Future Generation Computer Systems 97 (2019) 194–209

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

State and runtime-aware scheduling in elastic stream computing systems

Dawei Sun [a,b,*], Shang Gao [c], Xunyun Liu [d], Fengyun Li [e], Xinqi Zheng [a,**], Rajkumar Buyya [d]

[a] School of Information Engineering, China University of Geosciences, Beijing, 100083, PR China
[b] Key Laboratory of Geological Information Technology, Ministry of Natural Resources, PR China
[c] School of Information Technology, Deakin University, Victoria 3216, Australia
[d] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia
[e] School of Computer Science and Engineering, Northeastern University, Shenyang, 110819, PR China

HIGHLIGHTS

- Classify vertex into stateless vertex or stateful vertex.
- Achieve vertex parallelization by considering the state of vertex.
- State and runtime-aware schedule of stream application.
- Implement a prototype and evaluate the performance of the proposed Sra-Stream.

ARTICLE INFO

Article history:
Received 13 September 2018
Received in revised form 28 November 2018
Accepted 22 February 2019
Available online 1 March 2019

Keywords:
State awareness
Runtime awareness
Application scheduling
Elastic stream computing
Big data system

ABSTRACT

State and runtime-aware scheduling is one of the problems that is hard to resolve in elastic big data stream computing systems, as the state of each vertex is different, and the arrival rate of data streams fluctuates over time. A state and runtime-aware scheduling framework should be able to dynamically adapt to the fluctuation of the arrival rate of data streams and be aware of vertex states and resource availability. Currently, there is an increasing number of research work focusing on application scheduling in stream computing systems, however, this problem is still far from being completely solved. In this paper, we focus on the state of vertex in applications and the runtime feature of resources in a data center, and propose a state and runtime-aware scheduling framework (Sra-Stream) for elastic streaming computing systems, which incorporates the following features: (1) Profiling mathematical relationships between the system response time and the arrival rate of data streams, and identifying relevant resource constraints to meet the low response time and high throughput objectives. (2) Classifying vertex into stateless vertex or stateful vertex from a quantitative perspective, and achieving vertex parallelization by considering the state of the vertex. (3) Demonstrating a proposed stream application scheduling scheme consisting of a modified first-fit based runtime-aware data tuple scheduling strategy at the initial stage, and a maximum latency-sensitive based runtime-aware data stream scheduling strategy at the online stage, by considering the current scheduling status of the application. (4) Evaluating the achievement levels of low response time and high throughput objectives in a real-world elastic stream computing system. Experimental results conclusively demonstrate that the proposed Sra-Stream provides significant performance improvements on achieving the low system response time and high system throughput.

1. Introduction

In Big Data Era, an increasing number of application scenarios rely heavily on real-time processing of high-volume continuous data streams, such as social networks, telecommunications, emergency response, fraud detection, system monitoring, smart cities, etc. [1]. In real time computing environments, data streams must be immediately processed to get timely results. To respond to this

* Corresponding author at: School of Information Engineering, China University of Geosciences, Beijing, 100083, PR China.
** Corresponding author.
E-mail addresses: sundaweicn@cugb.edu.cn (D. Sun), shang.gao@deakin.edu.au (S. Gao), xunyunliu@gmail.com (X. Liu), lifengyun@mail.neu.edu.cn (F. Li), zhengxq@cugb.edu.cn (X. Zheng), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2019.02.053
0167-739X/© 2019 Elsevier B.V. All rights reserved.

need, big data stream computing systems can be employed to process heterogeneous, real-time, fast, fluctuating over time, and unbounded data streams in a distributed, scalable, and reliable computing manner. A new generation of big data stream computing system has been developed and deployed. Notable examples include Storm [2], Spark Streaming [3], Flink [4], and Samza [5]. Storm, one of the most popular open source big data stream computing systems, has been widely used in many well-known companies and organizations [6,7], such as Twitter, Alibaba, etc. Storm provides an on-the-fly computing paradigm, where data is directly processed by a running topology in memory without the need for storage. It can reduce the response time to a range of milliseconds or sub-seconds, overcoming the problem of long response times (which may vary from minutes to weeks) faced by many big data batch computing systems, such as Hadoop [8], which adopts a store-then-process computing paradigm.

Stream applications are commonly modeled as a set of subtasks interconnected via data dependencies described by a corresponding DAG (Directed Acyclic Graph). Vertices in a DAG represent sub-tasks as well as computations, and edges in a DAG represent data dependencies as well as communications between those vertices. Each DAG is submitted to a big data stream computing platform and scheduled to run on one or more computing nodes. Take Storm as an example, once a DAG is running in Storm system, its execution is continuous unless explicitly killed by the administrator. The goal of a stream scheduling strategy is typically related to scheduling inter-dependent sub-tasks onto currently available computing nodes so that a DAG can complete its execution within specified constraints, such as throughput and response time. It is beneficial if some vertices of the running DAG be rescheduled online according to the fluctuating arrival rate of streams and available computing resources.

Low system response time and high system throughput are two critical performance requirements for a stream computing system [9]. It is necessary to take advantage of distributed data centers and their multiple instances to achieve high throughputs, as well as using runtime scheduling to lower response time. Application scheduling is the key to achieve these goals, which focuses on scheduling tasks to computing nodes in a way that a set of objective constraints are satisfied. Application scheduling problem is also one of the most thought-provoking NP-hard problems in general cases [10]. Data streams arrive in real time and should be processed immediately. More importantly, a data stream is composed of continuous, ordered, unbounded stream of data tuples and its volume can fluctuate over time. An elastic stream computing system always needs elastic adjustment of computing resources and vertex parallelism. All of these requirements make the application scheduling problem important and challenging.

The arrival rate of a data stream fluctuates over time in an unpredictable manner. To effectively utilize resources and meet user's specified SLA (Service Level Agreement) constraints such as response time, a fundamental solution is to propose an elastic adaptive scheduling strategy to allow for stream application adaptation with regard to the data stream fluctuations. There is an increasing number of research work [11,12] focusing on elastic strategies in stream computing systems. An elastic steam computing system can be achieved by dynamically adjusting instance parallelism of vertex to enable dynamical adaptation to changing arrival rates of data streams. [13] proposed a system called Elastic-PPQ to process spatial preference queries over dynamic data streams. [14] and [15] focus on elasticity in data stream processing systems. The authors used Model Predictive Control to predict the system behavior and a set of energy-aware proactive strategies were employed to make a better adjustment decision. However, elastic scheduling strategy does not always work. Once an application is submitted to a computing system, it will keep running until being explicitly terminated. Therefore, an inappropriate scheduling strategy will cause the system imbalance for a long time, which is not acceptable in steam computing environments. There are also some works [11] [16] on static scheduling. Storm is a popular big data stream computing platform with large communities in both academia and industry. It adopts a static application model, which includes a static definition of the number of instances for each vertex, and a deployment of a DAG on a fixed number of computing nodes. However, the static strategies either require permanent peak-load resource provision to remain low latency in face of varying and busty data streams, which may cause poor resource utilization, or are unable to handle the unexpected fluctuation [17].

In this paper, we investigate a state and runtime-aware scheduling strategy for handling fluctuating and continuous data streams. The scheduling strategy is built into an elastic steam computing system, which minimizes system response time and maximizes system throughput. In order to mitigate the system imbalance in the scheduling process, we also provide a lightweight scheduling strategy in the online scheduling process. The state and runtime-aware scheduling strategy should be able to determine when and how the running vertices of a DAG should be re-scheduled according to the fluctuating arrival rate of data streams. To achieve this goal, we need to know the state of each vertex in each DAG, obtain a clear picture of the changing status of data streams and system resources, and at runtime reschedule critical vertices of the DAG to provide a lightweight and effective scheduling strategy.

A. Key contributions

Our contributions are summarized as follows:

(1) Profile mathematical relationships between the system response time and the arrival rate of data streams, and indicate resource constraints to meet the low response time and high throughput objectives for common stream computing environments.

(2) Classify vertex into stateless vertex or stateful vertex from a quantitative perspective, and achieve vertex parallelization by considering the state of the vertex.

(3) Schedule a stream application with a modified first-fit based runtime-aware data tuple scheduling strategy at the initial stage, and reschedule the application with a maximum latency-sensitive based runtime-aware data stream scheduling strategy at the online stage, by considering the current scheduling status of the application.

(4) Evaluate achievement levels of low response time and high throughput objectives in a real-world elastic stream computing system.

(5) Implement a prototype and evaluate the performance of the proposed Sra-Stream, which is capable of balancing between the low system response time and the high system throughput objectives efficiently and effectively.

B. Paper organization

The rest the paper is organized as follows. We discuss the background of stream computing system in Section 2, covering the logical graph, instance graph, and scheduling scheme of TOP_N in Apache Storm. Section 3 formalizes the data stream, application DAG model, and DAG makespan from a quantitative perspective in elastic stream computing systems. Section 4 presents a DAG scheduling model from a theoretical point of view. Section 5 focuses on the system architecture, vertex state, vertex parallelization, initial DAG scheduling, and online DAG rescheduling

```
builder.setSpout("a", new TestWordSpout(), 1);
builder.setBolt("b", new RollingCountBolt(9, 3), 4).
    fieldsGrouping("a", new Fields("word"));
builder.setBolt("c", new IntermediateRankingsBolt(TOP_N), 1).
    fieldsGrouping("b", new Fields("obj"));
builder.setBolt("d", new TotalRankingsBolt(TOP_N), 1).
    globalGrouping("c");
```

Fig. 1. Source code of TOP_N in Storm.

with runtime-awareness in the proposed Sra-Stream framework. Section 6 provides detailed information regarding the experimental environment, parameter setup and the performance evaluation of Sra-Stream. Section 7 reviews the related work on state management of stream computing system, runtime-aware scheduling in distributed systems, and application scheduling on Storm platform. Finally, conclusions and future work are given in Section 8.

2. Background

Storm is a popular big data stream computing platform both in academia and industry. On Storm platform, an application can be described by a DAG, which is also called a topology. There are two types of vertices in a DAG: spout vertex and bolt vertex. Spout vertex is a vertex of data source, and it sends data tuples (a data tuple is a key–value pair) to bolt vertices continuously. A bolt vertex is a vertex to process data tuples in the custom way implemented by users. There are two types of bolts: stateless bolt and stateful bolt. In a stateless bolt, bolt result is only related to an input data tuple, however, in a stateful bolt, bolt result is always related to a set of input data tuples. Both instance numbers of spout and bolt vertex can be set by user to increase parallelism.

The DAG of an application can be further divided into two types: logic graph in function, and instance graph in runtime. As shown in Fig. 1, the source code snippet is the main part of a streaming application on the Storm platform to achieve TOP_N computing function [18], which finds the most popular words from the input data streams over a period of time.

In Storm, a data stream is abstracted as a series of data tuples, with each data tuple being processed by a corresponding vertex, and transferred to a downstream vertex. There are eight data tuples partition strategies [18] provided by Storm to transfer data tuples from upstream vertex to downstream vertex, and to partition them among multiple instances of the downstream vertex. These strategies include shuffle grouping strategy, fields grouping strategy, partial key grouping strategy, all grouping strategy, global grouping strategy, none grouping strategy, direct grouping strategy, and local or shuffle grouping strategy. A custom stream grouping can also be employed by implementing the CustomStreamGrouping interface.

The logic graph of TOP_N is a linear pipeline (as shown in Fig. 2), where each vertex only has one upstream and one downstream vertex, f represents the fieldsGrouping strategy that transfers tuples from upstream vertex to downstream vertex, and g represents the globalGrouping strategy.

An instance graph of TOP_N is a precedence constraint based directed acyclic graph (as shown in Fig. 3), where vertex v_a is mapped into one parallel instance, vertex v_b is mapped into four parallel instances, and vertex v_c is mapped into one parallel instance. In Storm, the number of instances of each vertex can be set by the user.

A scheduling scheme [19] for instance graph of TOP_N is shown in Fig. 4. The data stream graph is submitted to Nimbus,

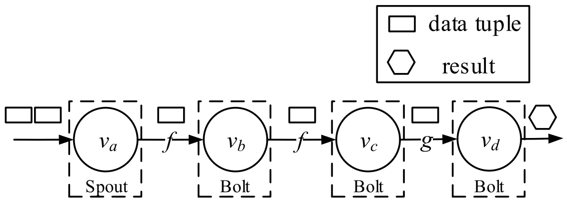

Fig. 2. Logical graph of TOP_N in Storm.

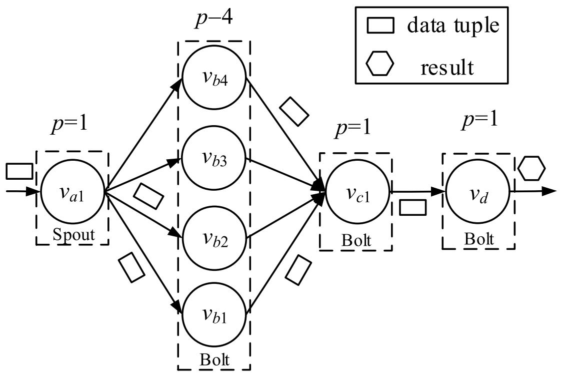

Fig. 3. Instance graph of TOP_N in Storm.

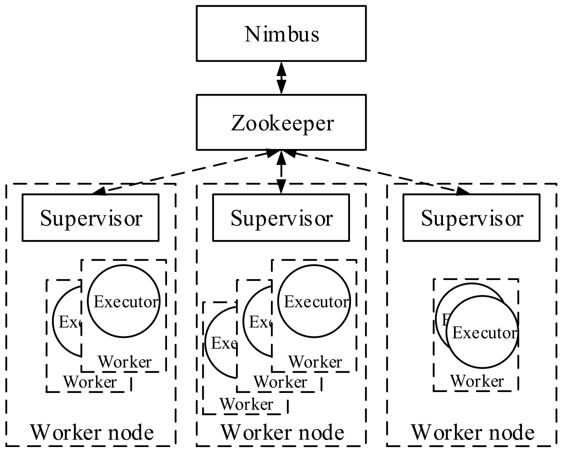

Fig. 4. A scheduling scheme for instance graph of TOP_N in Storm.

to be executed in a Storm cluster. A Nimbus in the cluster is used to receive and manage the topology of the running data stream graph. One or more Supervisors in the cluster are used to coordinate work nodes, to monitor states of tasks in the topology, and to inform state changes to Nimbus. One or multiple tasks (in executor) are running on each work node. Zookeeper [20] is a centralized service for maintaining configuration information, as well as providing distributed synchronization and group services. There could be one or more Zookeepers in the cluster to manage states for Supervisors and Nimbus, such as storing heartbeat states. Nimbus can monitor and coordinate the work of Supervisors and Nimbus, quickly restarting any failed workers in Supervisor. Zookeepers can also be used to submit and monitor stream applications. Supervisors and Nimbus are stateless, and Zookeepers is stateful. To avoid a single point of failure, a copy of Zookeeper is necessary.

Precedence-constraint based directed acyclic graph scheduling is a process of mapping inter-dependent sub-tasks onto available computing nodes so that an application can complete its execution within user's specified SLA (Service Level Agreement) constraints such as deadline. In Storm, the available resources of a computing node are called "Slots". When an application is submitted to Storm system, the scheduling strategy schedules the vertices of the instance DAG to free slots in computing nodes within the user's specified SLA. However, finding an optimal scheduling for precedence-constraint based directed acyclic graph has been studied for years, which is proved to be NP-hard [21]. Heuristics can be used to obtain a sub-optimal scheduling rather than parsing all the possible schedules [22].

The scheduling strategy of instance DAG plays a key role in improving system performance. In Storm 1.1.0, there are four kinds of built-in schedulers, which are namely DefaultScheduler, IsolationScheduler, MultitenantScheduler, and ResourceAwareScheduler. DefaultScheduler tries to schedule an instance DAG to all computing nodes with a round-robin strategy. IsolationScheduler supports setting custom priorities on the computing nodes to achieve isolation of resource usages among instance DAGs and to avoid competition. MultitenantScheduler tries to schedule instance DAGs to the computing nodes in a multiple tenant model. ResourceAwareScheduler considers both the available memory and CPU of a computing node in a cluster, with rack awareness being considered in the scheduling process. However, these existing schedulers cannot adaptively adjust a running instance DAG. State and runtime-aware factors are not considered. A state and runtime-aware adaptive scheduling strategy is needed to bridge this gap.

3. Overview of data stream computing

In this section, we formalize data stream, application DAG model, and DAG makespan from a quantitative perspective in elastic stream computing systems.

A. Data stream

A data stream S is a continuous, unbounded, bursty, and fluctuating-over-time sequence of data tuples, $S = \{(K_1, V_1, ts_1), (K_2, V_2, ts_2), \ldots, (K_i, V_i, ts_i), \ldots\}$. For the ith data tuple $dt_i = (K_i, V_i, ts_i)$, K_i, V_i, and ts_i represent the key, value, timestamp of the ith data tuple, respectively. For a data stream S, each data tuple is a key–value pair. The data size of each data tuple is different and independent of each other. All those data tuples are relatively ordered by timestamp.

Each vertex can set a dedicated sliding window [23], which is used to temporary storage the most recent data tuples if the relevant vertex is processing another data tuple. Usually, a sliding window can reduce the loss rate of data tuples and improve the system throughput of a data stream in distributed stream computing systems. As shown in Fig. 5, a sliding window with the length of l_{sw} is employed. At t_i, the sliding window is empty, and no data tuple is stored in it; At t_j, some data tuples are stored in the sliding window, but there is still some available space that can be used to store the newly arrived data tuples. However, at t_k, the sliding window is full of data tuples, and there is no space available. In this situation, we need to selectively discard some of the data tuples. We can discard the data tuples from the existing tuple set that is stored in the corresponding sliding window, or discard the data tuples from the latest incoming tuples. For the sake of simplicity, we directly discard the latest incoming data tuples. Therefore, if a new data tuple arrived at t_k, it would be discarded directly.

Each vertex should employ a sliding window to store the most recent data tuples. For stateless nodes, sliding windows can be used to store the newly arrived tuples that are not yet ready for processing. For stateful nodes, the sliding window can be further used to maintain a part of the processed tuples to complete the dependency calculation of neighboring tuple states.

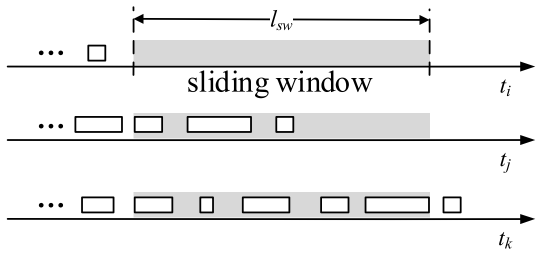

Fig. 5. Sliding window for a data stream.

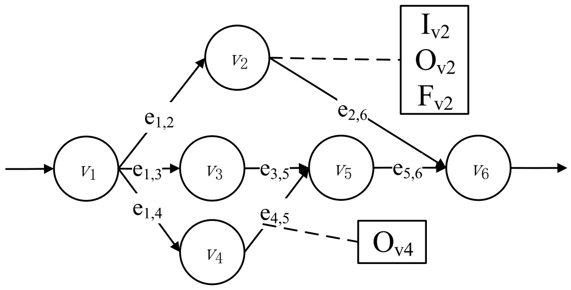

Fig. 6. A logical DAG of a stream application.

B. Application DAG model

The logic of each stream application in stream computing system is usually described by a Directed Acyclic Graph [24], which is composed of a vertex set and a directed edge set. Such a logical DAG can be denoted as $DAG = (V(DAG), E(DAG))$, where $V(DAG) = \{v_1, v_2, \ldots, v_n\}$ is a finite set of n vertices. $E(DAG) = \{e_{1,2}, e_{1,3}, \ldots, e_{n-i,n}\}, i \in \{1, 2, \ldots, n\}$ is a finite set of directed edges. The weight associated with a vertex or an edge represents its computation cost or communication cost, respectively.

As shown in Fig. 6, the logical DAG consists of 6 vertices, and 7 edges. $V(DAG) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $E(DAG) = \{e_{1,2}, e_{1,3}, e_{1,4}, e_{2,6}, e_{3,5}, e_{4,5}, e_{5,6}\}$. For vertex v_2, input data stream I_{v2} comes from edge $e_{1,2}$, output data stream O_{v2} is placed on edge $e_{2,6}$, function of vertex v_2 is F_{v2}, which is used to process output data stream O_{v2}.

Assuming vertex v_i is running on computing node cn_j with available resources R_{cn_j}, weight w_{v_i} of vertex v_i is determined by F_{v_i} and available resources R_{cn_j} on computing node cn_j, which can be described by (1).

$$w_{v_i} = f_{v_i}\left(F_{v_i}, R_{cn_j}\right). \tag{1}$$

For edge $e_{i,k}$, it transfers output data stream O_{v_i} from vertex v_i to v_k. Weight $w_{e_{i,k}}$ of edge $e_{i,k}$ is determined by O_{v_i} and the network link $I_{e_{i,k}}$ between vertex v_i and v_k, which can be described by (2).

$$w_{e_{i,k}} = f_{e_{i,k}}\left(O_{v_i}, l_{e_{i,k}}\right). \tag{2}$$

An application starts running when its logical DAG is submitted to the data center, in which the corresponding instances for each vertex in the instance DAG are created. Some vertices the instance DAG may have more than one instances. As shown in Fig. 7, instance number of vertex v_2 is 3, including $v_{2,1}$, $v_{2,2}$, and $v_{2,3}$. Instance number of vertex v_5 is 2, including $v_{5,1}$ and $v_{5,2}$. Instance number of vertex v_1 is 1, which is $v_{1,1}$.

Functions of all instances of a vertex are the same, and they are always scheduled for different computing nodes. For example, vertex v_i is composed of n instances at t_1, that is $v_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,n}\}$. The input data stream IS_{v_i} and the output data stream OS_{v_i} are the sum of input data stream and output data

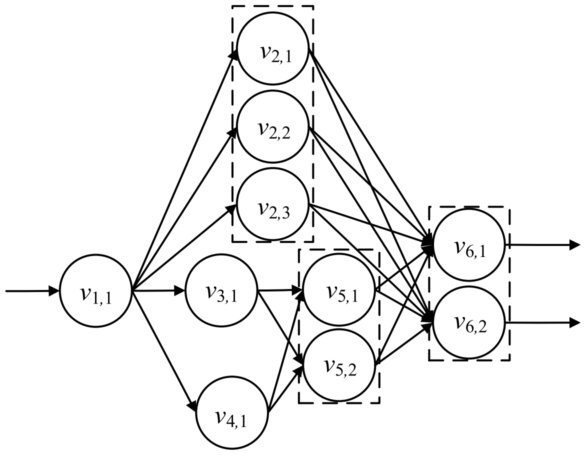

Fig. 7. Instance number of vertex in an instance DAG.

stream of all n instances of v_i, which can be calculated by (3) and (4), respectively.

$$\begin{cases} IS_{v_{i,k}} = \omega_{i,k} \cdot IS_{v_i}, \omega_{i,k} \in [0,1], \sum_{k=1}^{n} \omega_{i,k} = 1, \\ IS_{v_i} = \sum_{k=1}^{n} IS_{v_{i,k}} = \sum_{k=1}^{n} \left(\omega_{i,k} \cdot IS_{v_i}\right), \end{cases} \tag{3}$$

where $IS_{v_{i,k}}$ and $\omega_{i,k}$ are the input data stream and the weight of the kth instance of v_i, respectively. Weight $\omega_{i,k}$ of the kth instance of v_i is determined by tuples partition strategies from upstream vertex to vertex v_i.

$$\begin{cases} OS_{v_{i,k}} = w_{i,k} \cdot OS_{v_i}, w_{i,k} \in [0,1], \sum_{k=1}^{n} w_{i,k} = 1, \\ OS_{v_i} = \sum_{k=1}^{n} OS_{v_{i,k}} = \sum_{k=1}^{n} \left(w_{i,k} \cdot OS_{v_i}\right), \end{cases} \tag{4}$$

where $OS_{v_{i,k}}$ and $w_{i,k}$ are the output data stream and the weight of the kth instance of v_i, respectively. Weight $w_{i,k}$ of the kth instance of v_i is determined by the particular tuple partition strategy from vertex v_i to downstream vertex v_d.

C. DAG makespan

Makespan [25] M_{DAG} of an instance DAG is the average elapsed time required to process each data tuple in $[t_s, t_e]$ on the instance DAG, where t_s is the time that a data tuple enters the instance DAG, and t_e is the time that a data tuple leaves the instance DAG. All of these time measurements are obtained through the monitoring module. As shown in Fig. 8. For the ith data tuple dt_i, the makespan $M_{DAG}(i)$ of the instance DAG for the ith data tuple dt_i is the total elapsed time required to execute the ith data tuple dt_i on the instance DAG, which can be calculated by (5).

$$M_{DAG}(i) = t_{ei} - t_{si}, \tag{5}$$

where t_{si} and t_{ei} are the start time and end time of the instance DAG to process the ith data tuple dt_i, respectively.

As the processing time of each data tuple is not the same, the makespan of each data tuple in an instance DAG is different. Even if the processing time of some tuples is the same, the makespan of each data tuple may also be dynamically affected by other factors, such as available computing nodes in data center, arrival rate of a data stream at a time.

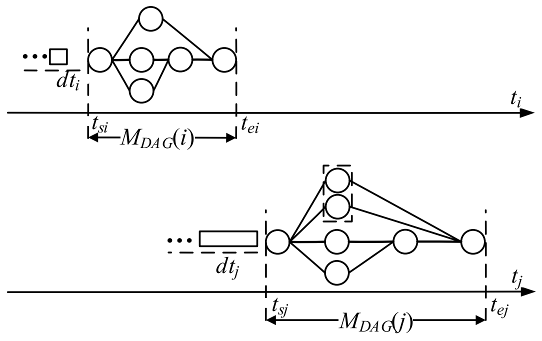

Fig. 8. Makespan of an instance DAG at different time.

Makespan M_{DAG} of an instance DAG for all n data tuples during $[t_s, t_e]$ is the average makespan of the instance DAG for all n data tuples, which can be calculated by (6).

$$\begin{aligned} M_{DAG} &= \tfrac{1}{n} \sum_{i=1}^{n} M_{DAG}(i) \\ &= \tfrac{1}{n} \sum_{i=1}^{n} (t_{ei} - t_{si}). \end{aligned} \tag{6}$$

The makespan M_{sys} of all m instance DAGs in a distributed stream computing system during $[t_s, t_e]$ can be calculated by (7).

$$\begin{aligned} M_{sys} &= \frac{1}{m} \sum_{k=1}^{m} M_{DAG_k} \\ &= \frac{1}{m} \sum_{k=1}^{m} \left(\frac{1}{n_k} \sum_{i=1}^{n_k} M_{DAG}(i) \right), \end{aligned} \tag{7}$$

where n_k is the number of data tuples of the kth instance DAG in $[t_s, t_e]$.

Makespan M_{sys} is one of the key factors to evaluate the performance of a distributed stream computing system.

4. Scheduling model

In a data center,the resources on one computing node cn can be measured in different dimensions, including CPU, Memory, and I/O, etc. [26]. In this paper, we consider CPU and memory resources.

For an instance $v_{i,j}$ of a vertex v_i running on computing node cn_k, the CPU consumption $C_{v_{i,j}}$ can be calculated by (8).

$$C_{v_{i,j}} = c_{v_{i,j},p} + c_{v_{i,j},it} \cdot \alpha_{v_{i,j},v_{i,j-pre}} + c_{v_{i,j},ot} \cdot \beta_{v_{i,j},v_{i,j-suc}}, \tag{8}$$

where $c_{v_{i,j},p}$, $c_{v_{i,j},it}$, and $c_{v_{i,j},ot}$ denote the CPU consumption of processing, input, and output of a data tuple, respectively. The parameter $\alpha_{v_{i,j},v_{i,j-pre}}$ and $\beta_{v_{i,j},v_{i,j-suc}}$ can be calculated by (9) and (10), respectively.

$$\alpha_{v_{i,j},v_{i,j-pre}} = \begin{cases} 0, & \text{if } v_{i,j} \text{ and } v_{i,j-pre} \text{ are run on } cn_k, \\ 1, & \text{otherwise}, \end{cases} \tag{9}$$

$$\beta_{v_{i,j},v_{i,j-suc}} = \begin{cases} 0, & \text{if } v_{i,j} \text{ and } v_{i,j-suc} \text{ are run on } cn_k, \\ 1, & \text{otherwise}, \end{cases} \tag{10}$$

where $v_{i,j-pre}$ is the direct predecessor node of $v_{i,j}$, and $v_{i,j-suc}$ is the direct successor node $v_{i,j}$.

At time t, a computing node cn_k with n_{cn_k} $\left(n_{cn_k} \geq 0\right)$ instances, is denoted as $V_{cn_k,t}$, with each instance needs to process at most one tuple at a time. Therefore, the CPU consumption $C_{cn_k,t}$ of all n_k instances at time t can be calculated by (11).

$$C_{cn_k,t} = \sum_{v_{ij} \in V_{cn_k,t}} \left(C_{v_{ij}} \cdot x_{v_{ij},t} \right), \tag{11}$$

where parameter $x_{v_{i,j},t}$ can be calculated by (12).

$$x_{v_{i,j},t} = \begin{cases} 1, & \text{one tuple is being processed on } v_{i,j} \text{ at } t, \\ 0, & \text{otherwise,} \end{cases} \tag{12}$$

To keep the CPU resources of computing node cn_k in a serviceable state, CPU consumption $C_{cn_k,t}$ must be less than the total available CPU resources $C_{cn_k,tal}$ of computing node cn_k at time t. It can be described by (13).

$$C_{cn_k,t} \leq \delta_{\max(CPU)} \cdot C_{cn_k,tal}, \delta_{\max(CPU)} \in (0, 1], \tag{13}$$

where $\delta_{\max(CPU)}$ is an upper limit adjustment parameter for CPU resources. The larger the adjustment parameter $\delta_{\max(CPU)}$ is, the less the CPU resources will be reserved without allocation.

To avoid continuous inefficient utilization of CPU resources of computing node cn_k, CPU consumption $C_{cn_k,t}$ at time t should be greater than the total CPU resources $C_{cn_k,tal}$ of computing node cn_k. It can be described by (14).

$$\begin{cases} C_{cn_k,t} \geq \delta_{\min(CPU)} \cdot C_{cn_k,tal}, \in (0, 1], \\ \delta_{\min(CPU)} < \delta_{\max(CPU)}, \end{cases} \tag{14}$$

where $\delta_{\min(CPU)}$ is a lower limit adjustment parameter for CPU resources.

If $C_{cn_k,t} < \delta_{\min(CPU)} \cdot C_{cn_k,tal}$, all those n_k instances running on computing node cn_k should be rescheduled, and computing node cn_k will be shut down.

The available CPU resources C_{cn_k,ava_t} of computing node cn_k at time t can be calculated by (15).

$$C_{cn_k,ava_t} = \delta_{\max(CPU)} \cdot C_{cn_k,tal} - C_{cn_k,t}, \tag{15}$$

The maximum available CPU resources is $C_{cn_k,ava_t} = \delta_{\max(CPU)} \cdot C_{cn_k,tal}$. In this situation, $C_{cn_k,t} = 0$, and there are no task instances running on cn_k at time t. The minimum available CPU resources is $C_{cn_k,ava_t} = 0$, which is $\delta_{\max(CPU)} \cdot C_{cn_k,tal} = C_{cn_k,t}$, and there are no more task instances that can be allocated on cn_k at time t. If and only if $C_{cn_k,ava_t} > 0$, new instances can be allocated on cn_k at time t.

Similarly, available resources measured in other resource dimensions on computing node cn_k at time t also need to meet similarly constraints, which includes memory consumption $Mem_{cn_k,t}$. It should be less than the total memory resource $Mem_{cn_k,tal}$. $\delta_{\max(M)}$ is an upper limit adjustment parameter for memory resources, and $\delta_{\min(M)}$ is a lower limit adjustment parameter for memory resources. The available memory resources Mem_{cn_k,ava_t} should meet the following constraint: $0 \leq Mem_{cn_k,ava_t} \leq \delta_{\max(M)} \cdot Mem_{cn_k,tal}$.

Let $DC = \left\{cn_1, cn_2, \ldots, cn_{n_{cn}}\right\}$ be a data center composed of n_{cn} computing nodes, and $U = \left\{u_1, u_2, \ldots, u_{m_u}\right\}$ be a user set composed of m_u users. The DAG scheduling problem of all m_u instance DAGs in the data center can be formalized as follows:

$$\min M_{sys} \tag{16}$$

subject to

$$\sum_{l=0}^{n_{cn}} \sum_{k=0}^{m_u} \sum_{v_i(k) \in DAG_k} \sum_{v_{i,j}(k) \in v_i(k)} x_{l,k,i,j} = 1, \tag{17}$$

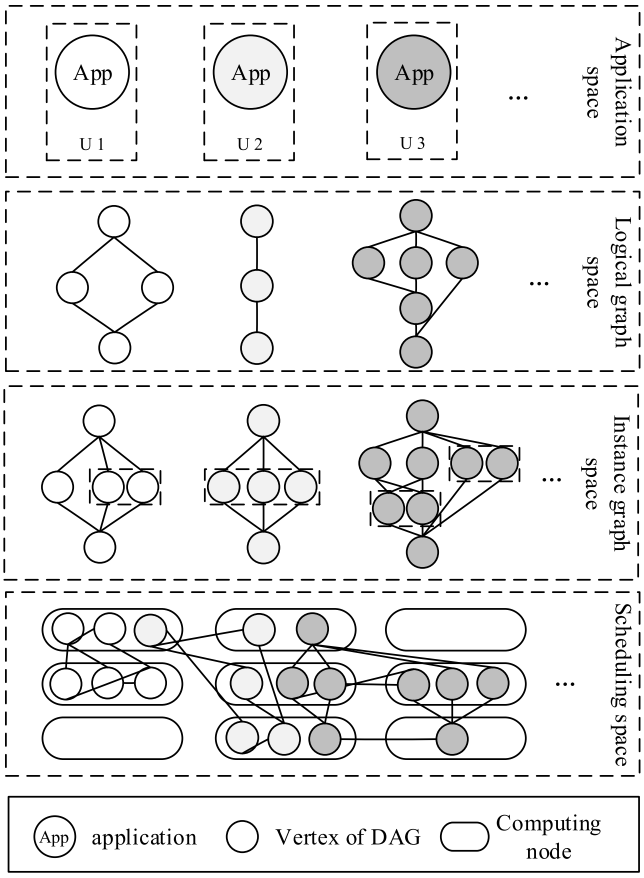

Fig. 9. Sra-Stream architecture.

$$\begin{aligned} &\forall cn_k \in \left\{cn_1, cn_2, \ldots, cn_{n_{cn}}\right\}, \\ &0 \leq C_{cn_k,ava_t} \leq \delta_{\max(CPU)} \cdot C_{cn_k,tal}, \end{aligned} \tag{18}$$

$$\delta_{\min(CPU)} \cdot C_{cn_k,tal} \leq C_{cn_k,t} \leq \delta_{\max(CPU)} \cdot C_{cn_k,tal}, \tag{19}$$

$$0 \leq Mem_{cn_k,ava_t} \leq \delta_{\max(M)} \cdot Mem_{cn_k,tal}, \tag{20}$$

$$\delta_{\min(M)} \cdot Mem_{cn_k,tal} \leq Mem_{cn_k,t} \leq \delta_{\max(M)} \cdot Mem_{cn_k,tal}. \tag{21}$$

If an instance $v_{i,j}(k)$ of a vertex $v_i(k)$ in the kth instance DAG_k running on computing node cn_l, then $x_{l,k,i,j} = 1$, otherwise, $x_{l,k,i,j} = 0$.

5. Sra-Stream overview

Based on the above theoretical analysis, we have proposed and developed Sra-Stream, a state and runtime-aware scheduling framework. To provide an overview of the framework, in this section, we discuss its overall structure, including system architecture, vertex state, vertex parallelization, initial stage DAG scheduling, and online stage DAG rescheduling with runtime awareness.

A. System architecture

The system architecture of Sra-Stream is composed of four spaces, which are scheduling space, instance graph space, logical graph space, and application space, as shown in Fig. 9.

In the scheduling space, a data center DC consists of n computing nodes. Some computing nodes have some instances of vertex assigned to them according to a designated scheduling

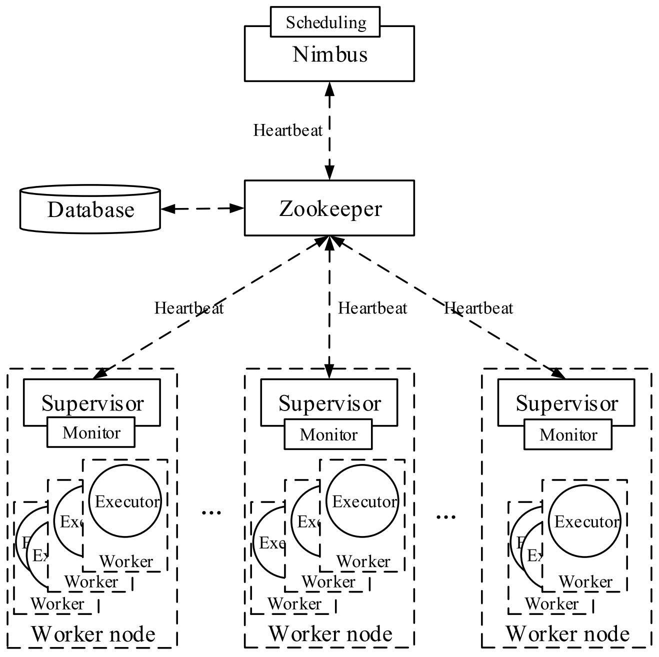

Fig. 10. Sra-Stream topology.

strategy. Resources on each computing node should be utilized in an efficient way without affecting node serviceability. Meanwhile, continuously inefficient utilization should be avoided. In this case, the instances running on one computing node will have to be rescheduled to other available computing nodes, and the current computing node will be shut down. On Storm platform, the scheduling strategy can be employed by implementing the IScheduler interface [5] in order to achieve runtime-aware scheduling objectives. The CPU and memory consumption of the computing nodes are obtained through the monitoring module, which provides the basis for subsequent optimization.

In the instance graph space, one or many instances of an instance DAG are created according to the function of the vertex, the available resources of the computing node, the number of running vertex, and the throughput of data tuples. The number of instances for each vertex is also dynamically adjusted as needed.

In the logical graph space, one or many logical DAGs are created according to the function of the user's application. The function of each application is described by a logical DAG, which has a specific structure and function, consisting of a vertex set and an edge set.

In the user space, each user can design and submit applications to Storm platform according to their needs. Once an application is submitted, it will run forever unless being terminated manually or interrupted due to failure.

As shown in Fig. 10, the topology of Sra-Stream system consists of a Nimbus subsystem, a Zookeeper subsystem, and a Supervisor subsystem. Each logical DAG is scheduled to an appropriate Supervisor subsystem by the Nimbus subsystem. Any specific scheduling strategy can be achieved through the implementation of the IScheduler interface, and the configurations in Storm.yaml (the configuration file of Storm platform) can be customized to specify which scheduling strategy is employed. A monitor module is added to the Supervisor subsystem and the Nimbus subsystem. All monitoring data is stored in the database and used for online scheduling and rescheduling.

B. Vertex state

The state of a vertex for a logical DAG can be stateless or stateful [27] in distributed stream computing systems.

For a stateless vertex v_k, all the input data streams $IS_{v_k} = \{dt_1, dt_2, \ldots\}$ of v_k are processed by v_k independently. Each output data tuple is only related to the corresponding input data tuple, but to other input data tuples. That is $odt_i = f_{v_k}(v_k, dt_i)$, where odt_i, f_{v_k}, and dt_i are the ith output data tuple, the function of v_k, and the ith input data tuple, respectively. The number of output data tuple is not always equal to the number of input data tuple.$\forall dt_i, dt_j \in IS_{v_k}, i \neq j$, if $\exists odt_i, odt_j \in OS_{v_k,1}, i \neq j$, then $\exists odt_i, odt_j \in OS_{v_k,2}$, and processing results of different ordered input data tuples will be the same. As shown in Fig. 11, the order of $dt_i$and dt_j is interchangeable, but the result sets $OS_{v_k,1} = \left\{odt_1, \ldots, odt_i, \ldots, odt_j \cdots\right\}$ and $OS_{v_k,2} = \left\{odt_1, \ldots, odt_j, \ldots, odt_i \cdots\right\}$ are identical. As well, the order of $odt_i$and odt_j is interchangeable.

In a distributed stream computing system, stateless vertices can be adjusted online (such as the number of vertex instances, the scheduling of vertex instances) at any time, regardless of their historical status.

For a stateful vertex v_t, all those input data tuples $IS_{v_t} = \{dt_1, dt_2, \ldots\}$ of v_t are processed by v_t sequentially. Each output data tuple is not only related to the corresponding input data tuple, but also related to other input data tuples that have been processed. It can be described as $odt_i = f_{v_t}(v_t, dt_i, \ldots, dt_l, \ldots)$, where odt_i, f_{v_t}, dt_i, and $\cdots, dt_l, \ldots$ are the ith output data tuple, the function of v_t, the ith input data tuple, and other input data tuples of v_t, respectively. Different input data tuples or different ordered input data tuples would result in different processing results. As shown in Fig. 12, in a vertex v_{sum} with a function of calculating the sum of the data tuples in the least sliding window, the results of output data tuple set $OS_{v_{sum},1} = \{odt_1, \ldots, odt_{k'}, \ldots\}$ and $OS_{v_{sum},2} = \{odt_{1'}, \ldots, odt_{k'}, \ldots\}$ are independent of each other.

In a distributed stream computing system, it is required to consider the historical status when a stateful vertex is to be adjusted online. For example, to reschedule a stateful vertex v_t from computing node cn_a to computing node cn_b, the historical status of v_t is to be synchronized from cn_a to cn_b. Adjusting stateful nodes is costlier and more complex than adjusting stateless nodes. The historical status of v_t is also stored in the sliding window belonging to v_t.

C. Vertex parallelization

Parallelism degree [9,28] pd_{v_i,t_i} of vertex v_i at t_i is determined by the function of vertex, the available resource of computing node, and the input rate of data tuples for the vertex at time t. All pd_{v_i} instances of vertex v_i run on pd_{v_i} computing nodes at the same time.

For an instance DAG, the parallelism degree of each vertex in the instance DAG should meet the following constraints (22).

$$pd_{v_i}\colon pd_{v_j}\colon \cdots pd_{v_e} = \frac{T_{v_i}}{C_{cn_{v_i}}}\colon \frac{T_{v_j}}{C_{cn_{v_j}}}\colon \cdots \frac{T_{v_e}}{C_{cn_{v_e}}}, \tag{22}$$

where $v_i, v_j, \ldots, v_i \in V(DAG)$. $T_{v_i}, T_{v_j}, \ldots, T_{v_e}$ are the average time to process each data tuple of $v_i, v_j, \ldots, v_i$, respectively. $C_{cn_{v_i}}, C_{cn_{v_j}}, \ldots, C_{cn_{v_e}}$ are the average CPU consumption to process each data tuple of $v_i, v_j, \ldots, v_i$ on computing nodes $cn_{v_i}, cn_{v_j}, \ldots, cn_{v_e}$, respectively.

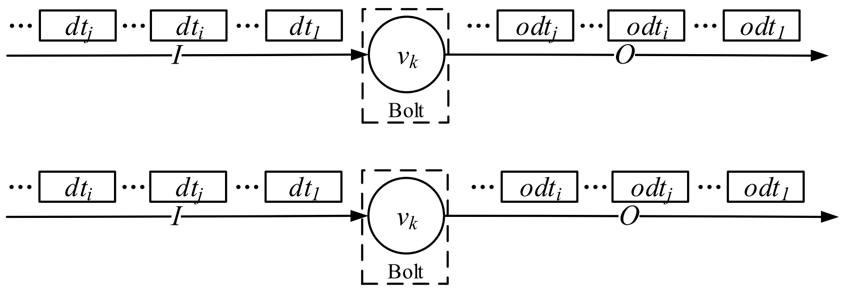

Fig. 11. Stateless vertex semantic.

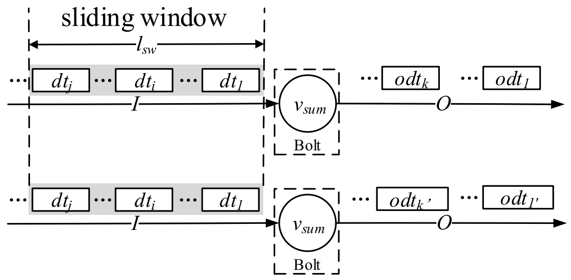

Fig. 12. Stateful vertex semantic.

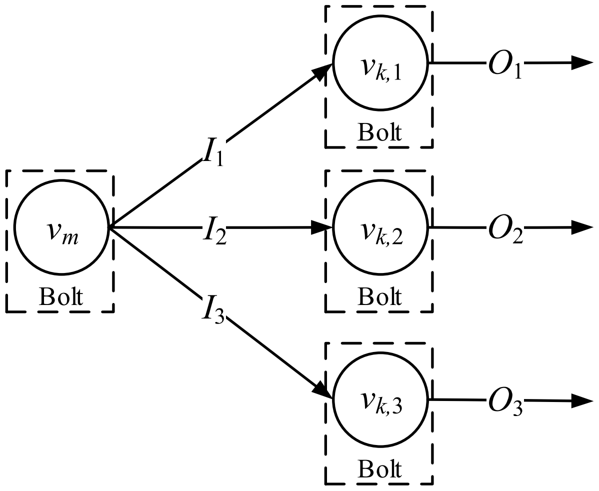

Fig. 13. Stateless vertex parallelization.

In the scheduling state, the input rate of data tuples for a vertex is directly determined by the input rate of source data stream. The input rate change Δir_s of source data stream at a different time can be described by (23).

$$\Delta ir_s = ir_{s,t_i} - ir_{s,t_j}, \tag{23}$$

where ir_{s,t_i} and ir_{s,t_j} are the input rate of source data stream at time t_i, and t_j, respectively, and $t_i > t_j$.

If $\Delta ir_s > 0$ and $\Delta ir_s > \alpha \cdot ir_{s,t_i}$, $\alpha \in [0, 1]$, then the parallelism degree pd_{v_i,t_i} of vertex v_i at t_i needs to be increased according to (24).

$$pd_{v_i,t_j} = (1+\alpha) \cdot pd_{v_i,t_i} \tag{24}$$

where α is the adjustment parameter, which can be set according to the system needs.

If $\Delta ir_s < 0$ and $\Delta ir_s < \alpha \cdot ir_{s,t_i}$, then the parallelism degree pd_{v_i,t_i} of vertex v_i at t_i needs to be decreased according to (25).

$$pd_{v_i,t_j} = (1-\alpha) \cdot pd_{v_i,t_i} \tag{25}$$

When an instance DAG is submitted to the data center, parallelism degree pd_{v_s,t_s} of sources vertex v_s at the submission time t_s is set to $k \cdot pd_{v_s,t_s}$, $k \in \{1, 2, \ldots\}$, where k can be calculated by (26), and $pd_{v_s,t_s} = k \cdot pd_{v_s,t_s}$. The parallelism degree of other vertices at submission time t_s can be determined according to the constraints in (22).

$$k = \max\left(1, \left\lfloor \frac{\sum_{i=0}^{n} num\left(c_{cn_i}\right)}{\sum_{v_j \in set(DAG)} pd_{v_j}} \right\rfloor\right), \tag{26}$$

where $num\left(c_{cn_i}\right)$ is the number of CUP resources available on the ith computing node cn_i.

For a stateless vertex v_k, all the task instances of v_k process data tuples independently, as shown in Fig. 13. The state of each instance does not need to be shared among the siblings.

However, for stateful vertex v_t, all the task instances of v_t process data tuples dependently [29], as shown in Fig. 14. In order to get global and dependency information of a data stream over a period of time, the state of each instance belonging to the same stateful vertex v_t needs to be summarized. In this scenario, a new vertex $v_{t'}$ for summarizing those states of all instances ofv_t also needs to be created.

D. Initial DAG scheduling with runtime-awareness

In the initial DAG scheduling phase, a modified first-fit [30] based runtime-aware data tuple scheduling strategy (FFRA) is proposed, which is described in Algorithm 1.

Algorithm 1: modified First-Fit based Runtime-Aware data tuple scheduling algorithm.

```
1.  Input: a logical DAG, a data tuple $dt_k = (Key_k, Value_k, ts_k)$ in a data stream, current
    available capacity matrix $C_{v_{n\times m}}$ of computing nodes in the data center.
2.  Output: modified First-Fit based Runtime-Aware data tuple scheduling.
3.  if the logical DAG or the number of available computing nodes is null then
4.      Return null.
5.  end if
6.  for each vertex in the logical DAG do
7.      Determine the state of vertex $v_i$ in the logical DAG according to its function.
8.      if $v_i$ is a stateful vertex then
9.          Create a new vertex $v_{i'}$ for sharing the states of all instances of $v_i$.
10.     end if
11. end for
12. for each vertex in the logical DAG do
13.     Determine the parallelism degree of each vertex in the instance DAG according to the
        constraints of (22) and the initial value of the input vertex
14. end for
15. Sort $n$ computing nodes in the data center by the available CPU resources in descending
    order as the first factor, and sort available memory resources in descending order as
    the second factor.
16. Insert all the vertices of the instance DAG into the un-scheduled queue.
17. Set the scheduling queue to null.
18. while the un-scheduled queue is not null do
19.     Fetch a vertex $v_i$ from the un-scheduled queue with the feature of in-degree of $v_i$ being 0.
20.     for each instance of $v_i$ then
21.         Select an available computing node with the feature of earliest finishing time to
            run the instance of $v_i$.
22.         To keep all instances of $v_i$ running on different computing nodes.
23.         Update the current available resources of the selected computing node.
24.     end for
25.     Set $v_i$ as the scheduled vertex of the instance DAG, and insert $v_i$ to the scheduling queue.
26.     Delete $v_i$ from the un-scheduled queue.
27.     Update the in-degree of successor vertex of $v_i$.
28.     Select an instance of $v_i$ to process data tuple $dt_k$. One or many new data tuples
        $dt_{new}$ are created by $v_i$, and sent to the successor vertex of $v_i$.
29. end while
30. return modified First-Fit based Runtime-Aware data tuple scheduling.
```

The Input of this algorithm includes an application logical DAG, a data tuple dt_k in a data stream, and a matrix $C_{v_{n\times m}}$ that denotes the current available capacity of the computing nodes in the data center. The output is the modified First-Fit based Runtime-Aware data tuple scheduling. Step 6–step 11 determine the state of each vertex in the logical DAG, and create a new vertex for sharing the states of all instances of the stateful vertex. Step 12–step 14 determine the parallelism degree of each vertex. Step 18–step 29 schedule the instance DAG to the data center based on the modified first-fit based runtime-aware data tuple scheduling strategy.

The main stage of the initial DAG scheduling with data tuple runtime-awareness is shown in Fig. 15. At stage 1, a logical DAG consists of v_s, v_i, v_j, and v_e, which is submitted to the data center. The state and parallelism degree of each vertex in the logical DAG is analyzed and determined. For v_j, two instances are created, which are $v_{j,1}$ and $v_{j,2}$. For other vertices, only one instance is created. At stage 2, a data tuple dt_1 is to be processed by the instantiated instance DAG. At stage 3, the in-degree of v_s is 0, so v_s is scheduled to a computing node. Data tuple dt_1 is being processed by v_s. At stage 4, two new data tuples dt_2 and dt_3 are created by v_s, and sent to v_i and $v_{j,2}$, respectively. As usual, if a vertex has more than one instances, only one instance is selected to process the data tuple at one time. In order to schedule the successor vertex of v_s, the in-degree of v_i, $v_{j,1}$, and $v_{j,2}$ are updated (changing from the solid line to the dotted line), as shown at stage 5. At stage 6, a new data tuple dt_5 is created by v_e, and all vertices of the instance DAG are scheduled to the compute nodes properly.

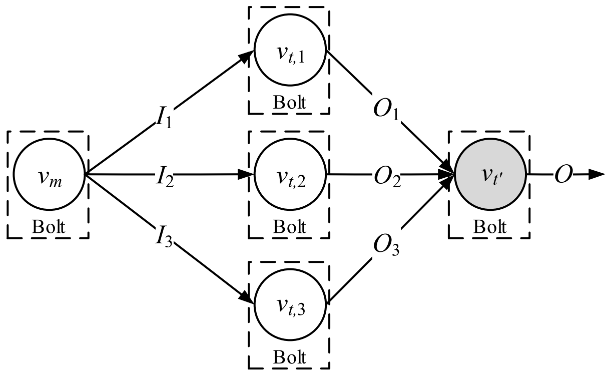

Fig. 14. Stateful vertex parallelization.

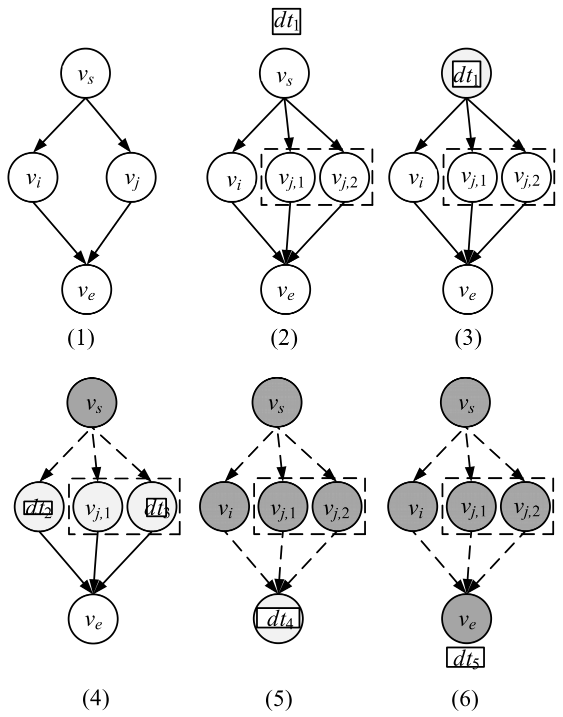

Fig. 15. Main stages of initial DAG scheduling with data tuple runtime-awareness.

E. Online DAG rescheduling with runtime-awareness

If the response time is higher or the throughput is lower than the users' expectation, some vertices of the running instance DAG need to be rescheduled online to improve the system performance.

In the online DAG rescheduling phase, a maximum latency-sensitive [31] based runtime-aware data stream scheduling strategy (MSRA) is employed, which is described in Algorithm 2.

Algorithm 2: Maximum latency-sensitive based runtime-aware data stream scheduling algorithm.

```
1.  Input: scheduling state of the online instance DAGs, current available capacity matrix $C_{v_{n\times m}}$ of computing nodes in data center, the input rate of a data stream.
2.  Output: maximum latency-sensitive based runtime-aware data stream scheduling.
3.  if the instance DAG $G$ or the number of computing nodes is null then
4.      Return null.
5.  end if
6.  Monitor the real-time arrival rate of a data stream and the makespan of each online instance DAG.
7.  while makespans of an instance DAG to be increased or a computing node is in high load or low load state do
8.      for each makespan of an instance DAG to be increased do
9.          Select a vertex with the maximum latency in the instance DAG.
10.         Create a new instance of the vertex on a computing node that can most reduce the latency of the vertex.
11.         Re-adjust data streams among all instances of the vertex.
12.     end for
13.     for each computing node in high load state do
14.         Select one or many vertices with the maximum delay from all vertices running on the computing node, and meet the load constraint.
15.         Reschedule one or many vertices to other computing nodes that meet the constraint of makespan of their respective instance DAGs.
16.     end for
17.     for each computing node in low load state do
18.         Reschedule all vertices running on the computing node to other computing nodes that meet the constraints of makespan of their respective instance DAGs.
19.         Shut down this low load computing node.
20.     end for
21.     Monitor the real-time arrival rate of a data stream and the makespan of each online instance DAG.
22.     Update the resource state of each computing node in the data center
23. end while
24. return maximum latency-sensitive based runtime-aware data stream scheduling.
```

The input of this algorithm includes the scheduling state of online instance DAGs, the current available capacity matrix $C_{v_{n\times m}}$ of computing nodes in the data center, and the input rate of a data stream. The output is the maximum latency-sensitive based runtime-aware data stream scheduling. Step 7–step 23 monitor the real-time arrival rate of a data stream and the makespan of each online instance DAG, and reschedule all those instance DAGs by the maximum latency-sensitive based runtime-aware data stream scheduling strategy.

In this algorithm, all the online instance DAGs are rescheduled as a whole. For a vertex, its runtime latency is considered instead of its location in an instance DAG. This greatly reduces the system decision-making time and therefore guarantees the performance. In most cases, the rescheduled vertex is the best choice, however, it is possible that a "wrong" vertex is selected. In the latter case, such "mistake" is temporary. In the long run, the chosen vertex is still able to contribute to performance improvement. It makes this algorithm simple and efficient.

Table 1
Hardware configuration of the cluster.

Computing node	CPU	Memory	Bandwidth
Storm Nimbus, Storm UI	Intel core (TM) i7-4790, 3.6 GHz, 6-core	8GB DDR4 3000 MHz	1 Gbps
Zookeeper 1∼3	Intel core (TM) i7-4790, 3.6 GHz, 6-core	4GB DDR3 1600 MHz	1 Gbps
Supervisor 1∼8	Intel core (TM) i5-8400, 2.8 GHz, 6-core	4GB DDR3 1600 MHz	1 Gbps
Supervisor 9∼16	Intel core (TM) i5-8400, 2.8 GHz, 6-core	4GB DDR3 1600 MHz	1 Gbps
Supervisor 17∼24	Intel core (TM) i3-8100, 3.6 GHz, 4-core	2GB DDR3 1600 MHz	1 Gbps
Supervisor 24∼32	Intel core (TM) i3-8100, 3.6 GHz, 4-core	2GB DDR3 1600 MHz	1 Gbps

Table 2
Software configuration of the Sra-Stream platform.

Software	Version
OS	CentOS 6.3 64 bit
Storm	Apache-storm-1.0.2
JDK	Jdk1.7 64 bit
Zookeeper	Zookeeper-3.4.6
Python	Python 2.7.2
Zeromq	Zeromq-2.1.7

6. Performance evaluation

This section evaluates the performance of the proposed Sra-Stream in a distributed computing environment. We firstly discuss the experimental environment and its parameter settings, then provide a detailed analysis of performance evaluation results.

A. Experimental environment and parameter setup

The proposed Sra-Stream system is developed as an extension on Storm 1.0.2 [18] [32], and installed on top of CentOS 6.3. Extensive experiments have been conducted on a computing cluster located at computer architecture laboratory in China University of Geosciences, Beijing. The cluster consists of 36 machines, with 1 designated machine serving as the master node, running Storm Nimbus, 3 designated as Zookeeper node, and the rest 32 machines working as Supervisor nodes. The hardware configuration of the cluster is shown in Table 1.

The software configuration of Sra-Stream platform is shown in Table 2.

We submit three types of logical DAGs — TOP_N, WordCount, and real-time user portrait to the computing cluster. The logic graph of TOP_N is shown in Fig. 2. The logic graph of WordCount is shown in Fig. 16, where the function of vertex v_r is a word reader, the function of vertex v_n is a word normalizer, and the function of vertex v_c is a word counter. The logic graph of real-time user portrait is shown in Fig. 17, where the function of each

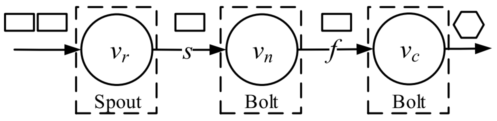

Fig. 16. Logical graph of WordCount in Sra-Stream.

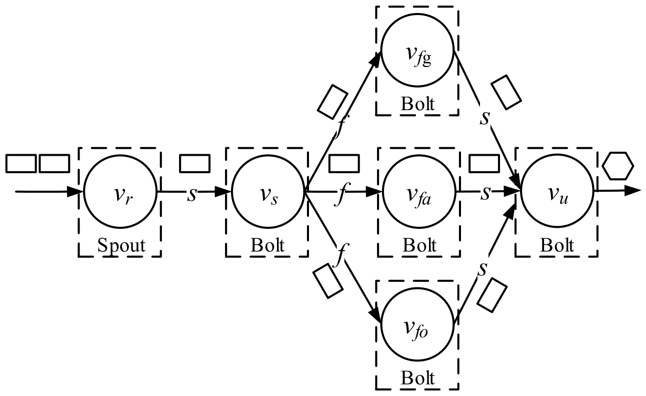

Fig. 17. Logical graph of real-time user portrait in Sra-Stream.

Table 3
Function of each vertex in logic graph of real-time user portrait.

Vertex	Function
v_r	Reader of search keyword
v_s	Word segmentation
v_{fg}	Feature extraction of user's gender
v_{fa}	Feature extraction of user's age
v_{fo}	Feature extraction of user's occupation
v_u	Construction of user portraits

vertex is shown in Table 3. The number of instance DAGs for each type is initialized to 10. The size of sliding window for spout vertices is set to 10,000 tuples and for bolt vertices it is set to 5000 tuples.

There are two types of data sets. The first type is text data collected from a book, so its input rate can be effectively controlled to observe the system response time, system throughput, and CPU utilization under different testing scenarios. Another type is the result of a real-time keyword searching on a social networking site, where the time span of the search is set to 24 h, and each data tuple is organized in chronological order. The first data set is used for TOP_N and WordCount and the second data set is used for the application of real-time user portrait.

B. Performance results

The experimental setting contains three evaluation parameters: system response time *RT*, system throughput *ST*, and average CPU utilization u_{avg} (*CPU*).

(1) *Response time*. System response time *RT* or makespan M_{sys} of an elastic stream computing system is defined by (8), which is considered to be acceptable by users if it stays at a millisecond level. On Storm platform, *RT* can be obtained through the Storm UI. The shorter the system response time is, the better the real-time performance of the elastic stream computing system would be.

When the input rate of data is stable, Sra-Stream has a better system response time as compared to the default scheduling strategy on Storm platform. As shown in Fig. 18, with the rate set at 1000 tuples/s, the average response time of Sra-Stream and that of the default Storm scheduling strategy at the stable stage

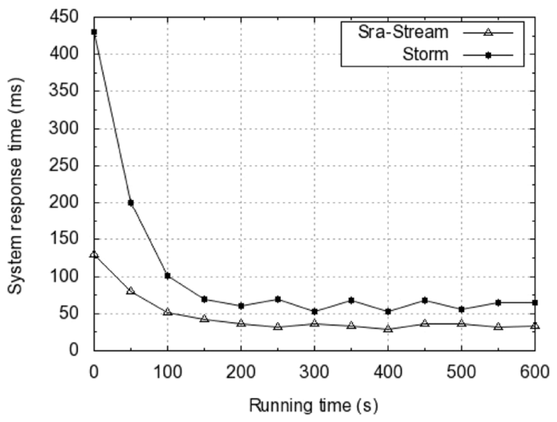

Fig. 18. System response time with data rates 1000 tuples/s.

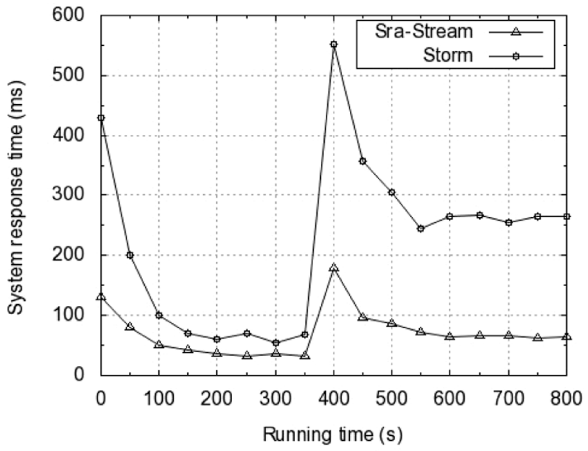

Fig. 19. System response time with data rates changing from 1000 tuples/s to 2000 tuples/s at 400.

are gauged at 32 ms and 61 ms, respectively. It is obvious that the average response time by Sra-Stream is significantly shorter than that of the default Storm scheduling strategy when the input rate is stable.

When the input rate of data is fluctuating over time, Sra-Stream has a better system response time as compared to the default strategy on Storm platform. As shown in Fig. 19, as data rates change from 1000 tuples/s to 2000 tuples/s at 400 s, that is, the data rate is 1000 tuples/s in [0, 400] s, and the data rate is 2000 tuples/s in [400, 800] s, the average response time by Sra-Stream is changing from 32 ms to 63 ms, whereas the average response time by default Storm scheduling strategy is changing from 61 ms to 253 ms. The average response time by Sra-Stream is significantly shorter than that of the default Storm scheduling strategy when the input rate is fluctuating over time.

When we select the real-time keyword searching data set, the input rate of data is fluctuating over time. Sra-Stream has a better system response time as compared to the default strategy on Storm platform. As shown in Fig. 20, at different points in time, the system response time is fluctuating along with the input rate. But at every point in time, the response time of Sra-Stream is shorter than that of Storm platform. The variances of response time fluctuation are also smaller.

(2) *System throughput*. System throughput reflects the overall processing ability for all running instance DAGs, which is evaluated by the number of output tuples per second of per instance DAG. The greater the system throughput, the stronger the data processing ability of the stream computing system.

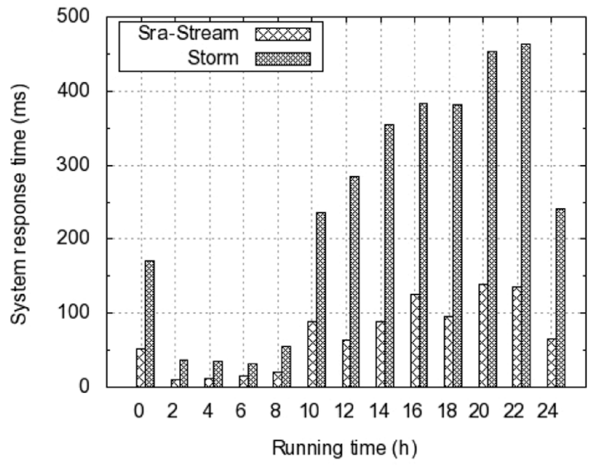

Fig. 20. System response time of real-time user portrait with a real-world data stream.

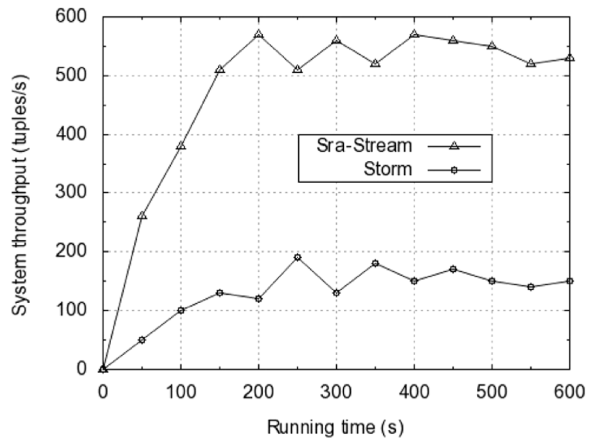

Fig. 21. System throughput with data rates 1000 tuples/s.

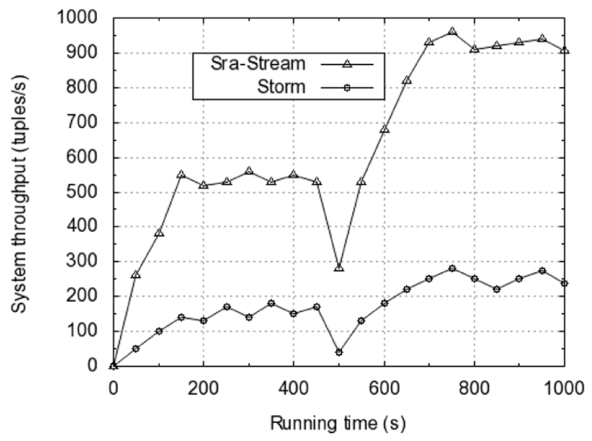

Fig. 22. System throughput with data rates changing from 1000 tuples/s to 2000 tuples/s at 500.

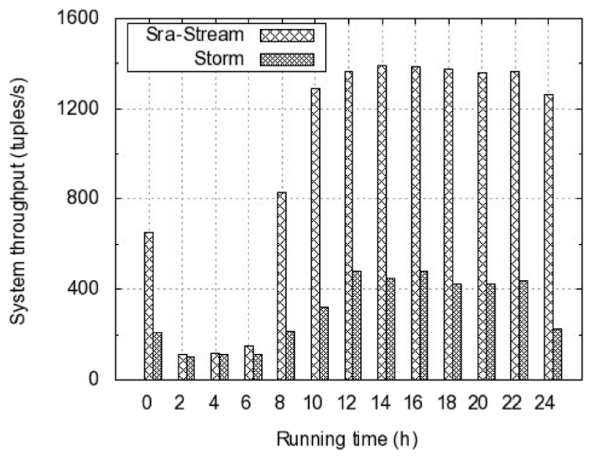

Fig. 23. System throughput of real-time user portrait with a real-world data stream.

When the input rate of data is stable, Sra-Stream has a higher system throughput as compared to the default strategy on Storm platform. As shown in Fig. 21, with the rate set at 1000 tuples/s, the average throughput by Sra-Stream and by default scheduling Storm strategy at the stable stage are 537 tuples/s and 156 tuples/s, respectively. It is proved that the average throughput by Sra-Stream is higher than that of the default Storm scheduling strategy when the input rate is stable.

When the input rate of data is fluctuating over time, Sra-Stream has a better system response time as compared to the default strategy on Storm platform. As shown in Fig. 22, as the data rate changes from 1000 tuples/s to 2000 tuples/s at 500, that is, the data rate is 1000 tuples/s in [0, 500] s, and the data rate is 2000 tuples/s in [500, 1000] s, the average throughput of the instance DAGs by Sra-Stream is changing from 537 tuples/s to 921 tuples/s, whereas the average throughput of the instance DAGs by default Storm scheduling strategy is changing from 156 tuples/s to 262 tuples/s. The average throughput by Sra-Stream is significant than that of the default Storm scheduling strategy when the input rate is fluctuating over time.

Sra-Stream also has a higher system throughput as compared to the default strategy on Storm platform. As shown in Fig. 23, at different points in time, the input rate is fluctuating over time, so the system throughput is also fluctuating. When the input rate is at a lower level and the system is in a lower load state, the difference of system throughput between the two strategies is not particularly noticeable. When the rate is at a higher level and the system is in a higher load state, the difference in system throughput between the two strategies is significant.

(3) *Average CPU utilization* $u_{avg}\,(CPU)$. CPU utilization reflects the effective use and overhead of the CPU of computing node during a period of time. The average CPU utilization $u_{avg}\,(CPU)$ in $[0, t]$ can be calculated by (27).

$$u_{avg}\,(CPU) = \frac{1}{n}\sum_{i=0}^{n} u_{cn_i}\,(CPU)\,, \tag{27}$$

where $u_{cn_i}\,(CPU)$ is the average CPU utilization of the ith computing node cn_i in $[0, t]$.

A good scheduling strategy keeps the $u_{avg}\,(CPU)$ at a high level, and no computing node is experiencing a high CPU load or low CPU load for a long period of time. In this experiment, we set $\delta_{\max(CPU)} = 0.9$, and $\delta_{\max(CPU)} = 0.2$.

When the input rate of data is stable, with the increase of the number of instance DAGs, the average CPU utilization in a computing cluster also increases. Sra-Stream strategy has a more efficient CPU utilization as compared to the default strategy on Storm platform in the same situation. As shown in Fig. 24, when the rate is set at 1000 tuples/s, and the number of instance DAGs is at a low level, the average CPU utilization by Sra-Stream strategy is higher than that of the default strategy on Storm platform.

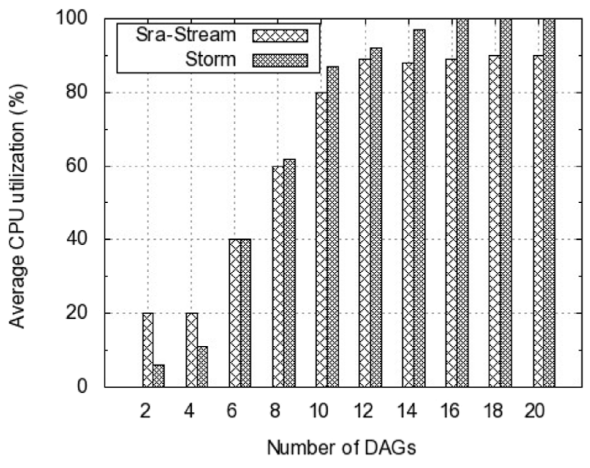

Fig. 24. Average CPU utilization with different numbers of instance DAGs.

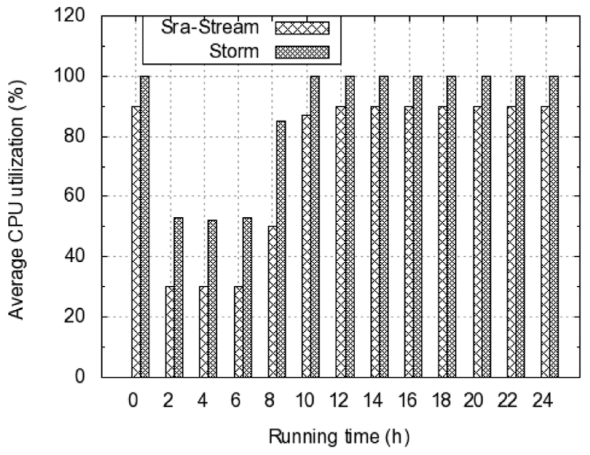

Fig. 25. Average CPU utilization of real-time user portrait with a real-world data stream.

This is because some computing nodes with low CPU load were shut down. When the number of instance DAGs is at a higher level, the average CPU utilization by Sra-Stream strategy can be controlled at a high and efficient level, however, the average CPU utilization by default strategy on Storm platform is always in an overload state, and this has directly affected the performance of the system.

Sra-Stream also has a more efficient CPU utilization as compared to the default strategy on Storm platform. As shown in Fig. 25, when the input rate is at a higher level and the system is in an overload state, the average CPU utilization produced by the default strategy on Storm platform is clocked at 100%, which causes the system performance to be unstable and degraded. However, the average CPU utilization produced by Sra-Stream strategy can be controlled at a reasonable level to ensure the system performance.

7. Related work

There are three broad categories of related works: (1) state management for stream computing systems, (2) runtime-aware scheduling in distributed systems, and (3) application scheduling on Storm platform.

A. State management for stream computing system

State management [12–14] plays a key role in stream computing systems as each vertex has different states at different time. It is fully considered in many popular stream computing systems, such as, Flink [4] which maintains data states that have been processed over time on each stateful vertex and provides exactly-once semantics for the stateful vertex. Samza [5] also manages snapshotting and restoration of the state of a stateful vertex.

In [17], a mechanism that supports stateful migration of application components is proposed. The stateful migration mechanism achieves the rescheduling of stateful vertices to different computing nodes, and enables the computing system to change the topology deployment by preserving the state of stateful vertices at run-time.

In [29], the authors introduced a set of state access patterns suitable for managing accesses to states in stream computing systems. Two orthogonal dimensions are used to characterize the behavior of stateful streaming computations, which are the nature of the state — divided into partitionable state and non-partitionable state, and the type of state access — divided into ordered state access and relaxed state access. The definition, implementation and performance evaluation of six state access patterns are given, which include serial state access pattern, all-or-none state access pattern, fully partitioned state access pattern, separate task/state function state access pattern, accumulator state access pattern, and successive approximation state access pattern.

In [33], a scheduler for parallel state machine replication is proposed, and a novel command handling and dependency tracking mechanism that favors high throughput in parallel state machine replication is present.

In [34], a lightweight state-management abstraction for big stateful computation is proposed, and a distributed system named ChronoStream is implemented for elastic stateful stream processing in a multi-tenant environment. The internal states in each operator are classified into computation state and configuration state.

In [35], parallel patterns for window-based stateful vertex are presented. Features of parallel patterns in relation to the distribution policy are identified. An internal state and the role of parallel executors in the window management are presented, and four patterns for window-based stateful vertex are further described.

To summarize, State management for stream computing system is considered in many works. However, most of them consider the state of a vertex from a qualitative perspective, in our work, we consider the state of a vertex from a quantitative perspective.

B. Runtime-aware scheduling in distributed systems

Runtime-aware scheduling in distributed systems is to schedule long-running applications on available computing nodes in a data center at the same time, while satisfying user's specified SLAs constraints. It is difficult to find an optimal schedule for precedence constraint-based directed acyclic graph at runtime because the arrival rate of a data stream fluctuates over time, and the amount of available computing resources also fluctuates over time.

In [30], a runtime-aware adaptive scheduling mechanism in stream computing is proposed. The scheduling applies to the task redistribution to accelerate the processing of the stage operators, applies to the availability of resources variation, and applies to the fluctuating data input rates. Stateless vertex is considered in the runtime-aware scheduling mechanism, but stateful vertex is not considered.

The estimate of running time for parallel jobs is an important factor in scheduling. In [36], a set of running time adjustment schemes are proposed, which can be directly used in production environments. A job scheduling strategy with adjusted runtime estimates on production supercomputers is given.

Lazy scheduling is a runtime scheduler for task-parallel codes. It does not maintain any additional state to infer system load, and it does not make irrevocable serialization decisions. In [37], two alternative approaches of lazy scheduling are proposed. How they scale on three different multicores platform is also shown.

Many devices have large demands on running real-time applications. In [38], a transition-aware runtime task scheduling strategy for multicore processors is proposed, and an integer linear programming model is constructed to find the optimal solutions for off-line scheduling.

Machine learning approaches are widely used in classification methods for data mining applications. However, the time-consuming training process greatly limits the efficiency of machine learning approaches. In [39], a runtime data layout scheduling for machine learning data set is proposed to improve the efficiency of machine learning approaches.

To summarize, the aforementioned solutions provide a valuable insight into the challenges and potential solutions for runtime-aware scheduling problem in distributed systems. However, in Big Data Era, novel approaches that address the particular challenges and opportunities of these technologies need to be developed, and some characteristics specific to big data stream computing environments need to be considered when developing online scheduling strategies.

C. Application scheduling on storm platform

The scheduling strategy of instance DAG in Strom plays a key role in improving system performance. In Storm 1.1.0, there are four kinds of built-in schedulers, which are namely DefaultScheduler, IsolationScheduler, MultitenantScheduler, and ResourceAwareScheduler. Some work [7,17,23] has been done to improve the application scheduling strategy on Storm platform.

In [40], a preventive auto-parallelization approach for elastic stream computing is proposed. The approach relys on a metric which estimates operator activity in the future. Parallelism degree of each operator can be increased or decreased according to the local and global information.

In [41], a slot-aware scheduling strategy for even scheduler in Storm is designed. The scheduling strategy achieves a fine-grained EvenScheduler using the slot-aware sorting queue and merger factor, and evenly allocates slots for multiple applications in a load balancing cluster.

In [42], a queue theory approach to the modeling of data streams is proposed. The optimization problem of application is defined to minimize the total resources required, and a corresponding algorithm is also proposed to mitigate the complexity order of the optimization problem of application

In [43], a replication-based state management system is proposed. The system actively manages multiple state backups on different computing nodes. The prototypes are built on Storm platform by extending its monitoring and recovery modules.

To summarize, current application scheduling on Storm platform is limited to one or a few aspects. It is obvious that this problem has been studied extensively over the years, and will continue to be the focus of research due to its theoretical significance and practical importance. In this paper, however, we also take account of the scheduling strategy on Strom platform.

Additionally, our past work [25] focused on achieving a fair scheduling among multiple applications and giving an elastic online scheduling framework for multiple online applications in stream computing systems. Different from the above work, this work aims at shortening the gap of system throughput and input data stream. Our primary goal is not fairness but system throughput, considering the vertex state and runtime-aware scheduling for online applications, which maximizes system throughput, guarantees system response time, and achieves elastic stream computing.

8. Conclusions and future work

Low system response time and high system throughput are two critical requirements for a stream computing system. Application scheduling is the key to achieve those goals, which focuses on how to schedule tasks to computing nodes while satisfying a set of objective constraints. Application scheduling problem is also one of the most thought-provoking NP-hard problems. Data streams arrive in real time, and demands to be processed immediately. An elastic stream computing system is always needed through the elastic adjustment of computing resources and flexible adjustment of vertex parallelism.

To minimize system response time, maximize system throughput, and effectively use resources, a fundamental requirement is a state and runtime-aware scheduling for elastic stream computing systems. It would be able to determine when and how to reschedule running vertices of a instance DAG according to a fluctuating data stream. To achieve this goal, we first get the state of each vertex in each instance DAG, obtain a clear picture of the changing status of data streams and system resources, and then perform running-aware scheduling for the critical vertices of the instance DAG. The rescheduling of the critical vertices helps minimize system response time and maximize system throughput.

Our future work will be focusing on the following directions:

(1) Developing a complete state and runtime-aware scheduling framework based on Sra-Stream as a part of elastic stream computing services.

(2) Deploying the Sra-Stream in a real big data stream computing environment for production applications.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61602428 and 61602106; the Key Laboratory of Geological Information Technology, Ministry of Natural Resources of the People's Republic of China; the Fundamental Research Funds for the Central Universities under Grant No. 2652018081; and Australian Research Council (ARC) Discovery Project.

References

[1] S. Imai, S. Patterson, C.A. Varela, Maximum sustainable throughput prediction for data stream processing over public clouds, in: Proc. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, IEEE Press, 2017, pp. 504–513.

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, Storm@twitter, in: Proc. 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD 2014, ACM Press, 2014, pp. 147–156.

[3] M. Zaharia, T. Das, H.Y. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proc. 24th ACM Symposium on Operating Systems Principles, SOSP 2013, ACM Press, 2013, pp. 423–438.

[4] C. Paris, E. Stephan, F. Gyula, H. Seif, R. Stefan, T. Kostas, State management in apache flink:® consistent stateful distributed stream processing, Proc. VLDB Endow. 10 (12) (2017) 1718–1729.

[5] S.A. Noghabi, K. Paramasivamy, Y. Pany, N. Rameshy, J. Bringhursty, I. Gupta, R.H. Campbell, Samza: Stateful scalable stream processing at linkedin, Proc. VLDB Endow. 10 (12) (2017) 1634–1645.

[6] C. Li, J. Zhang, Y. Luo, Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm, J. Netw. Comput. Appl. 87 (2017) 100–115.
[7] T. Li, J. Tang, J. Xu, Performance modeling and predictive scheduling for distributed stream data processing, IEEE Trans. Big Data 2 (4) (2016) 353–364.
[8] Hadoop, http://hadoop.apache.org/.
[9] B. Gedik, S. Schneider, M. Hirzel, K.L. Wu, Elastic scaling for data stream processing, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1447–1463.
[10] Y. Xu, K. Li, L. He, L. Zhang, K. Li, A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems, IEEE Trans. Parallel Distrib. Syst. 26 (12) (2015) 3208–3222.
[11] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, R. Grimm, A catalog of stream processing optimizations, ACM Comput. Surv. 46 (4) (2014) 46, 1-34.
[12] V. Cardellini, F. Lo Presti, M. Nardelli, G.R. Russo, Decentralized self-adaptation for elastic data stream processing, Future Gener. Comput. Syst. 87 (2018) 171–185.
[13] G. Mencagli, M. Torquati, M. Danelutto, Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams, Future Gener. Comput. Syst. 79 (Part 3) (2018) 862–877.
[14] T. De Matteis, G. Mencagli, Proactive elasticity and energy awareness in data stream processing, J. Syst. Softw. 127 (C) (2017) 302–319.
[15] T. De Matteis, G. Mencagli, Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing, in: Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, ACM Press, 2016, p. 13.
[16] A. Shukla, Y. Simmhan, Model-driven scheduling for distributed stream processing systems, J. Parallel Distrib. Comput. 117 (2018) 98–114.
[17] V. Cardellini, M. Nardelli, D. Luzi, Elastic stateful stream processing in storm, in: Proc. 2016 International Conference on High Performance Computing & Simulation, HPCS 2016, IEEE Press, 2016, pp. 583–590.
[18] Storm, http://storm.apache.org.
[19] M.A. Lopez, A.G.P. Lobato, O.C.M.B. Duarte, A performance comparison of open-source stream processing platforms, in: Proc. 59th IEEE Global Communications Conference, GLOBECOM 2016, IEEE Press, 2016, 7841533, 1–6.
[20] Zookeeper, https://zookeeper.apache.org/.
[21] J. Ghaderi, S. Shakkottai, R. Srikant, Scheduling storms and streams in the cloud, in: Proc. 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2015, ACM Press, 2015, pp. 439–440.
[22] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, G. Buttazzo, Schedulability analysis of conditional parallel task graphs in multicore systems, IEEE Trans. Comput. 66 (2) (2017) 339–353.
[23] G. Lucarelli, F. Mendonca, D. Trystram, A new on-line method for scheduling independent tasks, in: Proc. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, IEEE Press, 2017, pp. 140–149.
[24] R. Pathan, P. Voudouris, P. Stenstrom, Scheduling parallel real-time recurrent tasks on multicore platforms, IEEE Trans. Parallel Distrib. Syst. 29 (4) (2018) 915–928.
[25] D. Sun, H. Yan, S. Gao, X. Liu, R. Buyya, Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams, J. Supercomput. 74 (2) (2018) 615–636.
[26] H. Li, J. Wu, Z. Jiang, X. Li, X. Wei, Task allocation for stream processing with recovery latency guarantee, in: Proc. 2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, IEEE Press, 2017, pp. 379–383.
[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] T. Lorido-Botran, J. Miguel-Alonso, J.A. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, J. Grid Comput. 12 (4) (2014) 559–592.
[29] M. Danelutto, P. Kilpatrick, G. Mencagli, M. Torquati, State access patterns in stream parallel computations, Int. J. High Perform. Comput. Appl. (2017) http://dx.doi.org/10.1177/1094342017694134.
[30] Y. Liu, X. Shi, H. Jin, Runtime-aware adaptive scheduling in stream processing, Concurr. Comput.: Pract. Exper. 28 (14) (2016) 3830–3843.
[31] L. Yang, J. Cao, H. Cheng, Y. Ji, Multi-user computation partitioning for latency sensitive mobile cloud applications, IEEE Trans. Comput. 64 (8) (2015) 2253–2266.
[32] J. Zhang, C. Li, L. Zhu, Y. Liu, Yanpei 1, The real-time scheduling strategy based on traffic and load balancing in storm, in: Proc. the 18th IEEE International Conference on High Performance Computing and Communications, HPCC 2016, IEEE Press, 2017, pp. 372–379.
[33] O.M. Mendizabal, R.S. Moura, F.L. Dotti, Efficient and deterministic scheduling for parallel state machine replication, in: Proc. the IEEE 31st International Parallel and Distributed Processing Symposium, IPDPS 2017, IEEE Press, 2017, pp. 748–757.
[34] Y. Wu, K.L. Tan, ChronoStream: Elastic stateful stream computation in the cloud, in: Proc. 2015 IEEE 31st International Conference on Data Engineering, ICDE 2015, IEEE Press, 2015, pp. 723–734.
[35] T. De Matteis, G. Mencagli, Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach, Int. J. Parallel Program. 45 (2) (2017) 382–401.
[36] W. Tang, N. Desai, D. Buettner, Z. Lan, Job scheduling with adjusted runtime estimates on production supercomputers, J. Parallel Distrib. Comput. 73 (7) (2013) 926–938.
[37] A. Tzannes, G.C. Caragea, U. Vishkin, R. Barua, Lazy scheduling: A runtime adaptive scheduler for declarative parallelism, ACM Trans. Program. Lang. Syst. 36 (3) (2014) 10, 1–51.
[38] W.Y. Shieh, C.C. Pong, Energy and transition-aware runtime task scheduling for multicore processors, J. Parallel Distrib. Comput. 73 (9) (2013) 1225–1238.
[39] Y. You, J. Demmel, Runtime data layout scheduling for machine learning dataset, in: Proc. 46th International Conference on Parallel Processing, ICPP 2017, IEEE Press, 2017, pp. 452–461.
[40] R.K. Kombi, N. Lumineau, P. Lamarre, A preventive auto-parallelization approach for elastic stream processing, in: Proc. IEEE 37th International Conference on Distributed Computing Systems, ICDCS 2017, IEEE Press, 2017, pp. 1532–1542.
[41] W. Qian, Q. Shen, J. Qin, D. Yang, Y. Yang, Z. Wu, S-Storm: A slot-aware scheduling strategy for even scheduler in Storm, in: Proc. 18th IEEE International Conference on High Performance Computing and Communications, HPCC 2016, IEEE Press, 2016, pp. 623–630.
[42] S. Vakilinia, X. Zhang, D. Qiu, Analysis and optimization of big-data stream processing, in: Proc. the 59th IEEE Global Communications Conference, GLOBECOM 2016, IEEE Press, 2016, 7841598, 1-6.
[43] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, R. Buyya, E-storm: Replication-based state management in distributed stream processing systems, in: Proc. the 46th International Conference on Parallel Processing, ICPP 2017, IEEE Press, 2019, pp. 571–580.

Dawei Sun is an associate professor at 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 finished the Postdoctoral position research at the department of computer science and technology of Tsinghua University, China in 2015. He leads the research group of parallel and distributed systems. His current researches interests include big data computing, cloud computing, and distributed systems. He has authored or co-authored over 60 journal and conference papers in the above areas.

Shang Gao received her Ph.D. degree in computer science from Northeastern University, Shenyang, China in 2000. She is currently a Lecturer at the School of Engineering and Information Technology, Deakin University, Geelong, Australia. Her current research interests include distributed collaboration, adaptive learning, and cloud computing.

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 and now works as a postdoc researcher at the CLOUDS lab. His research interests include stream processing and distributed systems.

Fengyun Li is an associate professor at the School of Computer Science and Engineering, Northeastern University, P.R. China. She received her Ph.D. degree in computer architecture from Northeastern University, China in 2013. Her current researches interests include distributed systems, network security, privacy preserving, cloud computing.

Xinqi Zheng is a professor at the School of Information Engineering, China University of Geosciences, Beijing, P.R. China, and the head of the school. His research interests include big data and data analytics. He has authored or co-authored over 200 journal and conference papers.

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 650 publications and four text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 121, 77,400+ citations). He has 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.