Future Generation Computer Systems 128 (2022) 117–131

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

A multi-level collaborative framework for elastic stream computing systems

Dawei Sun [a,*], Shang Gao [b], Xunyun Liu [c], Rajkumar Buyya [d]

[a] *School of Information Engineering, China University of Geosciences, Beijing, 100083, China*
[b] *School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia*
[c] *Artificial Intelligence Research Center, National Innovation Institute of Defense Technology, Beijing, 100071, China*
[d] *Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia*

A R T I C L E I N F O

Article history:
Received 9 January 2021
Received in revised form 25 September 2021
Accepted 5 October 2021
Available online 12 October 2021

Keywords:
Multi-level framework
Stream computing
Elastic processing
Distributed system
Big data

A B S T R A C T

An elastic stream computing system is expected to process dynamic and volatile data streams with low latency and high throughput in timely manner. Effective management of stream application is considered one of the keys to achieve elastic computing by scaling in/out the workload of each computing node properly during runtime. Many existing work tried to build an elastic stream computing system from one perspective or at one level, which limited to some extent the system performance improvement. To address the problems brought by single level management, in this paper, we propose and implement a multi-level collaborative framework (called Mc-Stream) for elastic stream computing systems. This paper introduces our solution from the following aspects: (1) Extensive experiments show that system performance is affected by multiple factors locating at different levels. A multi-level collaborative optimization strategy can coordinate those factors and optimize the performance to a greater extent. (2) A system model is constructed to explain the multi-level collaborative framework, with the creation of topology model, data model and grouping model. The process of multi-level collaborative framework is formalized, including optimizing instances number, determining data stream load ratio among instances and deploying instances. (3) The system performance is optimized at multiple levels (user level, instance level, scheduling level, and resource level). It is further improved by the components of lightweight instances management, available resource-aware data stream redirection, fast and effective scheduling management, and asynchronous runtime redeployment without state loss. (4) Mc-Stream is implemented on top of Apache Storm platform. Metrics are evaluated with real-world stream applications, such as the fulfillment of system latency, throughput and resources utilization. Experimental results show the significant improvements made by Mc-Stream: reducing average system latency by 32%, increasing average system throughput by 26% and average resources utilization by 34%, compared with existing state-of-the-art scheduling strategies.

1. Introduction

Real-time data stream is an important data form in big data era. It exhibits features of high-velocity — tens of millions of data tuples generated per seconds, high-variability — data tuples coming from multiple independent data sources and each source generating data tuples in an unpredictable manner, high-unpredictability — each data tuple being asynchronous, affected by multiple factors from generating to processing, and high-valuable — the value of data stream itself trending to be less meaningful compared with the value of knowledge behind the data stream, which should be extracted freshly in a timely manner. Data parallelism is an essential feature of data stream processing. It calls for a new on-the-fly computing paradigm [1], where the architecture is designed from the perspective of "data as the center" rather than "computing as the center". In recent years, stream computing paradigm has become the de facto standard for processing continuous and unbounded data streams due to its distinctive features, such as low latency and high throughput, compared with batch computing paradigm. Many modern stream computing systems have been developed and deployed at scale, such as Google Millwheel [2], Google Dataflow [3], Apache

[*] Corresponding author.
E-mail addresses: sundaweicn@cugb.edu.cn (D. Sun), shang.gao@deakin.edu.au (S. Gao), xunyunliu@gmail.com (X. Liu), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2021.10.005
0167-739X/© 2021 Elsevier B.V. All rights reserved.

Storm [4], Twitter Heron [5] and Apache Samza [6]. Apache Storm is one of the most prominent open source software for distributed stream computing.

To process data streams promptly, an elastic stream computing system is always needed. It is expected to adjust resources allocated to stream applications at runtime, while bearing the objectives of high elasticity, low latency and high stability in mind. Stream application scheduling is one of the keys to make such elasticity possible by dynamically determining the relationships between vertices of stream applications and computing nodes in data center, and scaling in/out the workload of each computing node properly during runtime. However, this scheduling problem is also considered as one of the most thought-provoking NP-hard problems in general cases [7]. The real-time fluctuating data streams and complex vertex dependencies add more challenges to building such an elastic system.

To accomplish this goal, researchers have been trying to improve scheduling strategies in many different ways [7,8]. E.g., optimizing the deployment status of vertices on computing nodes, adjusting the number of computing nodes, the vertex instance number, or the distribution of data stream among multiple instances of a vertex. All those strategies help improve the system performance from one perspective or another.

Our work is motivated by the observation that unsatisfactory system performance is mainly caused by frequent online rescheduling. Each rescheduling state may optimize system performance. However, during the transition process from one state to another, the performance may deteriorate drastically. It is common that a new state has not been retained long, the computing environment such as data stream changes, requiring the state to be adjusted again. For each round of rescheduling, it is neither always necessary to optimize the performance to an optimal state, nor practical to simply adjust the state based on the changes of the environment. Moreover, the system performance is affected by multiple factors at different levels. Those factors are usually not independent of each other. By only optimizing the performance from one factor's perspective, the extent of improvement is limited, and sometimes even invalid. All these issues occur because resources are not fully utilized and multi-level comprehensive factors are not considered [9]. The fulfillment of the requirements raises challenges to achieving low system latency and effective resource utilization in big data stream computing environments.

As such, our aim is to address the aforementioned issues using a multi-level collaborative framework, which continuously optimizes the system performance using multiple factors at different levels. Each factor is first considered to improve the system performance accurately and quickly, then their collaboration is considered to maintain and further maximize the optimization. The new scheduling framework is expected to keep a relatively long-term online state and deal with fluctuating data streams while supporting features such as high system stability, low system latency and effective resource utilization.

1.1. Paper contributions

The key contributions of our work are as follows:

(1) Extensive experiments are conducted, showing that system performance is affected by multiple factors at different levels. A multi-level collaborative optimization strategy could help coordinate those factors from different levels and optimize system performance to a greater extent.

(2) A system model is constructed to explain the multi-level collaborative framework, with the creation of topology model, data model and grouping model. The process of multi-level collaborative framework is formalized, including optimizing instance number, determining data stream load ratio among instances and deploying instances.

(3) The system performance is optimized at multiple levels (e.g. user level, instance level, scheduling level, and resource level). It is further improved by the components of lightweight instance management, available resource-aware data stream redirection, fast and effective scheduling management and asynchronous runtime redeployment without state loss.

(4) A multi-level collaborative framework (Mc-Stream) is implemented on top of Apache Storm platform. Metrics are evaluated with real-world stream applications, such as the fulfillment of system latency, throughput and resources utilization. Experimental results show the significant improvements made by Mc-Stream: reducing average system latency by 32%, increasing average system throughput by 26% and average resources utilization by 34%, compared with existing state-of-the-art scheduling strategies.

1.2. Paper organization

The rest of the paper is organized as follows: Section 2 discusses the observed factors that affect system latency and resource utilization on current Storm platform and the motivations for our research work. Section 3 describes the system model, including the topology model, data model, and grouping model. Section 4 formalizes the process of multi-level collaborative framework, especially the parts for optimizing instances number, determining data stream load ratio among different instances and deploying instances. Section 5 focuses on the system architecture, instance management, redirection of data stream, scheduling management and resources management in Mc-Stream. Section 6 analyzes the performance evaluation results with metrics of average latency, average throughput and average resource utilization of a data center. Section 7 reviews the related work on application scheduling for elastic stream computing systems, as well as system performance optimization for distributed stream computing systems. Finally, conclusions and future work are discussed in Section 8.

2. Observations and motivations

To identify these most important factors affecting the system performance, we design a series of experiments on current Storm platform. In this section, the experiments are first discussed, followed by the motivations inspired by the observations.

2.1. Experimental environment

To observe the factors affecting system stability, latency and resource utilization on current Storm platform, we deploy and test the performance of Storm 1.2.2 [4] on top of CentOS 6.3. A monitor module is developed to monitor the Supervisors and Worker nodes. The cluster used in the experiments is provided by the school of Information Engineering, China University of Geosciences, Beijing. It consists of 35 computing nodes, being further divided into three types. Detailed configuration is shown in Table 1.

Table 1
Hardware configurations of computing nodes in the cluster.

Type	CPU	Memory	Bandwidth	External storage
1	Intel Core i3	4 GB RAM	1 Gbps LAN	1TB SSD
2	Intel Core i5	8 GB RAM	10 Gbps LAN	2TB SSD
3	Intel Core i7	16 GB RAM	100 Gbps LAN	4TB SSD

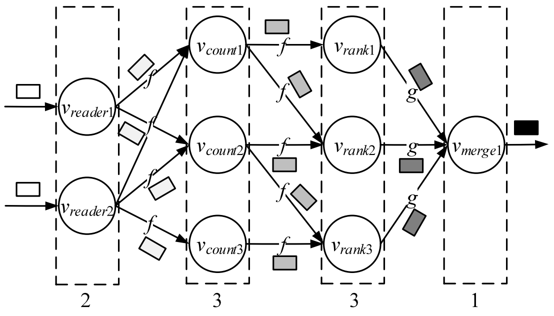


Fig. 1. The instance topology of Top_N.

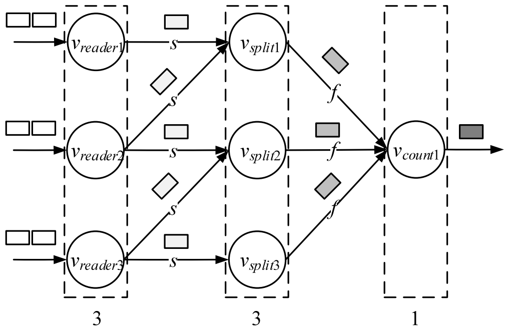


Fig. 2. The instance topology of WordCount.

In the cluster, one node runs Nimbus sub-system, two nodes run Zookeeper sub-system, and the rest 32 nodes run Supervisor sub-system.

Moreover, two applications Top_N and WordCount are used as the test cases, which are the most basic and commonly used streaming applications for performance testing and analysis. The instance topologies of Top_N and WordCount are shown as Fig. 1 and Fig. 2, respectively.

2.2. Observations

We use average load balancing (*ALB*) of data center (*DC*), average latency (*AL*) of streaming applications, and average resource utilization (*ARU*) of data center (*DC*) to evaluate the system load balancing, latency and resource utilization, respectively. The *ALB* of data center *DC* can be evaluated by calculating the standard deviation of the average CPU utilization of computing nodes in *DC* during time $[0, t]$. The *AL* and *ARU* can be evaluated by calculating the average latency of each streaming application and the average CPU utilization of computing nodes in *DC* during time $[0, t]$, respectively. All the evaluations are done at 4 levels: user level, instance level, scheduling level and resource level.

(1) User level

At the user level, user constructs a logical graph for a streaming application according to its function. When applications are submitted to Storm platform for processing, the total number of applications is one of the key factors affecting its overall performance, especially the metrics such as the load balance of data center, latency of streaming applications and resource utilization of data center. The determination of the critical states becomes more important for the timing of optimization.

When the input rate of data stream is 1000 tuples/s, with the increase of application number, the average load balancing *ALB* of data center *DC*, average latency *AL* of applications, and average resource utilization *ARU* of *DC* are increasing accordingly. As shown in Fig. 3, when the application number for WordCount is less than

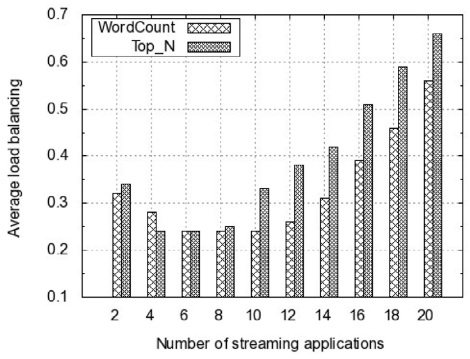


Fig. 3. Average load balancing (*ALB*) of data center (*DC*) with different number of applications.

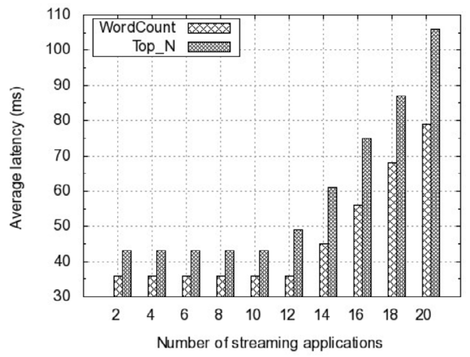


Fig. 4. Average latency (*AL*) of streaming applications with different number of applications.

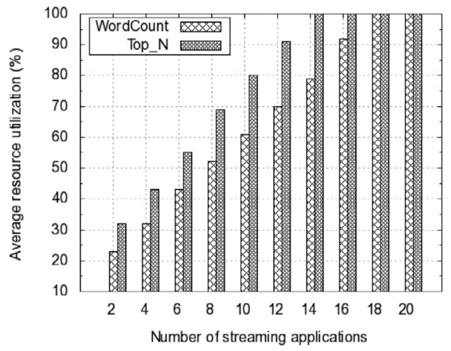


Fig. 5. Average resource utilization (*ARU*) of data center (*DC*) with different number of applications.

12, the average load balancing is less than 0.3. However, when the number is greater than 12, the average load balancing is greater than 0.3, worsening the load balance of *DC*. A similar situation also occurs for Top_N. With the increase of its application number (>10), the average load balancing is greater than 0.3.

As shown in Fig. 4, when the application number for WordCount is less than 12, the average latency is stabilized at a low level, which is 36 ms. When the number is greater than 12, the average latency continues increasing. A similar situation occurs for Top_N, where the average latency increases dramatically when the application number is greater than 10.

As shown in Fig. 5, when the application number for WordCount is less than 16, the average resource utilization of *DC* is less than 100%. When the number is greater than 16, the average resource utilization is 100%. The same trend applies to Top_N,

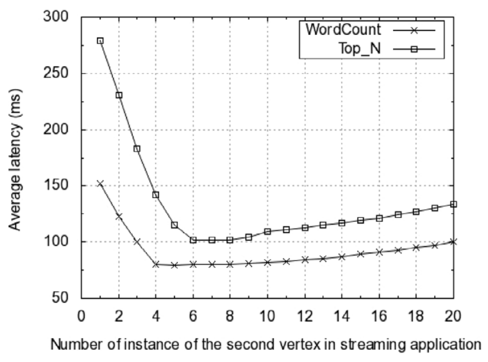


Fig. 6. Average latency (*AL*) of application with different instance numbers under a stable input rate (2000 tuples/s).

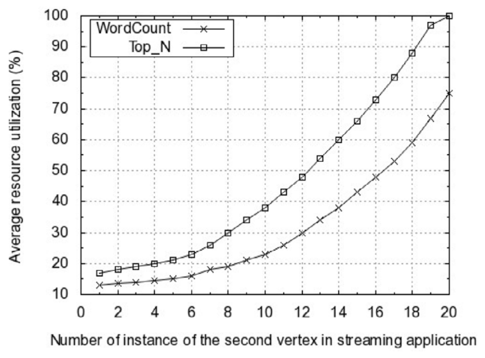


Fig. 7. Average resource utilization (*ARU*) of data center (*DC*) with different instance numbers under a stable input rate (2000 tuples/s).

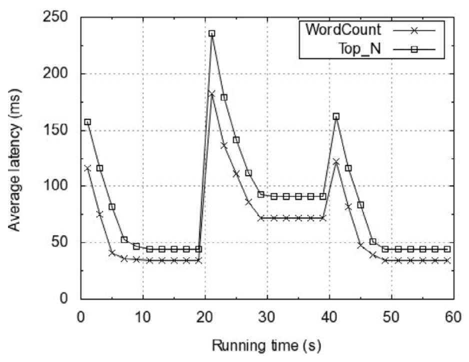


Fig. 8. Average latency (*AL*) of application under different input rates.

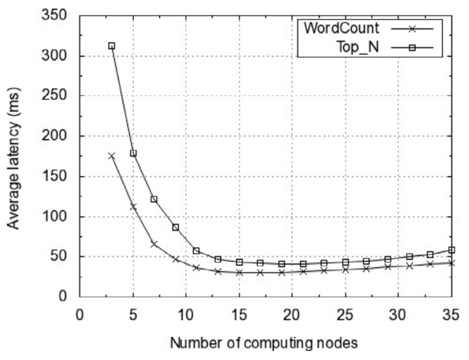


Fig. 9. Average latency (*AL*) of application with different amounts of resources.

where the average resource utilization reaches 100% when the application number is greater than 12.

(2) Instance level

At the instance level, the number of instances for each vertex in an application topology is set by users based on experience. For a static computing environment, it is important to set and keep a reasonable instance number for each vertex. Too few instances will overload each instance and affect the system performance. But too many instances will make the instances themselves a burden to the system and each instance cannot be fully utilized. For a dynamic computing environment, the timing and fitting of an adjustment to the instance number for each vertex impact the system performance.

When the input rate is stabilized at 2000 tuples/s, with the increase of instance number of vertex v_{count} in Top_N and v_{split} in WordCount, the best performance of average latency *AL* of application and average resource utilization *ARU* of data center *DC* can always be achieved given a suitable instance number. As shown in Fig. 6, within range [0, 4], when the instance number of v_{split} in WordCount increases, the average latency of WordCount is constantly decreasing. When the instance number is greater than 8, the average latency of WordCount climbs slowly. A similar situation occurs to the instance number of v_{count} in Top_N.

As shown in Fig. 7, with the increase of instance number of v_{split} in WordCount, the average resource utilization of *DC* is constantly increasing. When the number is less than 4, the increase of average resource utilization remains slow. After that, it climbs dramatically.

(3) Scheduling level

At the scheduling level, all instances of a topology are scheduled to computing nodes in a data center according to a chosen scheduling strategy. Different strategies may significantly affect the system performance. More specifically, in a stream computing environment, once the topology of an application is submitted, it will run forever unless being forcibly terminated. During the runtime, the data stream may change, so does the load of computing nodes. This unique characteristic requires the scheduling strategy further adjusts the scheduling state on the fly. This kind of adjustment should be done incrementally without triggering large system performance fluctuation while considering the existing scheduling state of stream applications.

As shown in Fig. 8, in time range [0, 19], [20, 39], and [40, 60], the input rates are stabilizes at 1000 tuples/s, 2000 tuples/s, and 1000 tuples/s, respectively. At time 20 s, the rate increases from 1000 tuples/s to 2000 tuples/s. At time 40 s, it decreases from 2000 tuples/s to 1000 tuples/s. Within each range, the average latency is stabilized at a certain level. When the rate changes, the latency fluctuates for a relatively long time due to the instance rescheduling in the instance topology. The fluctuation degree is closely related to the instance number to be rescheduled. The fewer the rescheduled instance number, the less fluctuation of the average latency.

(4) Resource level

At the resource level, the number of resources and the choice of appropriate resources have a significant impact on the system performance. Generally speaking, the more resources, the better performance. But this rule is not always valid in a distributed stream computing environment. The choice of appropriate resources also plays an important role.

As shown in Fig. 9, two WordCount streaming applications and two Top_N streaming applications are submitted to the data center. At the beginning, with the continuous increase of computing nodes, the average latency is decreasing accordingly. However, when the average latency reaches a certain level (minimum), increasing computing nodes does not further improve the average latency. Instead, it increases the latency slightly. This is caused by the increase of network delay that constitutes the response time in a distributed environment.

2.3. Motivations

Though the two experimental cases WordCount and Top_N are not complex streaming applications, the results are still representative. They verify that the system performance is affected by multiple factors at different levels and these factors are not independent of each other. The extent of performance improvement might be limited if only considering one factor or from one level. To optimize the performance to a greater extent, a multi-level collaborative optimization strategy may help. The motivations we get from the above analysis are summarized as below:

(1) Given the system stability, resource utilization and other issues are interdependent, how to balance these metrics and consider them in a comprehensive way when targeting low latency and high throughput?

(2) How to emphasize the influence of one specific factor at a specific level when they usually further cross-influence and restrict each other?

(3) How to optimize the system performance from multiple perspectives? One factor based optimization only improves performance to a certain degree, which limits the extent of improvement.

(4) The system performance is extremely sensitive to continuous fluctuation of data stream. Is it possible to find an effective multi-level collaborative optimization strategy helping coordinate those factors from different levels and dimensions? If yes, how should it perform compared to the existing system provided strategies?

3. System model

Motived by the above findings, we consider collaborative framework from multiple levels. Before proposing concrete algorithms, we first formalize the system model to precisely describe the multi-level scheduling problem in an elastic stream computing system. This includes the definitions of topology model, data model and grouping model.

3.1. Topology model

The function of a streaming application [10,11] is usually represented in a logical topology by users. The logical topology (LT) can be viewed as a directed acyclic logical graph $G_{LT} = (V(G_{LT}), E(G_{LT}))$, where $V(G_{LT}) = \{v_i | i \in 1, \ldots, n\}$ is a finite set of n vertices. All the vertices work together to deliver outputs. The vertex function $fun(v)$ of each vertex v is completely different, that is if $\forall v_i, v_j \in V(G_{LT})$, then $\nexists fun(v_i) = fun(v_j)$. $w(v_i)$ is the weight of vertex v_i in fulfilling the whole function, which can be evaluated by the time and space complexity. $E(G_{LT}) = \{e_{v_i,v_j} | v_i, v_j \in V(G_{LT})\}$ is a finite set of directed edges, and $e_{v_i,v_j} \in E(G_{LT})$ indicates that there is a data stream flowing from vertex v_i to v_j, where v_i and v_j are the upstream and downstream vertex of e_{v_i,v_j}, respectively. $w(e_{v_i,v_j})$ is the weight of e_{v_i,v_j}. It represents the traffic load of e_{v_i,v_j} in transferring data tuples from vertex v_i to v_j, which can be evaluated by the number of data tuples per unit time.

As shown in Fig. 10, the logical topology of Top_N consists of four vertices, and $V(G_{LT(Top\_N)}) = \{v_{reader}, v_{count}, v_{rank}, v_{merge}\}$. The functions of $v_{reader}, v_{count}, v_{rank}$ and v_{merge} are "reading data tuples", "counting key words", "ranking key words", and "ranking key words by count", respectively. There are five directed edges, three of which transfer data tuples from upstream and downstream; the first special edge, input edge, is responsible

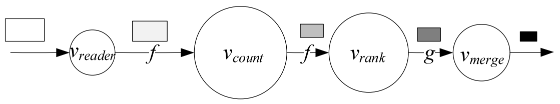
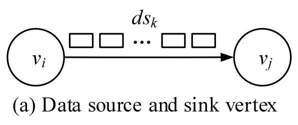
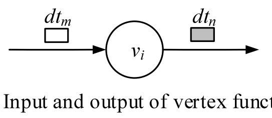
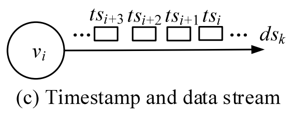


Fig. 10. The logical topology of Top_N.







ts_{i+3} ts_{i+2} ts_{i+1} ts_i
ds_k
v_i
(c) Timestamp and data stream

Fig. 11. Data stream and data tuple.

for reading data from data source; the last special edge, output edge, is responsible for generating the final processing result. The size of each vertex indicates the weight of corresponding vertex, and the size of data stream on each edge indicates the weight of corresponding edge. (For simplicity, the weights of vertex and edge in the following figures are not explicitly depicted).

The logical topology at runtime is usually viewed as an instance topology (IT). It is a directed acyclic instance graph $G_{IT} = (V(G_{IT}), E(G_{IT}))$, and $V(G_{LT}) \subseteq V(G_{IT})$, $E(G_{LT}) \subseteq E(G_{IT})$. If $\forall v_i \in V(G_{LT})$, then $\exists k \in \{1, 2, \ldots, m\}$, $v_{i1}, \ldots, v_{ik}, \ldots, v_{im}$ are instances of v_i, and $\{v_{i1}, v_{i2}, \ldots, v_{im}\} \subset V(G_{IT})$. The functions of m instances of v_i are the same, that is $fun(v_{i1}) = fun(v_{i2}) = \cdots = fun(v_{im})$. The weights of m instances of v_i are also the same, that is $w(v_{i1}) = w(v_{i2}) = \cdots = w(v_{im})$.

An instance topology of Top_N is shown as Fig. 1, where v_{count} has three instances v_{count1}, v_{count2}, and v_{count3}. Their functions and weights of v_{count} are the same, that is, $fun(v_{count1}) = fun(v_{count2}) = fun(v_{count3})$, and $w(v_{count1}) = w(v_{count2}) = w(v_{count3})$.

3.2. Data model

A data stream $ds = \{dt_1, dt_2, \ldots, dt_i, \ldots\}$ is a time series of data tuples [12,13], which originates from a data source, and ends in a data sink.

As shown in Fig. 11(a), data stream ds_k flows from data source v_i to data sink v_j. A data tuple $dt_i = (key_i, value_i, ts_i)$ is described by key_i, $value_i$ and ts_i, representing the key, value and timestamp of dt_i, respectively. As shown in Fig. 11(b), data tuple dt_m flows in v_i and data tuple dt_n flows out of v_i. The relationship between data tuple dt_m and dt_n can be described as (1).

$$dt_n = f_{v_i}(dt_m), \tag{1}$$

where $f_{v_i}()$ is the function of vertex v_i.

The timestamp [14] of a data tuple is assigned by a vertex, which produces that data tuple. Each vertex assigns a timestamp to a data tuple independently. All data tuples produced by the same vertex form a data stream, ordered by the timestamps, each of which is unique in its data stream. As shown in Fig. 11(c), the timestamp of each data tuple is assigned by vertex v_i. All data tuples form a stream ds_k.

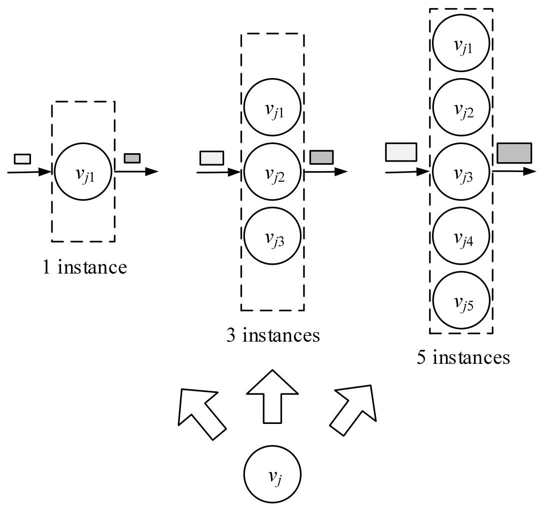


Fig. 12. Different instance sets of vertex v_j.

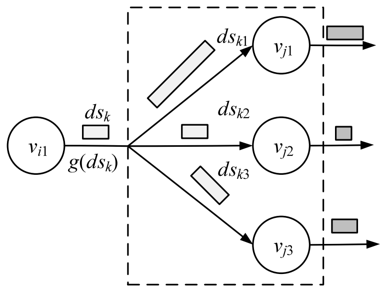


Fig. 13. Partitioning data stream among instances of v_j.

3.3. Grouping model

In an instance topology G_{IT}, it is critical to dynamically determine an appropriate instance number for each vertex for performance purposes [15,16]. For vertex v_j, different sets of instances can be created, each of which has different number of instances. Each instance fulfills the same function but different sets provide different processing capabilities. These instance sets of v_j can be elastically replaced with each other to adapt the dynamic data streams.

As shown in Fig. 12, in order to meet different processing requirements, e.g. low, medium, and high speed input, v_j can adjust the instance number accordingly by selecting one instance set out of the three.

For vertex v_j, if the instance number is greater than one, the input data stream needs to be partitioned among all the instances of v_j. The partitioning method of data stream between the instance sets of two directly connected vertices is called grouping strategy. As shown in Fig. 13, according to the grouping strategy $g()$ between vertex v_i and v_j, the data stream ds_k is partitioned into 3 sub-data streams ds_{k1}, ds_{k2} and ds_{k3} for v_{j1}, v_{j2} and v_{j3}, respectively.

4. Problem statement

Based on the system models defined, in this section, we formalize the problems of optimizing the instance number for each vertex, determining the data stream load ratio among the vertex instances, and deploying the instance topology to computing nodes in data center, which are located at the instance level, data stream level and scheduling level, respectively.

4.1. Optimizing instance number

At the instance level, one or more instances will be created for each vertex of a logical topology to improve the processing capacity. To adapt to the changing data stream and utilize the resources well, the instance number for each vertex needs to be adjusted from time to time. For a vertex $v_i \in V(G_{LT})$ in a logical topology G_{LT}, if $v_{i1}, v_{i2}, \ldots, v_{im_{v_i}}$ are the instances of v_i, the optimization problem at this instance level converts to finding the best fit number m_{v_i} for v_i, which helps maximize the throughput and minimize the latency of v_i.

$$\max\left(t\left(v_i\right)\right) \textit{ and } \min\left(l\left(v_i\right)\right), \tag{2}$$

subject to

$$1 \le m_{v_i} \le m_{\max},\ v_i \in G_{LT}, \tag{3}$$

where $t(v_i)$ and $l(v_i)$ are the throughput and latency of v_i, respectively. $m_{\max}$ is the system-specified SLAs (Service Level Agreements) [17] constraint.

4.2. Optimizing data stream load ratio among instances

At the data stream level, it is necessary to constantly monitor available resources and optimize data stream partition [18] among multiple instances. A good data stream grouping strategy can often well achieve load balancing among vertex instances, helping maximize the throughput and minimize the latency of G_{IT}. For a vertex $v_i \in V(G_{LT})$, if $v_{i1}, v_{i2}, \ldots, v_{im_{v_i}}$ are the instances of v_i, the load balancing deviation of all the m_{v_i} instances of v_i can be evaluated by (4).

$$lbd_{v_i} = \frac{1}{m_{v_i}} \sum_{k=1}^{m_{v_i}} \left| len_{v_{ik}} - \overline{len_{v_i}} \right|, \tag{4}$$

where $len_{v_{ik}}$ is the length of data tuple queue on the kth instance of v_i, and $\overline{len_{v_i}}$ is the average length of data tuple queue on all instances of v_i.

The data stream grouping optimization problem for m_{v_i} instances of v_i can be formalized as follows:

$$\min\left(lbd_{v_i}\right), \tag{5}$$

subject to

$$1 \le m_{v_i} \le m_{\max},\ v_i \in G_{LT} \tag{6}$$

4.3. Optimizing deployment of instance topology to computing nodes

At the scheduling level, it is necessary to constantly monitor the state of each instance on computing nodes and optimize the deployment according to the changing data stream and available resources. It is expected that both the stability and adaptability are being maintained, while trying to maximize the throughput and minimize the latency of G_{IT}. For an instance topology G_{IT}, $V(G_{IT})$ and $E(G_{IT})$ are finite set of vertices and directed edges. For a data center DC with n computing nodes $cn_1, cn_2, \ldots, cn_n$, the instance deployment optimization problem for all vertices in $V(G_{IT})$ can be formalized as follows:

$$\max\left(t\left(G_{IT}\right)\right) \textit{ and } \min\left(l\left(G_{IT}\right)\right), \tag{7}$$

subject to

$$\Delta tm \le tm_{\min}, \tag{8}$$

where $t(G_{IT})$ and $l(G_{IT})$ are the throughput and latency of G_{IT}. Δtm is the time interval to redeploy part or all the instances online from the current state to the next new state. $tm_{\min}$ is a user-specified SLAs constraint.

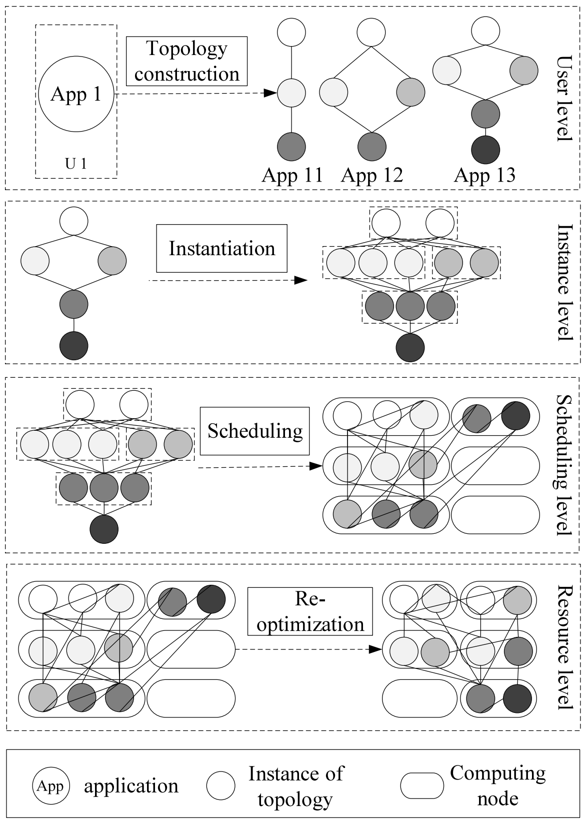


Fig. 14. Mc-Stream architecture.

5. Mc-Stream: Architecture and algorithms

Based on the above experimental and theoretical analysis, a multi-level collaborative framework, Mc-Stream, is proposed on top of the Storm platform. To provide an overview of the framework, this section discusses its system architecture and algorithms used for instance management, data stream redirection, scheduling management and resource management.

In Storm platform, once an application is submitted, the platform instantiates its logical topology and deploys the topology to computing nodes. If the system performance is not ideal, our multi-level collaborative framework will step in to optimize the initial settings for performance improvement. For instance, the instance number of each vertex and the data stream grouping strategy among instances are to be optimized at the instance level. This specific grouping strategy can be customized by implementing the CustomStreamGrouping interface on Storm [4]. The scheduling state of instances on each computing node is to be optimized at the scheduling level. The utilization of resources is to be optimized at the resource level.

5.1. System architecture

The system architecture of Mc-Stream includes four levels: user level, instance level, scheduling level and resource level, as shown in Fig. 14.

At the user level, user subdivides the application function, clarifying the internal logic, data dependencies and indicating the data stream grouping strategy between the upstream and downstream vertices. This logical topology is built via the Spout and Bolt interface provided by the Storm platform. Generally, the logical topologies constructed by different users for the same application might be different due to different function subdivision. It is difficult to make all the constructed logical topologies optimal. Instead, it is more practical to improve users' understanding of the application function and implementation logic to build a reasonable logical topology. Once the logical topology is submitted to the Storm platform, it can be further optimized through instantiation.

At the instance level, each vertex in the logical topology is instantiated to one or more instances. All the instances form an instance topology of the logical one. For a vertex, its instance number is the most critical parameter to determine in the instantiation process [19]. A reasonable instance number can significantly improve the throughput and latency of the instance topology, otherwise it may increase the burden of the system and affect the performance negatively. In a general sense, the instance number of a vertex is determined by factors such as the vertex function, the available computing resources and the data stream ratio, etc. Choosing a right number will enable the vertex to process the data stream effectively and reduce the risk of introducing a bottleneck in the instance topology.

At the scheduling level, each instance in the instance topology is deployed to a computing node in the data center according to a scheduling strategy. When the resources are limited or the data stream fluctuates, the scheduling strategy aims for meeting the user's requirements for throughput and latency; when the resources are sufficient, it aim for best utilizing the resources or trying to optimize the throughput and latency. In general, the scheduling strategy needs to consider factors such as instance topology, available resources in data center and rate of data stream, etc. [20]. On Storm platform, a scheduling strategy can be customized by implementing the IScheduler interface and specified in the configuration file Storm.yaml [4].

At the resource level, it is necessary to ensure the efficient utilization of resources and maintain the system stability. The realization of these objectives requires the multi-level coordination [21].

5.2. Instance management

While the Mc-Stream optimizing the initial scheduling, at the instance level, the instance number for each vertex in the logical topology is determined by the computational complexity of the vertex function, the processing capacity of the computing node and the rate of data stream.

For a vertex $v_i \in V(G_{LT})$, c_{v_i} and m_{v_i} are the computational complexity and instance number of vertex v_i, respectively. p_{v_i} is the processing capacity of the computing node that vertex v_i is running on. In order to set a reasonable instance number m_{v_i} and balance computing load, each vertex in $V(G_{LT})$ needs to meet the condition that the amount of calculation in one unit processing capacity is proportional to the instance number, as specified by (9).

$$\frac{c_{v_1}}{p_{v_1}}:m_{v_1} = \frac{c_{v_2}}{p_{v_2}}:m_{v_2} = \cdots = \frac{c_{v_n}}{p_{v_n}}:m_{v_n}. \tag{9}$$

Under a data stream input ratio r_{v_1}, for the first vertex v_1 in logical topology G_{LT}, its ideal instance number m_{v_1} can be determined by constantly increasing the instance number until the performance is no longer improved. E.g. throughput no longer increases and latency no longer decreases. Once the instance number m_{v_1} of v_1 under ratio r_{v_1} is determined, the instance number for the rest vertices can be determined according to (9).

The algorithm for instance management is described in Algorithm 1.

Algorithm 1: Instance management.

```
1.  Input: logical topology $G_{LT}$, processing capacity of computing node in DC, data stream input ratio $r_{v_1}$.
2.  Output: instance number of each vertex in logical topology $G_{LT}$ under ratio $r_{v_1}$.
3.  if the ratio $r_{v_1}$ is 0 then
4.      return null.
5.  end if
6.  for each vertex $v_i$ in logical topology $G_{LT}$ do
7.      Set the maximum throughput $\max(t(v_i))$ and minimum latency $\min(l(v_i))$ for each vertex $v_i$.
8.      Get the computational complexity $c_{v_i}$ for each vertex $v_i$ via code analysis on the execute() method.
9.  end for
10. for each computing node $cn_i$ in data center DC do
11.     Get the processing capacity $p_{cn_i}$ of node $cn_i$.
12.     Set the processing capacity $p_{v_i}$ of each vertex running on node $cn_i$ to $p_{cn_i}$, (i.e. $p_{v_i} = p_{cn_i}$).
13. end for
14. Set the instance number $m_{v_1}$ for input vertex $v_1$ to 1.
15. Initialize the optimization step size $\Delta m_{v_1}$ for input vertex $v_1$.
16. while throughput $t(v_1)$ or latency $l(v_1)$ of $v_1$ can be further improved do
17.     Update $m_{v_1} = m_{v_1} + \Delta m_{v_1}$.
18.     Adjust the optimization step size $\Delta m_{v_1}$.
19. end while
20. for the rest vertices in logical topology $G_{LT}$ do
21.     Calculate instance number $m_{v_i}$ according to (9).
22.     Update $m_{v_i}$ to be the nearest rounding up integer of $m_{v_i}$.
23. end for
24. return the instance number for each vertex in logical topology $G_{LT}$ under input ratio $r_{v_1}$.
```

The input of this algorithm includes logical topology G_{LT}, processing capacity of each computing node in data center DC and data stream input ratio r_{v_1}. The output is the instance number for each vertex in G_{LT} under ratio r_{v_1}. Step 6 to step 9 set the maximum throughput $\max(t(v_i))$ and minimum latency $\min(l(v_i))$, and get the computational complexity c_{v_i} for each vertex v_i. Step 10 to step 13 set the processing capacity p_{v_i} of each vertex under the current scheduling state. Step 16 to step 19 get the instance number m_{v_1} for input vertex v_1 under ratio r_{v_1} by constantly increasing the instance number until the throughput $t(v_1)$ and latency $l(v_1)$ are no longer improved. Step 20 to step 23 calculate and optimize the instance number m_{v_i} for the rest vertices in G_{LT} according to (9), ensuring that the processing capacity is proportional to the number of instances.

5.3. Data stream redirection

At the instance level, unbalanced data stream distribution among vertex instances may make some overloaded instances become system bottlenecks, while the others are under-loaded or even idle, compromising resources efficiency and system performance [22]. It is important to balance the load among these instances and optimize the local performance at the granularity of instance, helping improve the global performance of the instance topology.

For a vertex $v_i \in V(G_{LT})$, if $v_{i1}, v_{i2}, \ldots, v_{im_{v_i}}$ are instances of v_i in instance topology G_{IT}, data tuples flow from upstream vertex/vertices to downstream vertex v_i, to balance load among all m_{v_i} instances of v_i, the data tuples will be grouped according to the length of current idle queue of each instance. (Usually, a stream computing system maintains a local queue to buffer data tuples for each vertex.)

For the kth instance v_{ik}, its probability of being assigned a data tuple can be calculated by (10).

$$pb_{v_{ik}} = \frac{l_{v_{ik}} - len_{v_{ik}}}{\sum_{l=1}^{m_{v_i}} \left(l_{v_{il}} - len_{v_{il}}\right)} \quad (10)$$

where $l_{v_{ik}}$ is the queue length on the kth instance of v_i, and $len_{v_{ik}}$ is the queue occupancy length by data tuples on the kth instances.

The algorithm for data stream redirection is described in Algorithm 2.

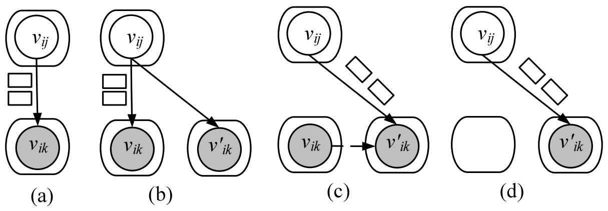


Fig. 15. Processing instance online scheduling.

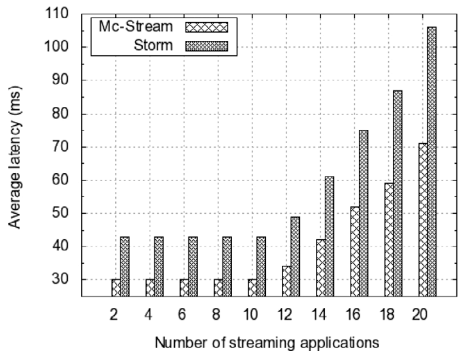


Fig. 16. Average latency (*AL*) of Top_N with different number of streaming applications.

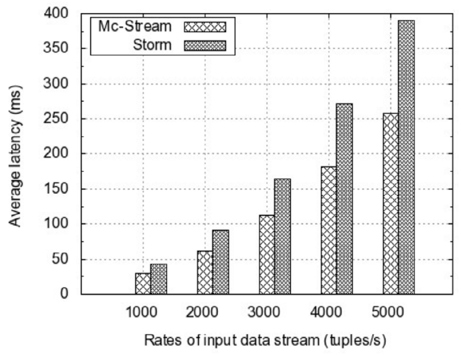


Fig. 17. Average latency (*AL*) of Top_N under different input rates.

Algorithm 2: Data stream redirection.

```
1.  Input: instance topology $G_{IT}$, queue state of each computing node in data center DC.
2.  Output: data stream grouping scheme for all the vertex instances in instance topology $G_{IT}$.
3.  for each instance $v_{ik}$ in $G_{IT}$ do
4.      Set the queue length $l_{v_{ik}}$ on the kth instance of $v_i$ according to the queue state of $cn_i$ where $v_{ik}$ is running.
5.      Set the queue occupancy length $len_{v_{ik}}$ by data tuples on the kth instance according to $cn_i$.
6.  end for
7.  for each vertex $v_i$ in instance topology $G_{IT}$ do
8.      Set the maximum average queue occupancy length $\max\left(\overline{len}\right)$ on all the $m_{v_1}$ instances of $v_i$.
9.      Set the minimum average queue occupancy length $\min\left(\overline{len}\right)$ on all the $m_{v_1}$ instances of $v_i$.
10.     Set the maximum load balancing deviation $\max\left(lbd_{v_i}\right)$ for each vertex $v_i$.
11. end for
12. for each vertex $v_i$ in logical topology $G_{LT}$ do
13.     Calculate the average queue occupancy length $\overline{len_{v_i}}$ for all the $m_{v_1}$ instances of $v_i$ instance topology $G_{IT}$.
14.     if $\overline{len_{v_i}} > \max\left(\overline{len}\right)$ or $\overline{len_{v_i}} < \min\left(\overline{len}\right)$ then
15.         Update the instance number of $v_i$ by Algorithm 1.
16.     end if
17. end for
18. for each vertex $v_i$ in instance topology $G_{IT}$ do
19.     if load balancing deviation $lbd_{v_i}$ of vertex $v_i$ is greater than the maximum load balancing deviation $\max\left(lbd_{v_i}\right)$ then
20.         for each instance of $v_i$ do
21.             Update the data tuple assigning probability $pb_{v_{ik}}$ of the kth instance $v_{ik}$ by (10).
22.         end for
23.     end if
24. end for
25. return data stream grouping scheme for all the vertex instances in instance topology $G_{IT}$.
```

The input of this algorithm includes instance topology G_{IT}, queue state of each computing node in data center DC. The output is data stream grouping scheme for vertex instances in instance topology G_{IT}. Step 3 to step 6 set the total queue length $l_{v_{ik}}$ and queue occupancy length $len_{v_{ik}}$ on the kth instances in G_{IT}. Step 7 to step 11 set the maximum average queue length $\max\left(\overline{len}\right)$ and the minimum average queue occupancy length $\min\left(\overline{len}\right)$ on all the m_{v_1} instances of v_i, and set the maximum load balancing deviation $\max\left(lbd_{v_i}\right)$ for each vertex v_i, where $\max\left(\overline{len}\right)$ is to prevent all instances from being overloaded, $\min\left(\overline{len}\right)$ is to prevent all instances from being underloaded. $\max\left(lbd_{v_i}\right)$ is to keep all instances in a reasonable load balancing state. Step 12 to step 17 update the instance number of each v_i in $G_{LT}$by Algorithm 1. Step 18 to step 24 update the data tuple assigning probability $pb_{v_{ik}}$ of the kth instance v_{ik} by (10), and get the data stream grouping scheme for all the instances in instance topology G_{IT}.

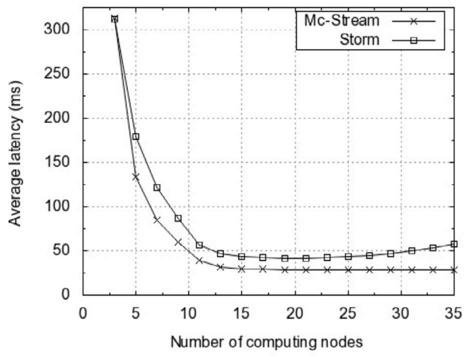


Fig. 18. Average latency (AL) of Top_N with different number of computing nodes.

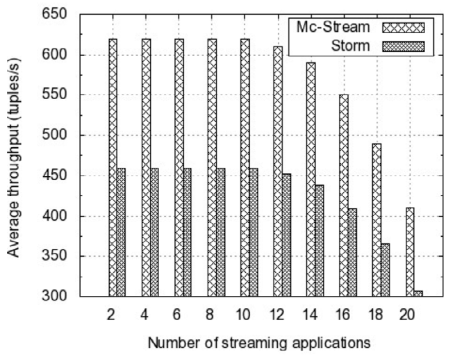


Fig. 19. Average throughput (AT) of Top_N with different number of streaming applications.

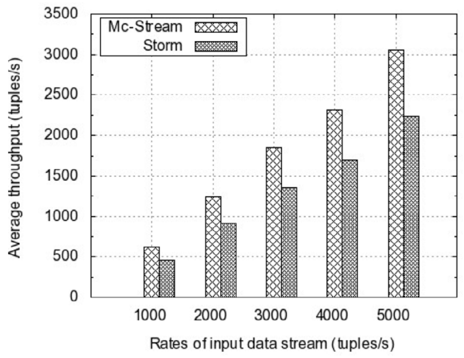


Fig. 20. Average throughput (AT) of Top_N under different input rates.

5.4. Scheduling management

At the scheduling level, instances in the instance topology need to be properly deployed to available computing nodes. This deployment should not only meet the requirements for throughput and latency imposed by users, but also ensure efficient utilization of resources in data center. In an elastic stream computing environment, a theoretical optimal schedule might not always be practically optimal due to continuously changing of data stream and available resources [23,24]. Therefore, instead of spending too much time on searching for a theoretical optimal schedule at the scheduling level, we use much less time to find a scheduling scheme to only meet users' SLOs (Service Level Objectives).

The algorithm for scheduling instance topology is described in Algorithm 3.

Algorithm 3: Scheduling instance topology.

```
1.  Input: instance topology $G_{IT}$, processing capacity of computing node in data center DC, data stream input ratio $r_{v_1}$.
2.  Output: instance scheduling scheme for $G_{IT}$ on computing nodes under ratio $r_{v_1}$.
3.  for each instance in instance topology $G_{IT}$ do
4.      Get the weight $w(v_i)$ share of each instance.
5.  end for
6.  for each directed edge in instance topology $G_{IT}$ do
7.      Get the weight $w(e_{v_i,v_j})$ of each directed edge.
8.  end for
9.  for each computing node $cn_i$ in data center DC do
10.     Get the available processing capacity $p_{cn_i}$ of computing node $cn_i$.
11. end for
12. Set the maximum throughput $\max(t(G_{IT}))$ and the minimum latency $\min(l(G_{IT}))$.
13. while throughput $t(G_{IT}) < \max(t(G_{IT}))$ or latency $l(G_{IT}) > \min(l(G_{IT}))$ do
14.     Sort all instances in $G_{IT}$ in a topological order and save them in the un-scheduled instance queue.
15.     Sort all computing nodes by the amount of available resources in a descending order and save them in the computing node queue.
16.     for each instance in the un-scheduled instance queue do
17.         Fetch an instance $v_{ik}$ from the un-scheduled instance queue.
18.         if a computing node $cn_j$ in the computing node queue has resources to satisfy the requirement of $v_{ik}$ and there has at least one upstream instance of $v_{ik}$ and no any other instances of $v_i$ running on $cn_j$ then
19.             Schedule instance $v_{ik}$ to computing node $cn_j$.
20.         else if there are instances of $v_i$ scheduled on other computing nodes then
21.             Schedule instance $v_{ik}$ to a computing node which has resources to satisfy $v_{ik}$ and does not run any other instances of $v_i$.
22.         else
23.             Schedule instance $v_{ik}$ to the computing node with the most available resources.
24.         end if
25.         Update the available resources of affected computing nodes in data center DC
26.         Resort the computing node queue.
27.     end for
28.     Update the throughput $t(G_{IT})$ and latency $l(G_{IT})$ of instance topology $G_{IT}$
29. end while
30. return instance scheduling scheme for $G_{IT}$ on computing nodes in data center DC under ratio $r_{v_1}$.
```

The input of this algorithm includes instance topology G_{IT}, processing capacity of each computing node in data center DC and data stream input ratio r_{v_1}. The output is an instance scheduling scheme for G_{IT} on computing nodes under ratio r_{v_1}. Step 3 to step 5 get the weight $w(v_i)$ share of each instance, where $w(v_i)$ is the computational complexity c_{v_i} obtained via code analysis. Step 6 to step 8 get the weight $w\left(e_{v_i,v_j}\right)$ of each directed edge via analysis on the size of data tuple volume per unit time from v_i to v_j at the instance level. Step 9 to step 11 get the available processing capacity p_{cn_i} of computing node cn_i. Step 12 sets the maximum throughput $\max(t(G_{IT}))$ and the minimum latency $\min(l(G_{IT}))$. Step 13 to step 29 schedule instances of G_{IT} to computing nodes. In this process, the following principles are applied: (1) instances that have data stream dependencies should be allocated to the same computing node as much as possible should resources allow; (2) different instances of the same vertex are scheduled to different computing nodes; (3) an instance should be deployed to a computing node with the most available resources, providing a good basis for potential future rescheduling. In addition, step 14 and 15 ensure that the instances are in a topological order and the computing nodes are in a descending order, which further improves the efficiency of instance scheduling.

5.5. Resource management

At the resource level, when the data stream input or the amount of available resources changes, it is necessary to adjust the deployment of instances to the computing nodes on the fly. This adjustment should be done in a timely manner to sustain a high-performance system and efficient utilization of resources [25].

For an instance v_{ik} in instance topology G_{IT}, if its weight $w(v_{ik})$ in current scheduling state is the highest, then instance v_{ik} is considered as a hotspot [26] instance. Similarly, the directed edge e_{v_i,v_j} with the highest weight $w\left(e_{v_i,v_j}\right)$ is considered as a hotspot edge. The instance v_{ik} with the lowest weight $w(v_{ik})$ is considered as a coldspot instance, and the directed edge e_{v_i,v_j} with the lowest weight $w\left(e_{v_i,v_j}\right)$ is considered as a coldspot edge. In the process of improving throughput and latency, the hotspot instances and edges can be the focus of optimization, while in cases for improving resource usage rate, the coldspot instances and edges can be the candidates.

To optimize current scheduling for the hotspot/coldspot instances and edges, the following principles can be applied: (1) increase the resource deployment for the hotspot instance; and (2) reduce the resource deployment for coldspot instances and edges.

To reschedule an instance to another computing node, the following conditions should be satisfied: (1) the instance topology G_{IT} is running continuously; (2) the data stream has no loss; and (3) the instance scheduling is realized through pseudo scheduling. As shown in Fig. 15(a), instance v_{ik} is to be rescheduled to another computing node and instance v_{ij} is the upstream instance of v_{ik}. At the beginning, a replica instance v'_{ik} of v_{ik} is started on the target node. By now, the upstream v_{ij} still distributes the data tuples to v_{ik} only, as shown in Fig. 15(b). When the replica instance v'_{ik} is fully deployed, the upstream v_{ij} starts to distribute data tuples to v'_{ik}, and stops the data distribution to v_{ik}. At the same time, v_{ik} performs state migration to v'_{ik}, as shown in Fig. 15(c). After that, instance v_{ik} is removed and relevant resources are released, as shown in Fig. 15(d).

The algorithm for online resource management is described in Algorithm 4.

Algorithm 4: Online resource management.

```
1.  Input: current instance scheduling scheme for $G_{IT}$, processing capacity of each computing node in data center DC, data stream input ratio $r_{v_1}$.
2.  Output: an optimized instance scheduling scheme for $G_{IT}$ on computing nodes in data center DC under ratio $r_{v_1}$.
3.  Set the maximum throughput $\max(t(G_{IT}))$, the minimum latency $\min(l(G_{IT}))$, and minimum usage efficiency for computing node $\min(e(DC))$ in data center DC.
4.  while throughput $t(G_{IT}) < \max(t(G_{IT}))$ or latency $l(G_{IT}) > \min(l(G_{IT}))$ or $e(DC) < \min(e(DC))$ do
5.      Get the hotspot and coldspot instances from $V(G_{IT})$ of $G_{IT}$.
6.      Get the hotspot and coldspot edges from $E(G_{IT})$ of $G_{IT}$.
7.      Sort all computing nodes in data center DC by the amount of available resources in descending order.
8.      if throughput $t(G_{IT}) < \max(t(G_{IT}))$ or latency $l(G_{IT}) > \min(l(G_{IT}))$ then
9.          Online schedule each hotspot instance $v_{ik}$ to a computing node by following the principles of Algorithm 3.
10.         Eliminate hotspot edges by scheduling the two nodes associated with each hotspot edge to the same computing node that meets SLOs.
11.         Update instance number by Algorithm 1.
12.         Update data stream distribution among multiple instances by Algorithm 2.
13.     end if
14.     if $e(DC) < \min(e(DC))$ then
15.         Online schedule each coldspot instance $v_{ik}$ to a computing node running the same vertex instance(s) by following the principles of Algorithm 3.
16.         Update instances number by Algorithm 1.
17.         Update data stream distribution among multiple instances by Algorithm 2.
18.     end if
19.     Update the hotspot and coldspot instances from $V(G_{IT})$ of $G_{IT}$.
20.     Update the hotspot and coldspot edges from $E(G_{IT})$ of $G_{IT}$.
21.     Update the available resources of each computing node in data center DC
22.     Update the throughput $t(G_{IT})$ and latency $l(G_{IT})$ of $G_{IT}$
23. end while
24. return an optimized instance scheduling scheme for $G_{IT}$ on computing nodes in data center DC under ratio $r_{v_1}$.
```

The input of this algorithm includes current instance scheduling scheme for G_{IT}, processing capacity of each computing node in data center DC and data stream input ratio r_{v_1}. The output is an optimized instance scheduling scheme for G_{IT} on computing nodes under ratio r_{v_1}. Step 8 to step 13 optimize the current instance scheduling scheme to improve throughput $t(G_{IT})$ and latency $l(G_{IT})$. Step 14 to step 18 further optimize the current instance scheduling scheme of G_{IT} by improving the usage efficiency of computing nodes $e(DC)$.

6. Experimental evaluation

This section focuses on the evaluation of the proposed Mc-Stream framework. For comparison purposes, the experimental settings are the same as those in Section 2. Three performance metrics are evaluated for Top_N application: average latency (AL), average throughput (AT) and average resource utilization (ARU).

6.1. Latency

The average latency (AL) of a streaming application is one of the key performance metrics for an elastic stream computing system. We evaluate the application latency under different application numbers, different data stream input rates and different numbers of computing nodes.

Given the data stream input rate is set to 1000 tuples/s, for each application number, the average latency of Top_N by Mc-Stream is shorter than the default scheduling strategy by Storm. With the increase of application number, the difference between the two strategies becomes apparent. As shown in Fig. 16, when the number is less than 12, both the average latencies of Mc-Stream and Storm can stabilize at a certain level, which are 30mz and 43mz, respectively. However, when the application number is greater than 12, both the average latencies are increasing accordingly. When the number reaches 20, the average latencies are 71mz and 106mz, respectively. The latency difference is significant, demonstrating the improvement brought by Mc-Stream against Storm in this scenario.

Given the application number of Top_N is set to 2, under different input rates, the average latency of Mc-Stream is shorter than that of Storm. With the increase of input rate, the difference between the two strategies also becomes apparent. Mc-Stream has a better average latency improvement under higher input rates. As shown in Fig. 17, when the rate is 1000 tuples/s, the average latencies of Mc-Stream and Storm are 30mz and 43mz, respectively. The difference between the two strategies is 13 mz. However, when the rate reaches 5000 tuples/s, the average latencies are 258mz and 391mz, respectively. The difference between the two is 133 mz. Mc-Stream outperforms Storm.

Given the application number of Top_N is set to 2, for different numbers of computing nodes, the average latency of Mc-Stream is shorter than that of Storm. To be more specific, when the number of computing nodes increases to a certain extent, the average latency of Mc-Stream still keeps stable, however, Storm's latency does not. As shown in Fig. 18, when the number of computing nodes is under 15, with the increase of node number, the average latencies of Mc-Stream and Storm both are decreasing. When the number is greater than 15, Mc-Stream's latency stabilizes at a certain level, which is 28 ms. However, for Storm, its latency is slowly increasing. When the number reaches 35, the average latency of Storm becomes 58 ms, 30 ms higher than that of Mc-Stream.

Generally, compared with Storm, Mc-Stream has shorter average latency and greater improvement under conditions of higher application numbers, higher data stream input rates and higher numbers of computing nodes. This is because Mc-Stream supports collaboration between multiple strategies at multiple levels, and tries to maximize the performance optimization space in the shortest possible time.

6.2. Throughput

The average throughput (AT) of a streaming application is its average output rate of data tuples in a stream computing environment. In general, the higher the system throughput, the greater the data processing capability of the computing environment. We evaluate the throughput of the streaming application Top_N under different application numbers, different input rates and different numbers of computing nodes.

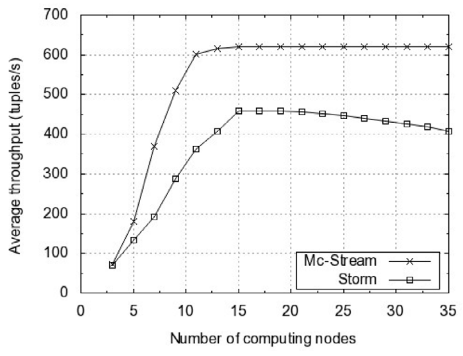


Fig. 21. Average throughput (*AT*) of Top_N with different number of computing nodes.

Given the input rate is set to 1000 tuples/s, for each application number, the average throughput of Mc-Stream is greater than that of Storm. With the increase of application number, a situation of relative resource shortage is developed and the average throughputs of both strategies continue to decline. As shown in Fig. 19, when the application number is less than 12, both the average throughputs stabilize at a certain level, which are 621 tuples/s and 459 tuples/s, respectively. By this point, the throughput difference between the two is quite significant (162 tuples/s) with Mc-Stream outperforming Storm. However, when the number is greater than 12, both throughputs decrease along with the increase of the application number. When the number reaches 20, the average throughputs of Mc-Stream and Storm are 411 tuples/s and 307 tuples/s, respectively. The difference is still notable with Mc-Stream taking the lead.

Given the application number of Top_N is set to 2, under different input rates, the average throughput of Mc-Stream is higher than that of Storm. With the increase of input rate, their difference also becomes apparent. Mc-Stream has a better average throughput improvement under a higher input rate. As shown in Fig. 20, when the rate is 1000 tuples/s, the average throughputs of Mc-Stream and Storm are 621 tuples/s and 459 tuples/s respectively. The difference is 162 tuples/s. However, when the rate reaches 5000 tuples/s, their average throughputs are 3051 tuples/s and 2231 tuples/s, respectively. The difference is 820 tuples/s with Mc-Stream outperforming Storm.

Given the application number of Top_N is set to 2, for different numbers of computing nodes, the average throughput of Mc-Stream is higher than that of Storm. More specifically, when the number of nodes increases to a certain extent, the average throughput of Mc-Stream keeps stable, however, Storm's throughput does not. As shown in Fig. 21, when the number is under 15, with the increase of node number, both the average throughputs are increasing. When the number is greater than 15, Mc-Stream's throughput stabilizes at a certain level (621 tuples/s). However, Storm's is slowly decreasing. When the number of nodes reaches 35, the average throughput of Storm is 407 tuples/s, 214 tuples/s lower than that of Mc-Stream.

From the above scenarios, it can be found that similar to the average latency, Mc-Stream also has higher average throughput and greater improvement over Storm.

6.3. Resource utilization

The average resource utilization (ARU) of computing nodes in data center *DC* can be evaluated by the average CPU utilization of these nodes, which reflects the average loading state of CPU queue. The average resource utilization needs to be kept at a reasonable load level. Either too high or too low is not ideal.

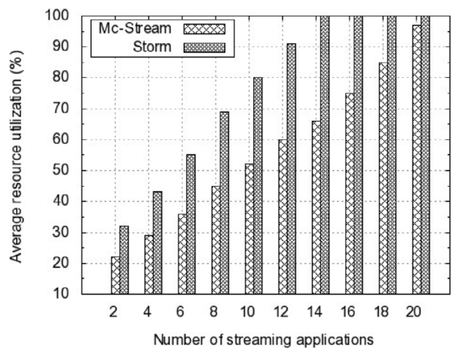


Fig. 22. Average resource utilization (*ARU*) of computing nodes with different number of streaming applications.

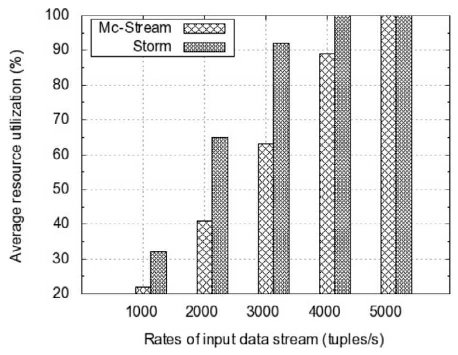


Fig. 23. Average resource utilization (*ARU*) of computing nodes under different input rates.

We evaluate the resource utilization of computing nodes under different application numbers, different input rates and different numbers of computing nodes.

Given the input rate is set to 1000 tuples/s, for each application number, the average resource utilization of Mc-Stream is more efficient than that of Storm. With the increase of application number, a situation of relative resource shortage is developed in the data center *DC*. The average resource utilization rates by both strategies continue increasing. As shown in Fig. 22, when the application number of Top_N is under 14, both the average resource utilization rates are increasing along with the application number. At this stage, compared with Storm, Mc-Stream reduces the resource utilization rate by more than 34%, which means it uses resources more wisely. However, when the application number is greater than 14, the rate of Storm is increasing to 100%, while Mc-Stream still has room to reach the full load. When the application number reaches 20, the rate of Mc-Stream is 97%, while Storm has reached 100% when the number is 14.

Given the application number of Top_N is set to 2, under different data stream input rates, the average resource utilization of Mc-Stream is more efficient than that of Storm. In the case of sufficient resources, with the increase of input rate, the difference between the two strategies becomes apparent. In the case of insufficient resources (e.g. less computing nodes than required), with the increase of input rate, the average resource utilization of Mc-Stream is more efficient than that of Storm. As shown in Fig. 23, when the input rate varies within range [1000, 3000] tuples/s, the utilization rate difference between the two changes from 10% to 29%. When the input rate varies within range [3000, 5000] tuples/s, the resource utilization of Mc-Stream is always more efficient than that of Storm. When the input rate reaches 4000 tuples/s, Mc-Stream and Storm use up to 89% and 100% of resources, respectively.

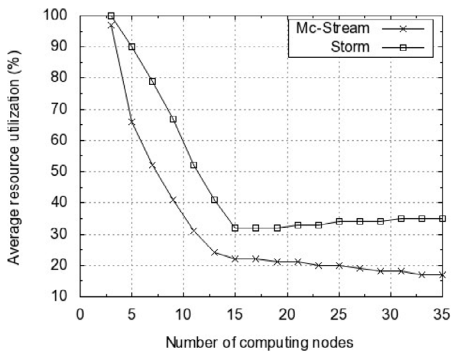


Fig. 24. Average resource (*ARU*) utilization of computing nodes with different number of computing nodes.

Given the application number of Top_N is set to 2, for different numbers of computing nodes, the average resource utilization of Mc-Stream is more efficient than that of Storm. In the case of insufficient resources (e.g. less computing nodes than required), with the increase of node number, both the utilization rates drop sharply. In the case of sufficient resources, with the increase of the node number, the utilization rates of Mc-Stream maintains a steady decreasing trend and achieves a more efficient utilization rates than Storm. As shown in Fig. 24, when the node number varies within range [0, 15], both the utilization rates are dropping sharply. When the node number is greater than 15, Mc-Stream's rate stabilizes at a relatively stable level with a decreasing trend, while Storm's rate also stabilizes at a certain level but maintains a slowly increasing trend. It can be seen that Mc-Stream uses resources more efficiently than Storm in the scenarios of sufficient and insufficient resources.

In summary, similar to the average latency and average throughput, Mc-Stream has more efficient resource utilization of computing nodes and greater improvement over Storm. The harsher the conditions, the more obvious this advantage.

7. Related work

In this section, we review the two broad categories of related work: application scheduling for elastic stream computing systems and system performance optimization of distributed stream computing systems.

7.1. Application scheduling for elastic stream computing systems

Application scheduling plays an important role [23,27] for building an elastic stream computing system. In recent years, more and more researchers [28–30] focus on application scheduling problem to provide a low system latency and high system throughput in an online stream computing environment. However, it is hard to find an optimal scheduling as the scheduling problem for stream applications is NP-hard [7,31,32]. What is more difficult is that the scheduling is carried out in an online environment, which needs to take into account both the system availability and scheduling efficiency.

To process large volume of data with low latency and minimal resources in a stream computing environment, similar to our work, in [9], the authors first showed their finding that is providing a wrong kind of container would lead to performance degradation, then they proposed a multi-level elastic resources provision strategy. The strategy considered different levels of execution containers and provided the least expensive container to increase performance.

Table 2
Comparison of our work and other related work.

Type	Other related work				Our work
	[9]	[23]	[25]	[35]	
User level	✘	✘	✘	✘	✓
Instance level	✓	✘	✓	✘	✓
Scheduling level	✓	✓	✓	✘	✓
Resource level	✓	✓	✓	✓	✓
Coordinate	✘	✘	✘	✓	✓

Focusing on the problem of scheduling streaming applications in a heterogeneous environment, in [33], a maximum throughput scheduler was developed, where a dynamic programming technique was used to efficiently schedule the streaming applications based on the computing and data transfer requirements.

To optimize the usage of heterogeneity resources, such as CPUs, GPUs and FPGAs, in [34], a heterogeneity-aware scheduling was proposed. It tried to facilitate holistic optimization over multiple running tasks with various service level agreements. A framework called HaaS was proposed, aiming at real-time data analytics by encompassing a collection of existing algorithms and primitive operations. Fault-tolerance and at-least-once guarantee were also considered in HaaS.

In [35], the problem of resource allocation and resource placement were considered separately. The proactive elastic resource scheduling problem for computation-intensive and communication-intensive applications was investigated. A dynamic collaborative strategy was proposed and a latency estimation model was constructed to estimate the latency of the streaming applications.

To minimize the cost of processing streaming applications in geographically distributed data centers, in [36], a transformation-based streaming application allocation algorithm was proposed, targeting the objective of cost-minimization. It considered the characteristics of streaming applications and price heterogeneity among geographically distributed data centers.

Reducing the energy consumption of data centers is a necessity. In [37], a holistic energy-efficient real-time scheduler was proposed for mixed stream and batch processing systems, where a DVFS (Dynamic Voltage and Frequency Scaling) technique was applied to minimize energy consumption.

To summarize, building an elastic stream computing system by optimizing scheduling state of applications has been extensively studied. However, most of them tried to build an elastic stream computing system from one perspective or at one level. The summary of the comparison between our work and other related works is given in Table 2. Extensive experiments in this paper show that the extent of performance improvement is limited if only optimizing scheduling from one single perspective or level. It is recommended to build an elastic stream computing system by considering multiple perspectives at multiple levels. Moreover, appropriate coordination is needed as the multiple perspectives or multiple levels are not isolated. To explore this idea, in our work, we build one multi-level collaborative framework for elastic stream computing systems. It can optimize the scheduling performance from multiple levels, including user level, instance level, scheduling level and resource level, and coordinate factors from these levels.

7.2. System performance optimization of distributed stream computing systems

To optimize the performance of distributed stream computing systems, many researchers have widely investigated this area from multiple different aspects, such as large parameter configuration [1,38], fault tolerance management [12,39,40] and state management [41]. All those work improved the system performance from one aspect or another.

To solve the problem of high variability and unpredictability of workloads in a cluster environment, in [1], a scalable and programmable control plane was designed. The plane supported continuous monitoring and feedback, enabled dynamic re-configuration and asynchronously executed control policies. In [38], a self-regulating streaming system was designed, which could automatically reconfigure topologies to meet throughput SLOs and scale resources consumption up and down as needed.

To build a fault tolerance system with fast recovery time and low system overheads, in [12], the authors provided a state management strategy for each vertex in a streaming application. The fault tolerance only focused on the stateful vertices, which reduced the amount of data for checkpoint drastically. It helped improve the recovery time and reduce the overheads. In [39], a fine-grained fault-tolerant strategy for economical resource utilization was proposed. It was on the premise of efficient data recovery. Both workload and fault tolerance were considered in the framework. In [40], a passive and partially active fault tolerance was proposed for parallel stream computing systems, where a passive approach was applied to all tasks while only a selected set of tasks were actively replicated.

To implement an elastic stateful stream computing system, in [41], a library for efficient reconfiguration of running queries in the presence of very large distributed state was proposed. It provided a handover protocol and a state migration protocol to consistently and efficiently migrate stream processing among servers.

Compared with the aforementioned research work, in this paper, we build an elastic stream computing system from the perspective of scheduling streaming applications. The problems such as large parameter configuration, fault tolerance management, and state management are beyond the scope of our work.

8. Conclusions and future work

Stream application scheduling is one of the keys to achieve elasticity in a stream computing system. It requires on-demand determining the relationships between vertices of stream applications and computing nodes in a data center, and on-demand scaling in/out the workload on computing nodes properly during runtime. This scheduling problem is one of the most thought-provoking NP-hard problems. The real-time fluctuating data streams and complex vertex dependencies also increase the challenge to resolving this problem.

Our work is motivated by the observation that unsatisfactory system performance is mainly caused by frequent online rescheduling. Moreover, the performance may be affected by multiple factors at different levels. As these factors are not independent of each other, by simply optimizing the performance from one factor's perspective, the extent of improvement can be limited, and sometimes might even be invalid. To address the above issues, we build a system which supports high system stability, low system latency and effective resource utilization using multi-level collaborative framework. It suits for environments which require relatively long-term online state and deal with fluctuating stream input.

In this paper, comparative experiments are conducted, demonstrating that the system performance is affected by multiple factors at different levels. Based on the Storm platform, a system model is constructed, including a topology model, a data model and a grouping model. The problem of multi-level collaborative framework is formalized, followed by the proposal of the multi-level strategies, which include a lightweight instance number adjustment strategy, an available resource-aware data stream redirection strategy, a fast and effective scheduling management and an asynchronous runtime redeployment method without state loss. Performance metrics such as system latency, throughput, and resources utilization are evaluated using real-world stream applications. The effectiveness of the proposed solution is verified.

Our future work will be focusing on integrating the state management as a part of Mc-Stream, considering the configuration of online optimization parameter to improve the system performance, and applying Mc-Stream in real elastic stream computing scenarios, such as real-time online product recommendation.

CRediT authorship contribution statement

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

Declaration of competing interest

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

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61972364; the Fundamental Research Funds for the Central Universities, China under Grant No. 2652021001; Melbourne-Chindia Cloud Computing (MC3) Research Network, Australia.

References

[1] M. Luo, Z. Kai, P. Rahul, X. Le, S. Steve, V. Shivaram, C. Paolo, K. Terry, M. Saravanan, K. Vamsi, D. Sudheer, R. Sriram, Chi: A scalable and programmable control plane for distributed stream processing systems, in: 44th International Conference on Very Large Data Bases, VLDB 2018, ACM Press, 2018, pp. 1303–1316.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: Fault-tolerant stream processing at internet scale, VLDB Endow. 6 (11) (2013) 1033–1044.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R.J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, VLDB Endow. 8 (12) (2015) 1792–1803.

[4] Storm, https://storm.apache.org.

[5] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J.M. Patel, K. Ramasamy, S. Taneja, Twitter heron: Stream processing at scale, in: Proc. the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015, ACM Press, 2015, pp. 239–250.

[6] Samza, http://samza.apache.org/.

[7] H. Röger, R. Mayer, A comprehensive survey on parallelization and elasticity in stream processing, ACM Comput. Surv. 1 (2019) 1–37, https://arxiv.org/abs/1901.09716.

[8] M. Dias de Assunção, A. da Silva Veith, R. Buyya, Distributed data stream processing and edge computing: A survey on resource elasticity and future directions, J. Netw. Comput. Appl. 103 (2018) 1–17.

[9] M.M. Vania, D.P. Noel, E.R. Ahmed, Multi-level elasticity for data stream processing, IEEE Trans. Parallel Distrib. Syst. 30 (10) (2019) 2326–2337.

[10] D. Sun, S. Gao, X. Liu, X. You, R. Buyya, Dynamic redirection of real-time data streams for elastic stream computing, Future Gener. Comput. Syst. 112 (2020) 193–208.

[11] Y. Wang, Z. Tari, X. Huang, A.Y. Zomaya, A network-aware and partition-based resource management scheme for data stream processing, in: Proc. the 48th International Conference on Parallel Processing, ICPP 2019, ACM Press, 2019, p. a20.

[12] R.C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, in: Proc. ACM International Conference on Management of Data, SIGMOD 2013, ACM Press, 2013, pp. 725–736.
[13] F. Kalim, T. Cooper, H. Wu, Y. Li, N. Wang, N. Lu, M. Fu, X. Qian, H. Luo, D. Cheng, Y. Wang, F. Dai, M. Ghosh, B. Wang, Caladrius: A performance modelling service for distributed stream processing systems, in: Proc. 2019 IEEE 35th International Conference on Data Engineering, ICDE 2019, IEEE Press, 2019, pp. 1886–1897.
[14] X. Liao, Y. Huang, L. Zheng, H. Jin, Efficient time-evolving stream processing at scale, IEEE Trans. Parallel Distrib. Syst. 30 (10) (2019) 2165–2178.
[15] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, T. Roscoe, Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows, in: Proc. the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, USENIX Association, 2018, pp. 783–798.
[16] J. Fang, R. Zhang, T.Z.J. Fu, Z. Zhang, A. Zhou, X. Zhou, Distributed stream rebalance for stateful operator under workload variance, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2223–2240.
[17] B. Remesh, R. Kaippilly, P. Samuel, Service-level agreement–aware scheduling and load balancing of tasks in cloud, Softw. - Pract. Exp. 49 (6) (2019) 995–1012.
[18] Z. Abbas, V. Kalavri, P. Carbone, V. Vlassov, Streaming graph partitioning: An experimental study, Proc. VLDB Endow. 11 (11) (2018) 1590–1603.
[19] D. Cheng, X. Zhou, Y. Wang, C. Jiang, Adaptive scheduling parallel jobs with dynamic batching in spark streaming, IEEE Trans. Parallel Distrib. Syst. 29 (12) (2018) 2672–2685.
[20] A. Singh, P. Ekberg, S. Baruah, Uniprocessor scheduling of real-time synchronous dataflow tasks, Real-Time Syst. 55 (1) (2019) 1–31.
[21] X. Ni, S. Schneider, R. Pavuluri, J. Kaus, K.L. Wu, Automating multi-level performance elastic components for IBM streams, in: Proc. 2019 20th International Middleware Conference, Middleware 2019, ACM Press, 2019, pp. 163–175.
[22] G. Hesse, C. Matthies, K. Glass, J. Huegle, M. Uflacker, Quantitative impact evaluation of an abstraction layer for data stream processing systems, in: Proc. 2019 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, IEEE Press, 2019, pp. 1381–1392.
[23] L. Eskandari, J. Mair, Z.Y. Huang, D. Eyers, T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster, Future Gener. Comput. Syst. 89 (2018) 617–632.
[24] T. Tuor, S. Wang, K.K. Leung, B.J. Ko, Online collection and forecasting of resource utilization in large-scale distributed systems, in: Proc. 2019 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, IEEE Press, 2019, pp. 133–143.
[25] F. Lombardi, L. Aniello, S. Bonomi, L. Querzoni, Elastic symbiotic scaling of operators and resources in stream processing systems, IEEE Trans. Parallel Distrib. Syst. 29 (3) (2018) 572–585.
[26] A. Arfeen, K. Pawlikowski, D. McNickle, A. Willig, Global and local scaling analysis of link streams in access and backbone core networks, Comput. Netw. 149 (2019) 154–172.
[27] J. Rho, T. Azumi, M. Nakagawa, K. Sato, N. Nishio, Scheduling parallel and distributed processing for automotive data stream management system, J. Parallel Distrib. Comput. 109 (2017) 286–300.
[28] H. Jin, F. Chen, S. Wu, Y. Yao, Z. Liu, L. Gu, Y. Zhou, Towards low-latency batched stream processing by pre-scheduling, IEEE Trans. Parallel Distrib. Syst. 30 (3) (2019) 710–722.
[29] A. Vulpe, M. Frincu, Scheduling data stream jobs on distributed systems with background load, in: Proc. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, IEEE Press, 2017, pp. 838–848.
[30] M. Barika, S. Garg, R. Ranjan, Cost effective stream workflow scheduling to handle application structural changes, Future Gener. Comput. Syst. 112 (2020) 348–361.
[31] M. Mortazavi-Dehkordi, K. Zamanifar, Efficient deadline-aware scheduling for the analysis of Big Data streams in public cloud, Cluster Comput. 23 (1) (2020) 241–263.
[32] M. Nardelli, V. Cardellini, V. Grassi, F. Lo Presti, Efficient operator placement for distributed data stream processing applications, IEEE Trans. Parallel Distrib. Syst. 30 (8) (2019) 1753–1767.
[33] A. Al-Sinayyid, M. Zhu, Job scheduler for streaming applications in heterogeneous distributed processing systems, J. Supercomput. 76 (12) (2020) 9609–9628.
[34] J. He, Y. Chen, T. Fu, X. Long, M. Winslett, L. You, Z. Zhang, HaaS: Cloud-based real-time data analytics with heterogeneity-aware scheduling, in: Proc. 2018 IEEE 38th International Conference on Distributed Computing Systems, ICDCS 2018, IEEE Press, 2018, pp. 1017–1028.
[35] X. Wei, L. Li, X. Li, X. Wang, S. Gao, H. Li, Pec: Proactive elastic collaborative resource scheduling in data stream processing, IEEE Trans. Parallel Distrib. Syst. 30 (7) (2019) 1628–1642.
[36] C. Chen, I. Paik, P.C.K. Hung, Transformation-based streaming workflow allocation on geo-distributed datacenters for streaming big data processing, IEEE Trans. Serv. Comput. 12 (4) (2019) 654–668.
[37] S. Maroulis, N. Zacheilas, V. Kalogeraki, A holistic energy-efficient real-time scheduler for mixed stream and batch processing workloads, IEEE Trans. Parallel Distrib. Syst. 30 (12) (2019) 2624–2635.
[38] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy, Dhalion: Self-regulating stream processing in heron, Proc. VLDB Endow. 10 (12) (2017) 1825–1836.
[39] J. Fang, P. Chao, R. Zhang, X. Zhou, Integrating workload balancing and fault tolerance in distributed stream processing system, World Wide Web 22 (6) (2019) 2471–2496.
[40] L. Su, Y. Zhou, Passive and partially active fault tolerance for massively parallel stream processing engines, IEEE Trans. Knowl. Data Eng. 31 (1) (2019) 32–45.
[41] B. Del Monte, S. Zeuch, T. Rabl, V. Markl, Rhino: Efficient management of very large distributed state for stream processing engines, in: Proc. the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD 2020, ACM Press, 2020, pp. 2471–2486.

Dawei Sun is an Associate 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 or co-authored over 60 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.

Xunyun Liu received the B.E. and M.E degree in Computer Science and Technology from the National University of Defense Technology in 2011 and 2013, respectively. He obtained the Ph.D. degree in Computer Science at the University of Melbourne in 2018. His research interests include stream processing and distributed systems.

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 text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 150 with 117,600+ citations). He served as the founding Editor-in-Chief (EiC) of IEEE Transactions on Cloud Computing and now serving as EiC of Journal of Software: Practice and Experience.