processing rate of the failed instance $v_{i,j}$. f denotes the number of failed instances in operator v_i.

From Eq. (26), it can be seen that the state repair time SR for operator v_i is determined by the worst-performing instance (i.e., with the lowest computing efficiency and/or the slowest data receiving rate from upstream). Increasing or decreasing the checkpoint interval CI can directly affect the state repair time of the worst-performing instance.

VI. Performance evaluation

In the section, we focus on evaluating the proposed Es-Stream framework. Es-Stream was implemented and tested in a simulated real-world production environment. We utilized the public dataset from Alibaba Tianchi [16] to design two application scenarios: real-time statistics of product exposure (COMMCount topology) and identification of best-selling products (Top_N topology), both of which are commonly found in the e-commerce field. The experimental settings are the same as those in Section II. We evaluate two key performance metrics for the COMMCount and Top_N applications: system performance and system overhead for scaling the parallelism of stateful operators. The evaluation of Es-Stream aims to answer the following questions:

(1) Can Es-Stream improve system performance when processing a high data stream rate?
(2) Can Es-Stream scale up/down the number of operator instances and nodes in a streaming application to adapt to a fluctuating stream data rate?
(3) Can Es-Stream decrease the system overhead when scaling the parallelism for stateful operators?

A. System latency

System latency is the time interval from the input of data tuples to the output of the system. We evaluate the latency under different input rates, and compare it with state-of-the-art works. Among these works, EvenScheduler [21], R-Storm [22] and DRS [12] are the most representative in resource management and parallelism configurations. In addition, we

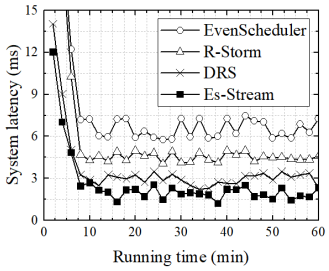


Fig. 9. System latency of COMM-Count under stable data rate.

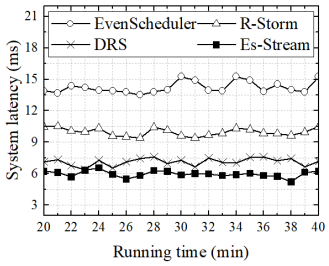


Fig. 10. System latency of Top_N under stable data rate.

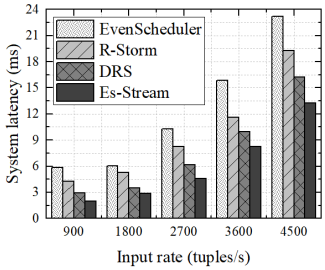


Fig. 11. System latency of COMM-Count under increasing data rate.

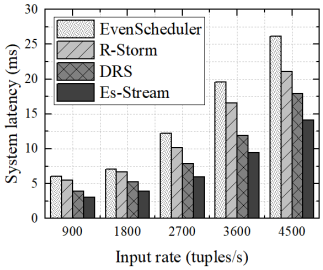


Fig. 12. System latency of Top_N under increasing data rate.

define the system stabilization time as the duration from the submission of a streaming application to the point where system latency variations become minimal. We consider the system to have reached a stable state when the system latency fluctuates below a preset threshold over a system-defined number of consecutive samplings. Specifically, if the difference between the maximum and minimum system latency over five consecutive samplings taken every 2 mins is less than 4 ms on Storm, we consider the system stable.

Given a stable data input rate of 1,800 tuples/s, Es-Stream exhibits lower latency compared to other solutions when the system becomes stable. Fig. 9 shows that the average response time of COMMCount topology are 2.1 ms, 6.4 ms, 4.5 ms and 2.9 ms for Es-Stream, EvenScheduler, R-Storm and DRS, respectively, after the systems stabilize. This indicates that Es-Stream outperforms EvenScheduler, R-Storm and DRS in terms of system latency.

Given a stable data input rate of 2,700 tuples/s, Es-Stream exhibits a shorter average response time compared to EvenScheduler, R-Storm and DRS after the systems stabilize. In Fig. 10, the average response of Top_N topology are 14.2 ms, 9.9 ms and 7.1 ms for EvenScheduler, R-Storm and DRS, respectively. However, Es-Stream achieves an average response time of 5.9 ms. This clearly demonstrates that Es-Stream has a lower latency compared to EvenScheduler, R-Storm and DRS. It was observed that DRS had the closest latency to Es-Stream, but Es-Stream consistently maintained a lower latency. This is because, while DRS is capable of constructing a well-optimized topology for streaming applications, it ignores the fact that changes in the parallelism of upstream operators can affect the data processing time of downstream operators.

Given an increasing data stream rate, Es-Stream consistently exhibits lower response time compared to EvenScheduler, R-Storm and DRS. Fig. 11 and 12 show the average response time for the COMMCount and Top_N topologies under different input rates after the systems stabilizes. The results show that the system latency increases as the input rate grows. However, Es-Stream shows a smaller incremental response time compared to EvenScheduler, R-Storm and DRS when the data stream rate increases. Overall, Es-Stream outperforms the other three in terms of system latency across different input rates.

In summary, Es-Stream has a lower system latency, because Es-Stream consistently identifies an optimal configuration for the operator parallelism of the topology under varying resource demands, and elastically scales the number of nodes used by the topology. Through these methods, Es-Stream effectively reduces the backlog of data tuples in operator instances, thereby further reducing the system latency.

B. *System bottleneck*

System bottleneck refers to the maximum data processing rate that a system can handle. As the data stream rate increases, if any task in a stream application fails, we consider the data stream rate at that point to be the system's bottleneck. To identify this bottleneck, we deploy two streaming applications respectively under EvenScheduler, R-Storm, DRS and Es-Stream in the same resource environment. To ensure fair comparison, all experiments use the same number of compute nodes, the same dataset, and applied identical stream rate increments.

Given an increasing data stream rate with increment of 500 tuples/s, Es-Stream has higher system throughput than EvenScheduler, R-Storm, and DRS. As shown in Fig. 13, in the two stream applications COMMCount and Top_N, Es-Stream's system bottlenecks are 15,000 tuples/s and 13,500 tuples/s, respectively, which are significantly higher than those of EvenScheduler, R-Storm, and DRS.

The reason for Es-Stream's better performance is its ability to dynamically adjust resource allocation based on operator requirements. For operators that need more resources, Es-Stream automatically configures higher parallelism; for those with lower demands, it configures lower parallelism. This dynamic adjustment ensures efficient resource utilization, reducing waste and competition among operators. By optimizing resource allocation, Es-Stream can optimize the system's processing capacity, thereby increasing overall throughput.

C. *Elastic scaling*

In this experiment, we aim to assess the effectiveness of the elastic scaling mechanism by observing the variations in the number of topology instances and computing nodes. To conduct the experiment, the streaming applications are deployed in a well-resourced cluster. For the stable input rate scenario, we set the time interval between adjustments in the parallelism of operators to 15 s. In the case of changing input rates, we set the adjustment time to 10 s. We disable the scaling of stateful operators, which will be discussed in the subsequent subsection. In addition, we set the trigger scaling factor to 0.2,

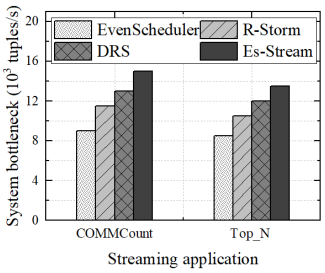


Fig. 13. System bottleneck of two streaming applications.

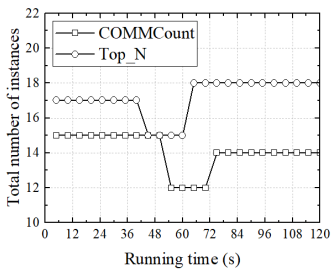


Fig. 14. Scaling up/down instances under stable input rate.

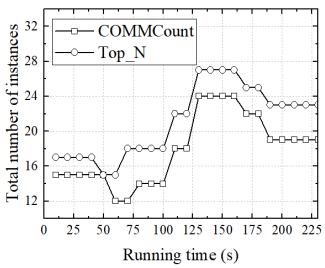


Fig. 15. Scaling up/down instances under changing input rate.

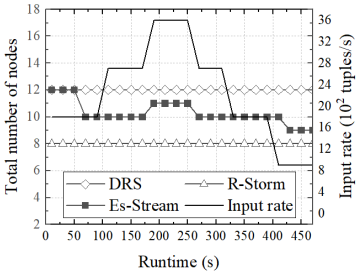


Fig. 16. Scaling up/down node number for COMMCount under fluctuating input rate.

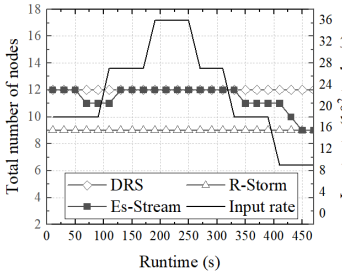


Fig. 17. Scaling up/down node number for Top_N under fluctuating input rate.

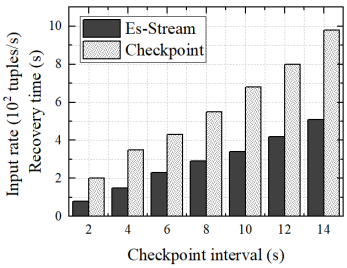


Fig. 18. Recovery time for stateful operator with different checkpoint intervals.

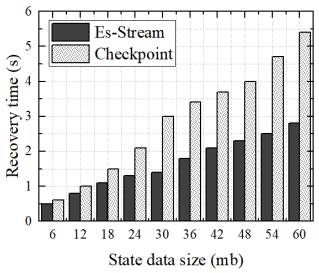


Fig. 19. State recovery time with increasing state data size.

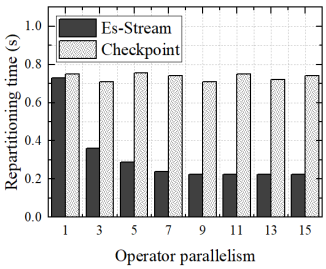


Fig. 20. Repartitioning time with increasing operator parallelism.

meaning that the data stream rate increases by more than 0.2x for scaling to be triggered.

Given a stable input rate of 1,800 tuples/s, Es-Stream demonstrates its ability to determine the optimal number of instances for each operator, thereby minimizing system latency. Fig. 14 illustrates the changes in instance numbers for the first and second operators in both the COMMCount and Top_N topologies overtime. The instance number of the first operator decreases, while that of the second operator increases. Moreover, after the system becomes stable, the total number of instances decreases for COMMCount and increases for Top_N. This observation suggests that different stream applications require varying levels of resources to process the data stream effectively. Es-Stream can automatically adjust the resources allocation of a stream application to adapt to the data stream rate.

Given a variable data stream rate, Es-Stream can adjust the number of instances in the topology to accommodate these variations. In our experiment, we change the data stream rate to 3,600 tuples/s at 100 s. As shown in Fig. 15, the total number of instances increases from 14 to 24 for COMMCount and from 18 to 27 for Top_N. Similarly, when we adjust the data stream rate to 2,700 tuples/s at 160 s, the total number of instances decreases from 24 to 19 for COMMCount and from 27 to 23 for Top_N. These results provide evidence that Es-Stream is able to elastically scale a stream application in response to fluctuating data streams.

Given a fluctuating data stream rate, Es-Stream can adjust the number of compute nodes for the topology to adapt to changing resource demands. As shown in Fig. 16 and Fig. 17, Es-Stream exhibits adaptability by dynamically adjusting the number of compute nodes based on the input rate, whereas DRS and R-Storm use a static number of nodes regardless of input rate fluctuations. In these experiments, Es-Stream dynamically adjusts node usage, indicating an adaptive design that scales with input load, while R-Storm maintains a minimal and consistent node usage, prioritizing resource efficiency. However, while R-Storm is resource-efficient, it may lead to system instability under high load conditions. DRS, with its static node usage, may ensure stability but at a higher resource cost. Therefore, Es-Stream is more suitable under fluctuating rates as it balances both resource efficiency and system stability.

D. System overhead

System overhead refers to the time required for scaling the parallelism of stateful operators and recovering from faulty nodes. In this experiment, we use COMMCount topology to evaluate the system overhead. The recovery time primarily includes the time of recovering state and the time of recomputing data tuples.

Given a stable data input rate of 1,800 tuples/s, the recovery time of Es-Stream for stateful operators is observed to be faster than that of the Checkpoint mechanism. As shown in Fig. 18, the recovery time for both Es-Stream and Checkpoint increases as the Checkpoint interval increases. However, Es-Stream exhibits a shorter recovery time compared to Checkpoint. In addition, the standard deviation of recovery time for Es-Stream and Checkpoint is 1.3 and 2.5, respectively. This indicates that the Es-Stream has a smaller fluctuation in recovery time as the checkpoint intervals increase, demonstrating its more stable and efficient performance compared to the checkpoint

mechanism. This improvement is mainly attributed to the fact that Es-Stream caches the data tuples of a checkpoint interval, and when stateful operators experience state recovery caused by scaling, it effectively reduces the state recovery time through partial state rollback.

Given a stable input data rate of 9,00 tuples/s and a checkpoint interval of 5s, the overhead of Es-Stream in terms of recovery time is observed to be smaller compared to the Checkpoint mechanism as the state data size increases. As shown in Fig. 19, the recovery time for both Es-Stream and Checkpoint gradually increases as the state data size grows. However, Es-Stream exhibits a shorter recovery time compared to Checkpoint, indicating its efficiency in handling larger state data sizes. In addition, the standard deviation of recovery time for Es-Stream and Checkpoint is 0.7 and 1.5, respectively, suggesting that Es-Stream achieves a more stable and consistent performance in recovering the state of instances for stateful operators with increasing state data size. Es-Stream performs well in increasing state size because Es-Stream performs concurrent state repartitioning and then merges the results, which enables Es-Stream to have better performance in executing larger state recovery.

Given the same data state size (e.g., 42 MB), Es-Stream demonstrates lower state repartitioning time through concurrent execution. As shown in Fig. 20, when the degree of parallelism increases, the time taken to repartition state using Es-Stream generally decreases until it stabilizes at 0.225 s. The time taken using Checkpoint does not show a consistent decreasing trend and remains relatively high, with slight variations around 0.71 s to 0.75 s. The reason Es-Stream shows lower state repartitioning times compared to Checkpoint is that Es-Stream performs concurrent state repartitioning and then merges these results. This concurrent approach allows Es-Stream to efficiently handle the repartitioning process, leading to reduced overall time compared to Checkpoint, which appears to handle the process in a less parallelized manner.

Given a stable data input rate of 1,800 tuples/s, Es-Stream exhibits faster fault recovery time for stateful operators compared to the Checkpoint mechanism. As shown in Fig. 21, Es-Stream consistently shows lower fault recovery times for every checkpoint interval. For instance, at the checkpoint interval time of 2 s, Es-Stream achieves a recovery time of 0.7 s, whereas Checkpoint requires 1.9 s. This trend holds across all interval times tested. In this experiment, the state repartitioning strategy is inactive due to unchanged operator parallelism. Es-Stream's efficiency in fault recovery time is attributed to upstream operators caching data tuples. When a stateful instance fails, these cached tuples are resent to the failed instance to facilitate partial rollback for the streaming application. In contrast, Checkpoint requires a global rollback of all task states across the entire streaming application by re-emitting tuples from the data sources.

VII. Related work

In this section, we review recent works in two related areas: elastic scaling of operators and state management of operators. A comparison between our work and the relevant research is summarized in Table I, where K_{max} denotes the maximum number of operator instances, k denotes the number of operator instances, and n denotes the number of operators in the topology.

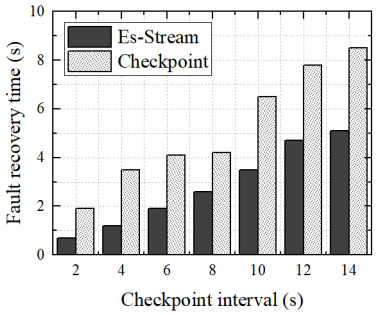


Fig. 21. Fault recovery time with different checkpoint intervals.

A. Elastic scaling operators

The elasticity of resource scaling in cloud computing environments has been widely researched. For example, [24]–[26] dynamically adjusted computing resources according to load variations, aiming to effectively utilize the compute nodes in cloud environments to reduce budget costs, while ensuring quality of service. However, in a stream computing environment, the strong dependencies between operators within a stream application make resource management more complex. Traditional elasticity scaling strategies in cloud computing environments cannot be directly applied to stream computing environments. To address this limitation, researchers have made efforts to optimize the scaling mechanisms for stream applications.

To balance operator parallelism and resource overhead, [8], [27] implemented operator managers using reinforcement learning to control the automatic scaling of operators. However, it may take a long time to collect information on operator parallelism changes for making a good operator scaling strategy. An optimal decision model requires continuous trial-and-error of the system, which incurs additional system overhead. Other research [28], [29] collected historical data on configuration parameters in cloud environments, such as operator parallelism, and applied machine learning to train models for determining suboptimal parameter configurations. However, collecting such historical data is time-consuming and labor-intensive, and the resulting models may not generalize well across different scenarios.

To ensure the consistency of operator states after scaling, Joker [30] iteratively increased the parallelism of operators until the system performance cannot be further optimized. In this process, it can adaptively redistribute operator states. However, iterative updating of operator parallelism can introduce more system overhead, and the redistributing of operator states with changing parallelism can exacerbate this problem. In addition, if the data processing is much smaller than the operator bottleneck, not promptly scaling down the parallelism will inevitably result in resource waste.

TABLE I
RELATED WORK COMPARISON

Aspect	Related Work					
	ELYSIUM [6]	DRS [12]	Li et al. [13]	EDS [15]	A-FP4S [23]	Our work
Operator parallelism	✔	✔	✔	✗	✔	✔
Resource scaling	✔	✔	✗	✗	✗	✔
State backup	✗	✗	✗	✔	✔	✔
Data rate awareness	✔	✗	✔	✔	✔	✔
Time complexity	$O(k \cdot n)$	$O(K_{max} \cdot n)$	$O(K_{max} \cdot n)$	Null	Null	$O(n \cdot e)$

In conclusion, the aforementioned solutions provide valuable insights for addressing elastic scaling operations. Es-Stream system, in particular, stands out by considering the resource overhead of operators and the sojourn time of tuples within operators. It also supports adjusting the operator parallelism, and scaling stateful operators through low-overhead state recovery mechanisms.

B. State management of operators

Effective state management not only reduces system overhead, but also improves system reliability [9]. To optimize state backup and recovery, an increasing number of researchers have studied different aspects of the area, such as upstream state backup, distributed state backup and state slicing.

A fault-tolerant mechanism was proposed by [15] to ensure state backup consistency and rollback recovery. The mechanism performs a self-adaptive upstream backup for operator state and elastic data slicing to enhance the reliability of operator state. To mitigate runtime overhead and output latency associated with full-backup mechanisms, an approximate fault-tolerant scheme was proposed by [31]. This scheme investigates the trade-off between fault-tolerance overhead and output accuracy in stream processing systems, aiming to minimize the overhead while meeting the accuracy requirements defined by users. However, this approach improves system performance at the cost of sacrificing some state accuracy.

In [32], the authors organize streaming application operators into a distributed hash table, where each operator is associated with a unique set of neighbors. It divides the in-memory state of each operator into multiple fragments and periodically saves them in the neighbors's node to ensure state reliability. However, this approach may face challenges when applied to state recovery caused by scaling the parallelism of operators.

Compared to the above state-of-the-art works, Es-Stream stands out by its ability to recover the state data when adjusting the parallelism of stateful operators. In addition, it incorporates a dynamic caching mechanism that effectively reduces system overhead by caching data tuples within a variable time interval.

VIII. CONCLUSIONS AND FUTURE WORK

In a volatile data flow rate scenario, the key objectives of a system implementation are to minimize system latency, reduce resource overhead, and ensure the reliability of system state. To achieve these objectives, it is essential to have a system that can sense the size of data stream and the resource consumption of operators in stream applications. The system should dynamically allocate resource weights to operators, adjust the parallelism of operators based on changing requirements, and maintain system state consistency.

To address these requirements, we propose a reliable resource scaling framework that adapts to volatile data streams. The framework encompasses three main aspects. First, it can handle changes in data streams by dynamically adjusting the parallelism of operators, while minimizing the system latency and resource consumed by operators. Second, it can expand or contract the number of compute nodes deployed by the stream application, minimizing system resource overhead. Third, it can respond to changes in the parallelism of stateful operators in a stream application. This ensures the consistency and reliability of the system state.

In the future, we will further explore the following areas.

(1) Integrate load balancing among the instances of stateful operator into Es-Stream to further reduce system latency.
(2) Design probability models to support approximate computing results for specific scenarios.

REFERENCES

[1] Z. Wen, R. Yang, B. Qian, Y. Xuan, L. Lu, Z. Wang, H. Peng, J. Xu, A. Y. Zomaya, and R. Ranjan, "Janus: Latency-aware traffic scheduling for iot data streaming in edge environments," *IEEE Transactions on Services Computing*, vol. 16, no. 6, pp. 4302–4316, 2023.
[2] W. Li, D. Liu, K. Chen, K. Li, and H. Qi, "Hone: Mitigating stragglers in distributed stream processing with tuple scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 99, pp. 2021–2034, 2021.
[3] W. Chen, I. Paik, and P. C. K. Hung, "Transformation-based streaming workflow allocation on geo-distributed datacenters for streaming big data processing," *IEEE Transactions on Services Computing*, vol. 12, no. 4, pp. 654–668, 2019.
[4] F. Zhao, S. Li, B. B. Zhou, H. Jin, and L. T. Yang, "Hcache: A hash-based hybrid caching model for real-time streaming data analytics," *IEEE Transactions on Services Computing*, vol. 14, no. 5, pp. 1384–1396, 2021.
[5] H. Herodotou, L. Odysseos, Y. Chen, and J. Lu, "Automatic performance tuning for distributed data stream processing systems," in *2022 IEEE 38th International Conference on Data Engineering*, 2022, pp. 3194–3197.
[6] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 572–585, 2018.
[7] X. Liu, A. Dastjerdi, and R. Buya, "A stepwise auto-profiling method for performance optimization of streaming applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 12, no. 24, pp. 1–33, 2018.
[8] G. Russo Russo, V. Cardellini, and F. Lo Presti, "Hierarchical auto-scaling policies for data stream processing on heterogeneous resources," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 18, no. 4, pp. 1–44, 2023.
[9] 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, pp. 1–16, 2022.

[10] Z. She, Y. Mao, H. Xiang, X. Wang, and R. T. B. Ma, "Streamswitch: Fulfilling latency service-layer agreement for stateful streaming," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10.

[11] 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, pp. 1628–1642, 2019.

[12] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: Dynamic resource scheduling for real-time analytics over fast streams," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 411–420.

[13] W. Li, Z. Zhang, Y. Shu, H. Liu, and T. Liu., "Toward optimal operator parallelism for stream processing topology with limited buffers," *Journal of Supercomputing*, vol. 78, pp. 13 276–13 297, 2022.

[14] J. a. Francisco, M. E. Coimbra, P. F. R. Neto, F. Freitag, and L. Veiga, "Stateful adaptive streams with approximate computing and elastic scaling," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, 2023, p. 174–183.

[15] X. Wei, Y. Zhuang, H. Li, and Z. Liu, "Reliable stream data processing for elastic distributed stream processing systems," *Cluster Computing*, vol. 23, pp. 555–574, 2020.

[16] Aliyun, "tianchi," https://tianchi.aliyun.com/dataset/649?t=1679727494514.

[17] D. Sun, Y. Cui, M. Wu, S. Gao, and R. Buyya, "An energy efficient and runtime-aware framework for distributed stream computing systems," *Future Generation Computer Systems*, vol. 136, pp. 252–269, 2022.

[18] Y. Fang, J. Cui, and H. Zhong, "An efficient sdn load balancing scheme based on server response time," *Future Generation Computer Systems*, vol. 68, pp. 183–190, 2017.

[19] Z. Li, J. Yu, C. Bian, Y. Pu, Y. Wang, Y. Zhang, and B. Guo, "Flink-er: An elastic resource-scheduling strategy for processing fluctuating mobile stream data on flink," *Mobile Information Systems*, vol. 2020, pp. 1–17, 2020.

[20] J. Wang, Y. Zhang, and Z. G. Zhang, "Strategic joining in an m/m/k queue with asynchronous and synchronous multiple vacations," *Journal of the Operational Research Society*, vol. 72, pp. 161–179, 2021.

[21] Apache, "Evenscheduler," https://github.com/apache/storm/blob/v2.4.0/storm-server/src/main/java/org/apache/storm/scheduler/EvenScheduler.java.

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

[23] H. Xu, P. Liu, S. T. Ahmed, D. Da Silva, and L. Hu, "Adaptive fragment-based parallel state recovery for stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2464–2478, 2023.

[24] P. Osypanka and P. Nawrocki, "Qos-aware cloud resource prediction for computing services," *IEEE Transactions on Services Computing*, vol. 16, no. 02, pp. 1346–1357, 2023.

[25] M. Islam, H. Wu, S. Karunasekera, and R. Buyya, "Sla-based scheduling of spark jobs in hybrid cloud computing environments," *IEEE Transactions on Computers*, vol. 71, no. 05, pp. 1117–1132, 2022.

[26] X. Gao, R. Liu, and A. Kaushik, "Hierarchical multi-agent optimization for resource allocation in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 03, pp. 692–707, 2021.

[27] J. Xu and B. Palanisamy, "Model-based reinforcement learning for elastic stream processing in edge computing," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 292–301.

[28] H. Herodotou, Y. Chen, and J. Lu, "A survey on automatic parameter tuning for big data processing systems," *ACM Computing Surveys*, vol. 53, no. 2, pp. 1–37, 2020.

[29] I. Dharmadasa and F. Ullah, "Co-tuning of cloud infrastructure and distributed data processing platforms," in *2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 207–214.

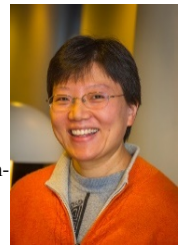
[30] B. Kahveci and B. Gedik, "Joker: Elastic stream processing with organic adaptation," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 205–223, 2020.

[31] Y. Zhuang, X. Wei, H. Li, M. Hou, and Y. Wang, "Reducing fault-tolerant overhead for distributed stream processing with approximate backup," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020, pp. 1–9.

[32] P. Liu, H. Xu, D. Da Silva, Q. Wang, S. T. Ahmed, and L. Hu, "Fp4s: Fragment-based parallel state recovery for stateful stream applications," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1102–1111.

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

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



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

Keqin Li is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, big data computing and distributed system etc. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, and the IEEE Transactions on Services Computing. He is an AAAS Fellow, an IEEE Fellow, and an AAIA Fellow. He is also a Member of Academia Europaea (Academician of the Academy of Europe).

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored over 750 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 168 with 149,400+ citations). He is among the world' s top 2 most influential scientists in distributed computing in terms of both singleyear impact and career-long impact based on a composite indicator of Scopus citation database.