Future Generation Computer Systems 112 (2020) 193–208

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Dynamic redirection of real-time data streams for elastic stream computing

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

[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] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia
[d] Beijing Key Laboratory of Internet Culture and Digital Dissemination Research, Beijing Information Science & Technology University, Beijing, 100101, China
[e] Polytechnic Center for Territory Spatial Big-data, MNR of China, China

ARTICLE INFO

Article history:
Received 3 September 2019
Received in revised form 8 March 2020
Accepted 15 May 2020
Available online 22 May 2020

Keywords:
Data stream redirection
Stream computing
Elastic processing
Load balancing
Distributed system
Big data

ABSTRACT

An elastic stream computing system needs elastic adjustment of computing resource allocation and vertex parallelism to improve latency and throughput, which includes continuously or periodically scaling in/out the workload of computing nodes at runtime. Dynamic redirection can help with this elasticity issue by dynamically redirecting real-time data streams to computing resources. Due to the time-varying and unpredictable nature of real-time data streams, implementing redirection of data streams is challenging. Currently, the requirements of data streams redirection are not fully fulfilled, which directly affects the latency and throughput of stream computing systems. To bridge this gap, we proposed a dynamic redirection framework (called Dr-Stream) for elastic stream computing systems. This paper discussed the following aspects: (1) Investigating the dynamic redirection of real-time data streams, providing a general stream application model, a data stream model and a data stream grouping model, as well as formalizing the problem of load balancing optimization and data stream redirection. (2) Redirecting data streams among multiple instances of an operator at runtime by a lightweight load balancing strategy to improve the load balancing of a data center at the vertex level. Managing system states, especially the states of stateful operators by a logical ring-based strategy to improve accuracy. (3) Determining the number of instances for each operator, and deploying the instance(s) to computing nodes by a modified first-fit strategy at runtime. (4) Evaluating the fulfillment of low latency, high throughput, and load balancing objectives in a real-world distributed stream computing environment. Experimental results showed that Dr-Stream reduced the average system latency and load balancing of the data center by more than 20% and 15%, respectively. It also improved the average system stability by more than 15% and avoided over-utilization of computing nodes, as compared to the existing strategies in Storm.

1. Introduction

In the big data era, stream computing is an on-the-fly computing paradigm [1] that processes and reliably extracts valuable insights from high-velocity continuous data streams in timely manner. A stream computing system is an instance of the stream computing paradigm, fulfilling the requirements of streaming applications and processing the dynamic and volatile data streams in real-time. An elastic stream computing system achieves low latency and high throughput by continuously or periodically scaling in/out the workload of computing nodes at runtime in a distributed computing environment.

One of the major challenges to implement elastic stream computing is how to adaptively adjust streaming applications to the available computing resources in real-time. As shown in Fig. 1, the input rate of a data stream continuously changes over time, which can be divided into 5 stages. At each stage, the deployment status of the vertices and the allocated computing resources need to be adjusted on-the-fly. This adjustment needs to be precise and effective. It is preferable to make the system performance less fluctuating, so the adjustment duration should be as short as possible. As shown at stage 2, given a short time window, if the

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

https://doi.org/10.1016/j.future.2020.05.021
0167-739X/

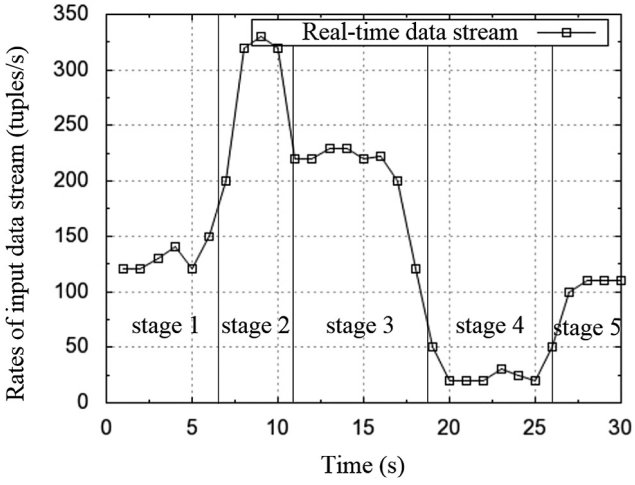

Fig. 1. Real-time data stream at different input rates.

adjustment duration is longer than the window, the adjustment solution will not be deployed in time, and the computing environment might undergo significant changes due to the delay. This kind of adjustments may only increase the system load without delivering any performance improvement. However, if there were no adjustments being made, some of the data tuples at stage 2 would be discarded if the current resources could not meet the requirements imposed by the increasing volume of data streams. The consequence of load shedding is also unacceptable. How to avoid the loss of status information or streaming data during online processing is another major challenge to realize elastic stream computing. While performing the online adjustment process, it is also challenging to migrate some of the operators from one computing node to another at the right time. There are a few factors to be taken into account, e.g. whether the operator states and/or data tuples would be lost during the migration process, especially for the stateful operators.

To address these challenges, researchers have been working on new generation of elastic stream computing systems, trying to build low latency and high throughput environments for real-time data stream applications. Most of these work [2,3] tries to optimize system performance from the perspective of resource adjustment (rescheduling operator among computing nodes). However, the problem of operator scheduling in distributed big data stream computing systems is also one of the most thought-provoking NP-hard problems in general cases [4]. A rescheduling scheme with performance optimization might not work well due to the fact that the rescheduling duration is usually long, and the stream applications may have undergone substantial changes because of data stream fluctuation. In addition, the loss of state and data tuples during operator rescheduling has not been fully studied, and continuous rate fluctuations in real-time data streams are common. Therefore, in our study, we try to optimize system performance from the perspective of data stream redirection. We adjust the processing load of computing resources, reducing the possibilities of data stream and state loss, and effectively optimizing system performance. Fine-grained resource optimization is also considered to further improve performance metrics.

A dynamic redirection framework should be able to determine when and how data streams should be redirected with regard to specific elastic stream computing requirements. To achieve this goal, we need to know the rates of real-time data streams, the distribution of data tuples among instances of the same vertex, the current load of each computing node, and the proper strategies that can improve system performance. Currently, the research on dynamic redirection of fluctuating data stream for elastic stream computing has not fully addressed the concern [5,6]. This creates the need for investigating a dynamic redirection framework for elastic stream computing, processing data stream in a scalable and elastic manner with low latency and high throughput.

Our work is motivated by the observation that poor performance is mainly caused by heavyweight rescheduling strategies trying to optimize system performance from the perspective of online resources adjustment. They are usually not the best option for stream computing systems. Significant system performance fluctuations may occur as the result of large-scale, frequent online rescheduling. As such, our goal is to improve system performance with a lightweight load balancing strategy from the perspective of data stream redirection at the vertex level. It suits elastic stream computing systems and provides a scalable way to deal with fluctuating data streams.

1.1. Paper contributions

Contributions of our work include the following:

(1) Investigate the dynamic redirection of real-time data streams over distributed stream computing systems, providing a general stream application model, a data stream model and a data stream grouping model, as well as formalizing the problem of load balancing optimization and data stream redirection.

(2) Redirect data streams among multiple instances of an operator at runtime with a lightweight load balancing strategy that improves load balancing performance at the vertex level; managing the system state, especially the states of stateful operators by a logical ring-based strategy to improve the accuracy.

(3) Determine the number of instances for each operator, and deploying the instances on computing nodes by a modified first-fit strategy throughout the runtime of a stream application.

(4) Evaluate the fulfillment of low latency, high throughput, and load balancing objectives with two types of streaming applications Top_N and WordCount in a real-world distributed stream computing environment.

(5) Implement a prototype software system, called Dr-Stream, with performance evaluations. Experimental results conclusively demonstrate that the proposed Dr-Stream provides significant performance improvements on system latency, throughput and load balancing metrics.

1.2. Paper organization

The rest of the paper is organized as follows: In Section 2, we introduce 8 kinds of built-in data stream grouping modes on Storm platform. Section 3 describes the Dr-Stream model, including its stream application model, data stream model, and data stream grouping model. Section 4 formalizes the problem of load balancing within a data center, as well as the problem of data stream redirection. Section 5 focuses on the system architecture, the redirection among instances, the state management, and the deployment optimization in Dr-Stream. Section 6 discusses the experimental environment and parameter settings, and analyzes performance evaluation results of Dr-Stream. Section 7 reviews the related work on big data stream computing, optimization of load balancing in elastic stream computing systems, as well as stream application deployment on Storm platform. Finally, conclusions and future work are given in Section 8.

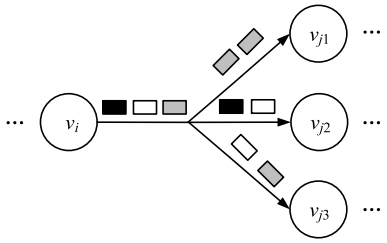

Fig. 2. Shuffle grouping.

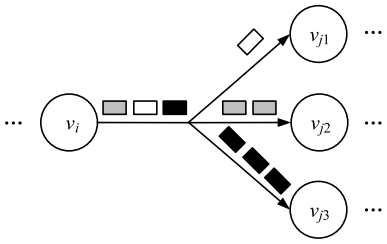

Fig. 3. Fields grouping.

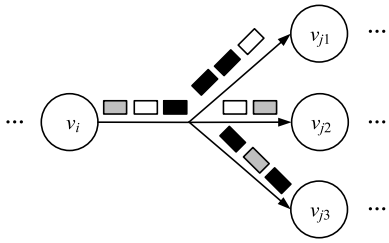

Fig. 4. Partial key grouping.

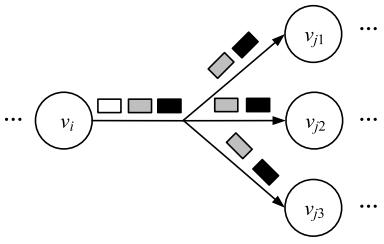

Fig. 5. All grouping.

2. Background

A data stream grouping strategy determines how to partition the output data tuples of an upstream vertex to the relevant downstream vertices. In the process of defining an application topology, the data stream grouping strategy between upstream and downstream vertices needs to be clarified one by one, which is an important part of the application topology. Storm [7], as one of the most popular distributed stream computing systems, provides eight types of built-in data stream grouping modes, namely shuffle grouping, fields grouping, partial key grouping, all grouping, global grouping, none grouping, direct grouping, and local grouping [8].

(1) Shuffle grouping

Shuffle grouping partitions the data tuples from an upstream vertex to the relevant downstream vertices in a round-robin way. All relevant downstream vertices will get output data tuples one by one, with each receiving the same number of data tuples. As shown in Fig. 2, all those output data tuples of v_i are stored in an output queue, and will be partitioned to vertices v_{j1}, v_{j2} and v_{j3} consecutively. Shuffle grouping is one of the most common grouping strategies for real-time data streams partition. It partitions data streams in a lightweight manner, which suits most application scenarios. However, the heterogeneity of downstream vertices is not considered.

(2) Fields grouping

Fields grouping partitions the data tuples from an upstream vertex to the relevant downstream vertices by one or many fields' value of the output data tuples. The data tuples with the same fields' value will be partitioned to the same downstream vertex. As shown in Fig. 3, all those output data tuples of v_i are stored in an output queue, and will be partitioned to vertices v_{j1}, v_{j2} or v_{j3} according to its fields' value. v_{j1}, v_{j2} or v_{j3} will process data tuples with different fields' values, respectively. Fields grouping is also one of the most common grouping strategies. It partitions data streams by fields' value, and data tuples with the same fields' value are always partitioned to the same downstream node. It also suits many application scenarios, but the heterogeneous downstream vertices are not considered.

(3) Partial key grouping

Partial key grouping partitions the data tuples from an upstream vertex to the relevant downstream vertices by one or many key values of the output data tuples. The data tuples with the same key value will be partitioned to the specific downstream vertex or vertices, similar to the Fields grouping. In addition, the load balancing mechanism is also primitively considered between the downstream vertices to improve the system adaptability to fluctuating data streams.

As shown in Fig. 4, all those output data tuples of v_i are stored in an output queue, and will be partitioned to vertices v_{j1}, v_{j2} or v_{j3} according to its key values. v_{j1} and v_{j2} will process white data tuples, v_{j1} and v_{j3} will process black data tuples, and v_{j2} and v_{j3} will process gray data tuple. The number of data tuples to be processed by each pair of vertices is not necessarily the same, which can be adjusted according to the load of each vertex to achieve load balancing between the two vertices of the pair.

Based on the Fields grouping, partial key grouping adds a primitive load balancing mechanism between the two paired vertices, improving the elastic adaptation to dynamic data streams to a certain extent.

(4) All grouping

All grouping replicates each data tuple from an upstream vertex to each of the downstream vertex. As shown in Fig. 5, all those output data tuples of v_i are stored in an output queue, and will be replicated to vertices v_{j1}, v_{j2} and v_{j3}. v_{j1}, v_{j2} and v_{j3} will receive the same replicas of output data tuples for further processing. All grouping can be used for full-scale multidimensional data processing, and downstream nodes perform simultaneous processing from different dimensions on the same set of data tuples, which is useful in high-dimensional data processing.

(5) Global grouping

Global grouping partitions the data tuples from one or many upstream vertices to a specific downstream vertex. The specific downstream vertex can be specified according to computational semantics. As shown in Fig. 6, all the output data tuples of v_i and v_k are stored in their output queue, respectively, and will only be sent to vertex v_{j1} for further processing. No output data tuples will be partitioned to vertex v_{j2}, though it is also a downstream vertex of v_i and v_k. Global grouping can be used to aggregate data from multiple upstream vertices to a single downstream node to

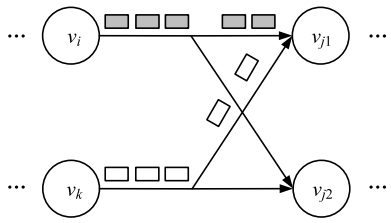

Fig. 6. Global grouping.

process global data. It is particularly useful in reducing topology size.

(6) Direct grouping, None grouping, and Local grouping

Direct grouping partitions each data tuple from an upstream vertex to a specific downstream vertex, determined by the upstream vertex. None grouping, and Local grouping are similar to Shuffle grouping to some extent. Due to the length limit of the paper, their details are not discussed. For more information, please refer to [8].

All of those data stream grouping modes are designed with the idea of partitioning data tuples from upstream to downstream. All those modes are static, which means once it is selected, it will be applied throughout the lifespan of the application. It cannot dynamically adapt to the fluctuation of data streams, or take into account the current load of each node. On Storm platform, a custom stream grouping mode can be implemented via the CustomStreamGrouping interface [8], which introduces a certain level of flexibility into the partition process. In this paper, we focus on dynamically redirecting real-time data streams on Strom platform. A lightweight load balancing strategy is employed to redirect real-time data streams among multiple instances of an operator. A sub-optimal deployment scheme can be obtained by a modified first-fit strategy at runtime, further improving the scalability and the performance of the system.

3. System model

Before introducing the proposed system and the relevant algorithms, we first explain the formalization of fundamental models in an elastic stream computing system, including the stream application model, the data stream model, and the data stream grouping model.

3.1. Stream application model

A stream application is defined by user and submitted to a stream computing system. The function of a stream application [9] can be formalized as a directed acyclic graph $G = (V(G), E(G))$, where $V(G) = \{v_i | i \in 1, \ldots, n\}$ is a finite set of n vertices, vertex $v_i \in V(G)$ is an operator with a specific function, $E(G) = \left\{e_{v_i,v_j} | v_i, v_j \in V(G)\right\}$ is a finite set of directed edges, and edge $e_{v_i,v_j} \in E(G)$ represents a data stream path from vertex v_i to vertex v_j.

The complexity of vertex v_i is determined by the functions it implements, and can be measured by the time complexity and space complexity of the corresponding computing algorithm v_i implements. The complexity of vertex v_i is an important factor during the process of allocating instance and resource to v_i. Vertex v_i can be further categorized into stateful or stateless vertex according to whether there is a dependency relationship between the current and precedent data tuples of v_i [10]. For a stateful vertex, there is a state dependency between two adjacent data tuples. For stateless vertex, there is no state dependency.

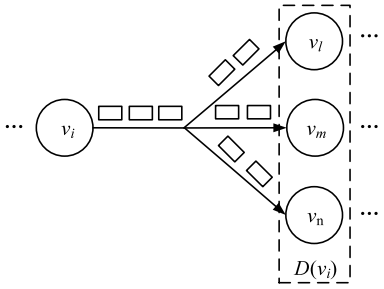

Fig. 7. Downstream vertices set of v_i.

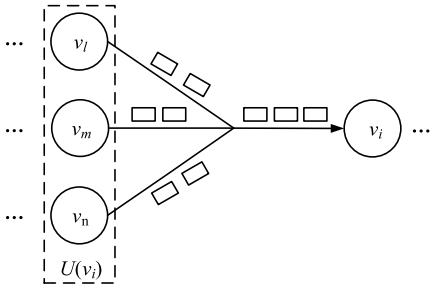

Fig. 8. Upstream vertices set of v_i.

Vertex v_i emits its output data stream to its downstream vertices set $D(v_i)$. $n_{D(v_i)} = |D(v_i)|$ is the number of downstream vertices of v_i, and $n_{D(v_i)} \geq 0$. If $n_{D(v_i)} = 0$, then v_i is a sink vertex of stream application G, which means it has no downstream vertex in G. The output data tuples of v_i will be partitioned or replicated into $n_{D(v_i)}$ sub-streams, and become the input data streams to downstream vertex set $D(v_i)$. As shown in Fig. 7, $D(v_i) = \{v_l, v_m, v_n\}$ is the downstream vertex set of v_i, and $n_{D(v_i)} = 3$.

Vertex v_i receives its input data stream from its upstream vertices set $U(v_i)$. $n_{U(v_i)} = |U(v_i)|$ is the number of upstream vertices of v_i, and $n_{U(v_i)} \geq 0$. If $n_{U(v_i)} = 0$, then v_i is source vertex of stream application G, which means it has no upstream vertex in G. Each vertex of $U(v_i)$ outputs a data stream to v_i. All data steams aggregate as one single input data stream to v_i. As shown in Fig. 8, $U(v_i) = \{v_l, v_m, v_n\}$ is the upstream vertex set of v_i, and $n_{U(v_i)} = 3$.

When a stream application G is submitted and running in a stream computing system, one or many instances of vertex v_i in G are created according to the complexity of vertex v_i, rate of input data stream, and available computing resources [11]. All instance vertices of G and their data dependency relationships form a runtime graph of G.

3.2. Data stream model

A data stream ds is composed of a sequence of data tuples in the form of a continuous stream, denoted as $ds = \{dt_1, dt_2, \ldots, dt_i, \ldots\}$. The ith data tuple dt_i can be characterized by a three-tuple $dt_i = (key_i, value_i, ts_i)$, where key_i, $value_i$ and ts_i are key, value and timestamp of the ith data tuple dt_i, respectively. Each vertex has at least one input data stream and/or one output data stream. As shown in Fig. 9, $ds_{i(I)}$ and $ds_{i(O)}$ are the input data stream and output data stream of vertex v_i, respectively.

In a data stream ds, the order of all data tuples can be obtained by the timestamp of each data tuple. However, when a vertex has multiple input data streams, one or more data streams may

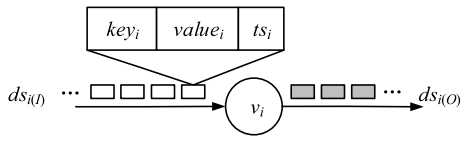

Fig. 9. Data stream.

come from a heterogeneous computing environment. In this situation, it is extremely difficult to sort out the order of all data tuples globally as if they originated from the same homogeneous environment. In this paper, we do not consider the potential difference brought by heterogeneousness [12] as it involves specific semantics that are beyond the scope of this paper. Instead, it is assumed that the same data stream would produce the same result in the computing process.

The rate r_{ds} of data stream ds is one of the most important factors [13], affecting the performance of an elastic stream computing system. It is always in a dynamic state as a data stream continuously fluctuates over time. When multi-source data streams aggregate at vertex level, the input rate ir_{v_i} of vertex v_i is the sum of all input data stream rates of vertex v_i. It can be described as (1).

$$ir_{v_i} = \sum_{k=1}^{n} r_{v_i,k(I)}, \tag{1}$$

where $r_{v_i,k(I)}$ is the kth input data stream rates, n is the number of input data streams of vertex v_i.

3.3. Data stream grouping model

Each vertex needs to clarify the grouping strategy for its output data tuples, and then decides how to partition them to the relevant downstream vertices. According to whether a data tuple is replicated, a grouping strategy can be roughly categorized as replication grouping strategy or a non-replication grouping strategy.

For a replication grouping strategy, each data tuple in the output data stream $ds_{i(O)}$ of vertex v_i is replicated at the replication point, which can be set to any time point after the data tuple leaves v_i and before its transmission downwards. After replication at the replication point, data tuples forms n output data streams $ds_{i(O)}(1), ds_{i(O)}(2), \ldots, ds_{i(O)}(n)$, where n is determined by the downstream vertices of v_i, and $n = n_{D(v_i)}$. The data tuples are replica of the output data stream $ds_{i(O)}$, that is $\forall dt_i \in ds_{i(O)}(k), k \in \{1, 2, \ldots, n\}$, then $\exists dt_i \in ds_{i(O)}(j), j \in \{1, 2, \ldots, n\}$, and $\exists ds_{i(O)}(k) = ds_{i(O)}(j)$,. As shown in Fig. 10, each data tuple of vertex v_i is replicated 3 times by partition function $f\left(ds_{i(O)}\right) = r(dt(n = 3))$, and forms 3 output data streams for each downstream vertex of v_i. On Storm platform, all grouping strategy is a replication grouping strategy.

For a non-replication grouping strategy, each data tuple in the output data stream $ds_{i(O)}$ of vertex v_i is emitted to one of downstream vertices of v_i by partition function $f\left(ds_{i(O)}\right)$, which is implemented according to specific grouping semantics. The output data stream $ds_{i(O)}$ of vertex v_i is partitioned into n input data streams, given n downstream vertices of v_i. That is $\forall dt_i \in ds_{i(O)}(k), k \in \{1, 2, \ldots, n\}$, then $\exists dt_i \in ds_{i(O)}$, and $ds_{i(O)} = ds_{i(O)}(1) \cup ds_{i(O)}(2) \cup \cdots \cup ds_{i(O)}(n)$, described as (2). Each data tuple only exists in one of input data streams of n downstream vertices of v_i. All those n sub-streams are non-overlapping. That is if $\forall dt_i \in ds_{i(O)}(k)$, then $\nexists dt_i \in ds_{i(O)}(j), k \neq j$, and $ds_{i(O)}(k) \cap ds_{i(O)}(j) = \varnothing, k \neq j$. It can be described as (3).

$$ds_{i(O)} = \bigcup_{k=1}^{n} ds_{i(O)}(k).k \in \{1, 2, \ldots, n\} \tag{2}$$

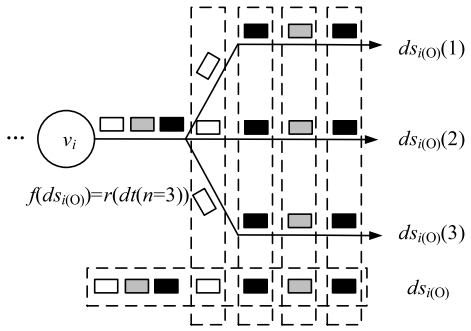

Fig. 10. Replication based grouping model.

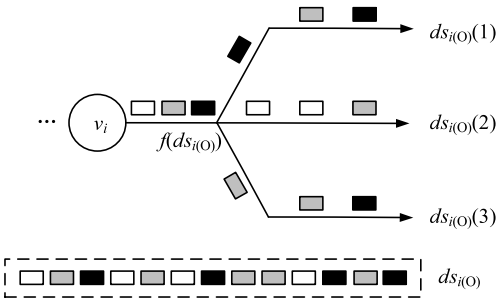

Fig. 11. Non-replication based grouping model.

$$\bigcap_{k=1}^{n} ds_{i(O)}(k) = \varnothing, k \in \{1, 2, \ldots, n\}. \tag{3}$$

As shown in Fig. 11, the output data stream $ds_{i(O)}$ of vertex v_i is partitioned into three sub-streams by the partition function $f\left(ds_{i(O)}\right)$. One data tuple of output data stream $ds_{i(O)}$ only belongs to one of the three sub-streams. On Storm platform, shuffle grouping, fields grouping, partial key grouping, global grouping, none grouping, direct grouping, and local grouping are non-replication grouping strategies, with different partition functions though.

4. Problem formalization

In this section, we formalize the problems of load balancing of a data center, its optimization and data stream redirection before applying them in the proposed Dr-Stream in Section 5.

4.1. Load balancing problem of a data center

The load $l_{cn,[t_s,t_e]}$ of a computing node cn for running vertex set $V_{cn,[t_s,t_e]}$ over a period of time $[t_s, t_e]$ can be evaluated by the computing node's CPU queue state during $[t_s, t_e]$. It directly reflects the amount of load on computing node cn, described as (4).

$$l_{cn,[t_s,t_e]} = \sum_{v_i \in V_{cn,[t_s,t_e]}} n_{v_i,cn,[t_s,t_e]}, \tag{4}$$

where $n_{v_i,cn,[t_s,t_e]}$ is the number of data tuples of vertex v_i during $[t_s, t_e]$, and $v_i \in V_{cn,[t_s,t_e]}$.

The load ratio $lr_{cn,[t_s,t_e]}$ of computing node cn during $[t_s, t_e]$ can be calculated by (5).

$$lr_{cn,[t_s,t_e]} = \frac{l_{cn,[t_s,t_e]}}{length_{cn}} = \frac{\sum_{v_i \in V_{cn,[t_s,t_e]}} n_{v_i,cn,[t_s,t_e]}}{length_{cn}}, \tag{5}$$

where $length_{cn}$ is the length of CPU queue of computing node cn.

The value of load ratio $lr_{cn,[t_s,t_e]}$ of computing node cn during $[t_s, t_e]$ should be greater than 0. If $lr_{cn,[t_s,t_e]} > 1$, then computing node cn during $[t_s, t_e]$ is overloaded, which means the CPU queue is full, and the new incoming data tuples will be discarded. If $lr_{cn,[t_s,t_e]} \in [0, 1]$, the load of cn during $[t_s, t_e]$ is acceptable, and the larger the load ratio $lr_{cn,[t_s,t_e]}$, the more the node cn is utilized.

Let $DC = \{cn_1, cn_2, \ldots, cn_{num}\}$ be a data center with num computing nodes. Its load balancing $lb_{DC,[t_s,t_e]}$ metric during $[t_s, t_e]$ can be evaluated by (6).

$$lb_{DC,[t_s,t_e]} = \frac{1}{num} \cdot \sum_{k=1}^{num} \left| lr_{cn_k,[t_s,t_e]} - \overline{lr_{[t_s,t_e]}} \right|, \tag{6}$$

where num is the number of computing nodes in data center DC, $cn_k \in DC$, $k \in [1, num]$, $\overline{lr_{[t_s,t_e]}}$ is average load ratio of all num computing nodes in DC.

The load balancing metric $lb_{DC,[t_s,t_e]}$ of a data center DC during $[t_s, t_e]$ is one of the key measures [14] to evaluate and improve the performance of an elastic stream computing system. $lb_{DC,[t_s,t_e]} \in [0, 1]$. If $lb_{DC,[t_s,t_e]} = 0$, then DC is absolutely load balanced during $[t_s, t_e]$, which is hard to achieve; If $lb_{DC,[t_s,t_e]} = 1$, then DC is absolutely load unbalanced during $[t_s, t_e]$, which significantly leads to unbalanced use of computing nodes in DC and seriously affects the system performance. The less the load balancing metric $lb_{DC,[t_s,t_e]}$, the better the load status of the DC. $lb_{\min,DC}$ and $lb_{\max,DC}$ are the minimum and maximum load balancing metrics of DC, respectively, which are specified according to the needs of the system. Usually, $lb_{DC,[t_s,t_e]}$ should satisfy the constraint $lb_{DC,[t_s,t_e]} \in \left[lb_{\min,DC}, lb_{\max,DC}\right]$ to keep the load balancing of DC acceptable.

4.2. Load balancing optimization

Load balancing optimization for stream computing is an online optimization problem [15,16]. It aims at finding a schedule s: $V(G) \rightarrow DC$ and a data stream partition p: $ds \rightarrow V(G)$, which help maximize system throughput and minimize response time, optimizing resource use and load balancing metric $lb_{DC,[t_s,t_e]}$ of DC during $[t_s, t_e]$ without overloading any single computing node, and satisfying user's specified SLA constraints on system latency and throughput.

For a data center $DC = \{cn_1, cn_2, \ldots, cn_{num}\}$, let $G_{sa} = \{G_1, G_2, \ldots, G_m\}$ be a set of m stream applications. $\{r_{ds}(G_1), r_{ds}(G_2), \ldots, r_{ds}(G_m)\}$ are the rate of input data stream of $\{G_1, G_2, \ldots, G_m\}$, respectively. The load balancing optimization problem of all m stream applications G_{sa} in the DC is then formalized as follows:

$$\min\left(\max_{G_i \in G_{sa}} l(G_i)\right), \tag{7}$$

subject to

$$0 \le l(G_i) \le l_{\max}(G_i), \forall G_i \in G_{sa}, \tag{8}$$

and,

$$\min lb_{DC,[t_s,t_e]}, \tag{9}$$

subject to

$$0 < lr_{\min,cn_i} \le lr_{cn_i,[t_s,t_e]} \le lr_{\max,cn_i} < 1, \forall cn_i \in DC, \tag{10}$$

in which $l(G_i)$ is the latency of the ith stream application in the set of G_{sa}. $l_{\max}(G_i)$ is the maximum latency of the ith application. $lr_{\min,cn_i}$ and $lr_{\max,cn_i}$ are the minimum and maximum load ratio of the ith computing node cn_i, respectively. Both are user-specified SLAs constraints.

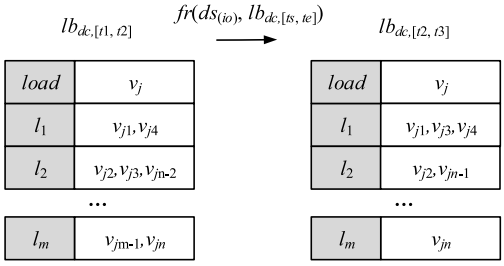

Fig. 12. Data stream redirection.

4.3. Data stream redirection

Multiple data tuples $\{dt_1, dt_2, \ldots\}$ of data stream $ds_{i(O)}$ can be redirected from vertex v_i to n instances $\{v_{j1}, v_{j2}, \ldots, v_{jn}\}$ of v_j by redirection function $fr()$, where v_j is the direct downstream vertex of v_i, such that a data tuple can be redirected among multiple instances of v_j. Redirection function is a mapping fr: $ds_{i(O)} = \{dt_1, dt_2, \ldots\} \rightarrow \{v_{j1}, v_{j2}, \ldots, v_{jn}\}$. Usually, redirection function $fr\left(ds_{(io)}, lb_{DC,[t_s,t_e]}\right)$ can be described as (11).

$$fr\left(ds_{(io)}, lb_{DC,[t_s,t_e]}\right) = \begin{cases} f\left(ds_{(io)}\right), \\ \quad if\ lb_{DC,[t_s,t_e]} \in \left[lb_{\min,DC}, lb_{\max,DC}\right], \\ f\left(lb_{DC,[t_s,t_e]}\right), otherwise. \end{cases} \tag{11}$$

where $f\left(ds_{i(O)}\right)$ is a partition function and employed for data stream grouping, and $f\left(lb_{DC,[t_s,t_e]}\right)$ is a load based redirection function among multiple instances of a vertex for load balancing improvement.

As shown in Fig. 12, in the redirection process, the mapping between data tuples with the same key and the instances of downstream vertices v_j is readjusted for load balancing optimization.

5. Dr-Stream: Architecture and algorithms

Based on the above theoretical analysis, we have proposed and developed Dr-Stream, a dynamic redirection framework for elastic stream computing systems. To provide an overview of the framework, this section discusses its overall structure, including the system architecture, the redirection among instances, the state management, and the deployment optimization.

5.1. System architecture

The system architecture of Dr-Stream includes four stages: topological construction, instantiation, scheduling, and redirection and rescheduling, as shown in Fig. 13.

In the topological construction stage, the logical topology of a stream application is designed by user. The application function determines the structure of the topology. The semantics [17] between any pair of upstream and downstream vertices determines the choice of data stream grouping strategy. Once the topology of the stream application is constructed, the user can submit it to a stream computing environment, e.g. Storm platform. The topology will keep running until being terminated manually or interrupted due to environmental failure.

In the instantiation stage, to improve the topological structure and the load balancing of each vertex, the number of instances for each vertex is determined by first analyzing its computational complexity, then consequently one or more instances are created for each vertex. The functionality and semantics are the same for all instances of the same vertex. Therefore, the existence of multiple instances is also an important prerequisite

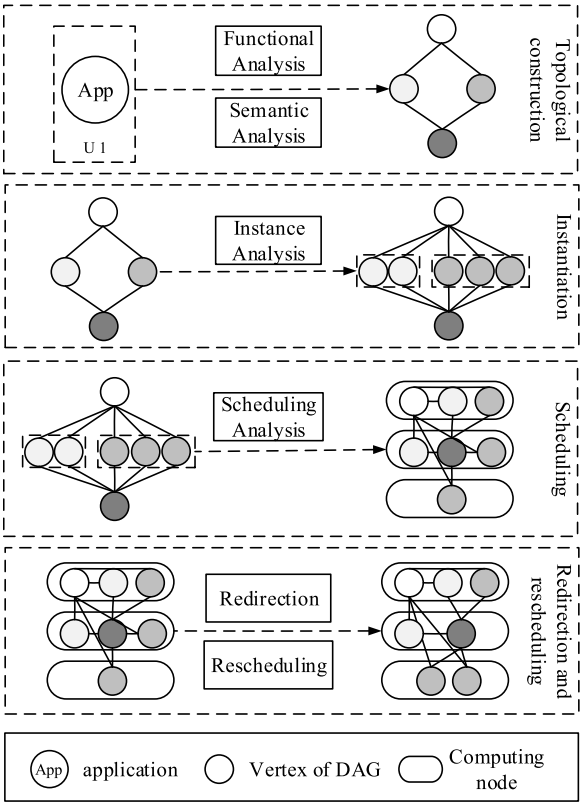

Fig. 13. Dr-Stream architecture.

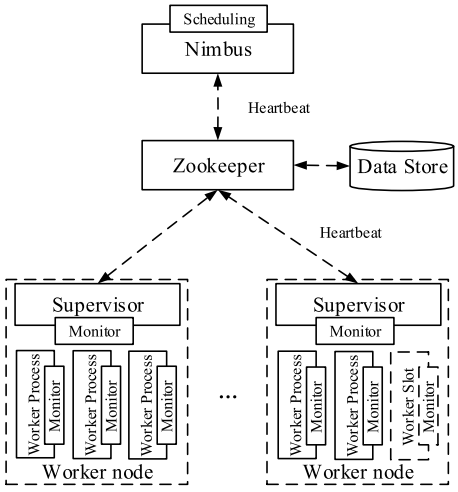

Fig. 14. Dr-Stream topology.

for dynamic redirection of real-time data stream [18]. On Storm platform, the specific stream grouping modes can be customized by implementing the CustomStreamGrouping interface [8].

In the scheduling stage, the instantiated topology is deployed onto available computing nodes of *DC*, which is determined by a specified scheduling strategy. The strategy aims at maximizing throughput and minimizing response time, while optimizing resource use and load balancing without overloading any single computing node. In order to improve the throughput and fault tolerance of a streaming application, multiple instances of a vertex are often deployed to different computing nodes. On Storm platform, the scheduling strategy can be customized by implementing the IScheduler interface and specified through the configuration file Storm.yaml where the deployment strategy is employed.

With the fluctuating rate of data stream and the dynamic change of available resources in *DC*, in the redirection and rescheduling stage, real-time data stream is redirected among multiple instances of a vertex and instances are redeployed among available computing nodes for load balancing and performance purposes. During this process, it is unwise to make the system to fluctuate too much [19,20], so all decisions need to factor in the current status of data stream partition among multiple instances of a vertex and the instance deployments on the computing nodes. In addition, the number of instances for each vertex can be further adjusted as needed.

As shown in Fig. 14, the topology of Dr-Stream system consists of one Nimbus, a few Zookeepers, and a bunch of Supervisors. Nimbus first receives stream application and instantiates each vertex in the topology, then it deploys instances to workers on appropriate Supervisors as determined by the specified scheduling strategy. Each supervisor executes and monitors vertices continuously. Zookeeper coordinates Nimbus and Supervisors, and stores the status of Nimbus and Supervisors. All monitor data of Supervisors and Worker nodes is also stored in Zookeeper, which can be used in the later redirection and rescheduling stage.

5.2. Lightweight load balancing based redirection among instances

To improve the load balancing of data center *DC*, redirection of data stream $ds_{i(O)}$ among n instances $\{v_{j1}, v_{j2}, \ldots, v_{jn}\}$ of downstream v_j is an effective way at the vertex level. For the instances of v_j, the influencing factors on the load state are only restricted by all the n instances of v_j, and there is no need to consider the state information of other vertices, which greatly reduces the complexity of decision making process and achieves lightweight real-time adjustments.

For upstream v_i, each data tuple dt in $ds_{i(O)}$ is emitted to one of the n instances of v_j during $[t_s, t_e]$ by load based redirection function $f\left(lb_{v_i,v_j,[t_s,t_e]}\right)$. For the kth instance, the load-based redirection function $f\left(lb_{v_i,v_j^k,[t_s,t_e]}\right)$ is defined by (12).

$$f\left(lb_{v_i,v_j^k,[t_s,t_e]}\right) = \frac{length_{q\_f}^k}{\sum_{l=1}^{n} length_{q\_f}^l}, \tag{12}$$

where $length_{q\_f}^k$ is the available queue length of the kth instance. It suggests that for a downstream instance, the longer its idle queue, the higher the probability that it receives a data tuple.

The following two rules are employed to improve the load balancing of *DC* at the vertex level.

Rule 1: An instance with an empty queue is always prioritized in data tuple assignment.

If there is an instance with an empty queue, a data tuple is preferentially assigned to it to balance computing load among instances of a vertex.

Rule 2: An instance with a full queue no longer receives new data tuples until there is new space available.

If the queue of an instance is full, it is unable to store new data tuples, therefore there should be no more new data tuples received. This rule minimizes the probability of discarding data tuples.

The algorithm for lightweight load balancing based data stream redirection among instances is described in Algorithm 1.

Algorithm 1: Lightweight load balancing based redirection algorithm among instances.

1. **Input**: output data stream $ds_{i(O)}$ of upstream v_i, instances of downstream v_j.
2. **Output**: input data tuple assigned to each instance of v_j.
3. **if** the number of instances is 1 **then**
4. Input data tuple of the instance is the output data stream $ds_{i(O)}$.
5. **return** input data tuple assigned to each instance of v_j.
6. **end if**
7. **for** each computing node cn_i running instance **do**
8. Set configuration parameters minimum load ratio $lr_{\min,cn_i}$ and maximum load ratio $lr_{\max,cn_i}$.
9. **end for**
10. **while** new data tuple is produced by v_i **do**
11. **for** each instance v_j^k of v_j **do**
12. Get the queue state of each instance.
13. Calculate the load ratio for each computing node by (5).
14. **end for**
15. **for** each instance v_j^k of v_j **do**
16. **if** the queue of the kth instance v_j^k is empty **then**
17. Emit the new data tuple to v_j^k.
18. **end if**
19. **if** the queue of the kth instance v_j^k is full **then**
20. Skip v_j^k.
21. **end if**
22. **end for**
23. Sort all n instances by available queue length in descending order.
24. Calculate the load-based redirection function $f\left(lb_{v_i,v_j^k,[t_s,t_e]}\right)$ by (12).
25. Output the new data tuple to the instance with the load-based redirection function.
26. **end while**
27. **return** input data tuple assigned to each instance of v_j.

The input of this algorithm includes the output data stream $ds_{i(O)}$ of upstream v_i, instances of downstream v_j. The output is the input data tuple for each instance of v_j. Step 7 to step 9 set the minimum load ratio $lr_{\min,cn_i}$ and maximum load ratio $lr_{\max,cn_i}$ for each computing node. Step 11 to step 14 get the queue state and calculate the load ratio for each computing node. Step 15 to step 22 decide the output data tuple for instances with an empty or full queue. Step 23 to step 25 output the new data tuple by the load-based redirection function. The time complexity of Algorithm 1 is $O(n)$, where n is the number of instances of vertex v_j.

5.3. Logical ring-based state management

Besides of data stream redirection, vertex state management also affects computing performance. For example, if a stateful vertex v_s fails, its state will be lost, undermining the accuracy of data tuple processing [21,22]. To maintain the states of stateful vertices, checkpoint mechanism is employed in Storm for state management [8]. However, checkpoint data is stored in external storage, requiring longer fault tolerance time to restore the system state. It is also possible that part of state data is lost during a failure, which may further affect the accuracy of system recovery. We implement an online state backup and recovery mechanism to address this problem. It can conduct real-time fault recovery without losing state data, thus improving the accuracy of data recovery.

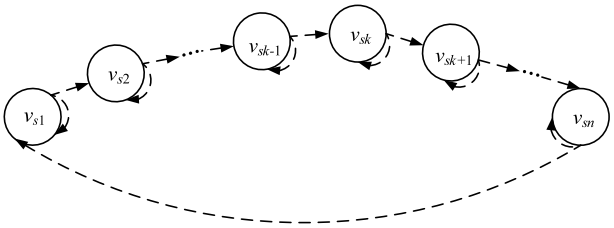

Fig. 15. State backup for a stateful vertex.

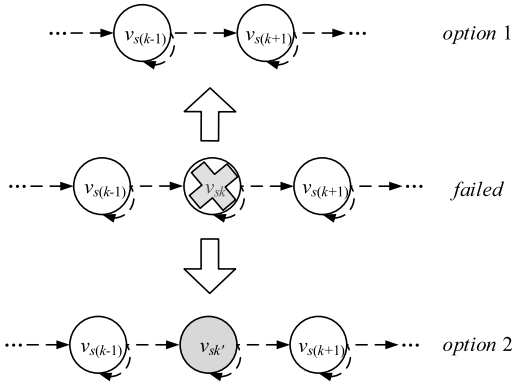

Fig. 16. State recovery for a failed stateful vertex.

For a stateful vertex v_s, if the number of instances is n, then $n \geq 2$. All the n instances form a logical ring as they share state backup and synchronization information, where each instance manages its own state and periodically synchronizes its state with adjacent instance nodes. The synchronization period T parameter can be set as needed. As shown in Fig. 15, the kth instance v_{sk} of v_s manages its own state, simultaneously synchronizing its own state with the $(k+1)$th instance v_{sk+1}, and the $(k-1)$th instance v_{sk-1}.

If an instance node fails, its state still exists and does not stop the system from further processing. As shown in Fig. 16, when the kth instance v_{sk} failed, at this point, there are two options to resolve its failure: the first option is to let the $(k+1)$th instance $v_{s(k+1)}$ take over the work of v_{sk}, and receive synchronization state information from the $(k-1)$th instance $v_{s(k-1)}$. If v_{s1} fails, synchronization state information is received from instance v_{sn}. Another option is to create a new instance $v_{sk'}$, recover the state of v_{sk} from $v_{s(k+1)}$, and take over the work of the failed instance v_{sk}. If v_{sn} fails, recovering the state from v_{s1}.

The logical ring-based state management algorithm at the instance level is described in Algorithm 2.

Algorithm 2: Logical ring-based state management algorithm at the instance level.

1. **Input**: instances of stateful vertex v_s, input data tuple of v_s, and state of computing node for each instance.
2. **Output**: state dependency of all instances of v_s.
3. **if** the number of instances is 1 **then**
4. Create one instance and make the number of instances of v_s meet the condition $n \geq 2$.
5. **end if**
6. Built a logical ring for the n instances of v_s according to their logical sequence number.
7. Set configuration parameter minimum synchronization period T.
8. **while** new data tuple is coming to k^{th} instance v_{sk} of v_s **do**
9. **if** the synchronization interval of the k^{th} instance v_{sk} is greater than minimum synchronization period T and the state of v_{sk} has also been updated **then**
10. Synchronize state of v_{sk} to the $(k+1)^{\text{th}}$ instance $v_{s(k+1)}$.
11. **end if**
12. **if** the kth instance v_{sk} failed **then**
13. **if** the number of instances of v_s is larger than 1 **then**
14. Select a new computing node, create a new instance $v_{sk'}$, recovery the state of v_{sk} from v_{sk+1}, and take over the work of the failed instance v_{sk}.
15. **else**
16. Select the $(k+1)^{\text{th}}$ instance $v_{s(k+1)}$ to take over the work of v_{sk}, and make $v_{s(k+1)}$ synchronize status from the $(k-1)^{\text{th}}$ instance $v_{s(k-1)}$.
17. **end if**
18. Update the state of logical ring for all instances of v_s.
19. **end if**
20. Update the computing nodes in data center DC.
21. **end while**
22. **return** state dependency of all instances of v_s.

The input of this algorithm includes instances of stateful vertex v_s, input data tuple of v_s, and state of computing node for each instance. The output is state dependency of all instances of v_s. Step 3 to step 5 check and create instance as needed to make the number of instances of v_s meet the condition $n \geq 2$. Step 9 to step 11 synchronize state of v_{sk} to the $(k+1)$th instance v_{sk+1} according to the condition that synchronization interval of the kth instance v_{sk} is greater than the minimum synchronization period T and the state of v_{sk} has been updated. Step 12 to step 19 recover failed instance by creating new instance or by selecting an existing instance to take over its work. The time complexity of Algorithm 2 is $O(1)$, as only one instance of v_s needs to update the state information in each cycle.

5.4. Deployment optimization

Deployment optimization is the key to accomplish computing node-level data stream redirection, as well as to optimize resource use, minimize the load balancing of data center DC, and improve throughput and response time of elastic stream computing systems [23].

In the initial deployment stage, we employ a modified first fit strategy with full-scale vertices deployment to minimize response time. In the online optimization stage, we employ the modified first-fit strategy again but with differential-scale vertex re-deployment, taking into account the current deployment of vertices. To maximize the throughput, we adjust the number of instances in a timely manner under the various resource constraints. The deployment optimization algorithm is described in Algorithm 3.

The input of this algorithm includes m stream applications set $G_{sa} = \{G_1, G_2, \ldots, G_m\}$, num available computing nodes in data center $DC = \{cn_1, cn_2, \ldots, cn_{num}\}$, input data stream $\{r_{ds}(G_1), r_{ds}(G_2), \ldots, r_{ds}(G_m)\}$ for each stream application. The output is the deployment of stream applications on available computing nodes. Step 6 and step 7 initialize configuration parameters. Step 8 to step 15 deploy each vertex in one stream application to an available computing node in a topologically ordered manner by the first fit strategy. Especially, all instances of any vertex are deployed to different compute nodes. Step 17 to step 22 improve load balancing $lb_{DC,[t_s,t_e]}$ of the data center during $[t_s, t_e]$ by redirecting the input data tuple at instance level with Algorithm 1. Step 23 to step 28 keep load ratio $lr_{cn_i,[t_s,t_e]}$ of computing node cn_i during $[t_s, t_e]$ within range $\left[lr_{\min,cn_i}, lr_{\max,cn_i}\right]$ by redirecting the input data tuple at instance level with Algorithm 1. Step 29 to step 33 optimize latency of application G_i by first fit strategy with differential-scale vertex re-deployment. Step 34 to step 38 optimize throughput of application G_i by increasing the number of instances by p_{ins}% for each vertex, and deploying new instances to an available compute node in a topologically ordered manner with the first fit strategy. Step 39 to step 43 build state dependency of all instances for each stateful vertex by Algorithm 2. The time complexity of Algorithm 3 is $O(m \cdot n)$, where m is the number of stream applications, and n is the number of instances of a vertex.

6. Performance evaluation

This section focuses on the evaluation of the proposed Dr-Stream framework, discussing the experimental environment and parameter settings, and providing a performance analysis on the results.

6.1. Experimental environment and parameter setup

The proposed Dr-Stream framework is implemented and run as an extension to Storm 1.2.2 [8] on top of CentOS 6.3 operation system. A monitor module is developed to monitor the performance of Supervisors and Worker nodes. The redirection algorithm among instances is implemented through the CustomStreamGrouping interface. The deployment optimization algorithm is implemented through the IScheduler interface and specified through configurations file Storm.yaml. Extensive experiments have been conducted in a cluster hosted in the school of Information Engineering, CUGB. The cluster consists of 35 computing nodes connected through a 1 Gbps LAN, with 1 designated computing node serving as the master node, running Storm Nimbus, 2 designated as the Zookeeper nodes, and the rest 32 machines as the slave nodes, running Supervisor nodes. Each machine in the cluster is equipped with Intel Core (TM) i5-8400 @ 2.8 GHz with 6-core and 8 GB RAM.

Algorithm 3: Deployment optimization algorithm.

Input: m stream application set $G_{sa}=\{G_1,G_2,\cdots,G_m\}$, num available computing nodes in data center $DC=\{cn_1,cn_2,\cdots,cn_{num}\}$, and input data stream $\{r_{ds}(G_1),r_{ds}(G_2),\cdots,r_{ds}(G_m)\}$ for each stream application.
Output: Deployment of stream applications on available computing nodes.
if stream application set G_{sa} or available computing nodes is null **then**
  return null.
end if
Initialize configuration parameters, such as the minimum load balancing $lb_{\min,DC}$ and the maximum load balancing $lb_{\max,DC}$ of DC, the maximum latency $l_{\max}(G)$ and minimum throughput $T_{\min}(G)$ for each application, the minimum load ratio $lr_{\min,cn_i}$ and maximum load ratio $lr_{\max,cn_i}$ for each computing node in DC.
Set the initial number of instances per vertex for each stream application, and the incremental proportion p_{ins}% of the number of instances for each vertex.
for each stream application in G_{sa} **do**
  Sort all computing nodes by available CPU in descending order.
  Deploy each vertex of the stream application to an available compute node in a topologically ordered manner by the first fit strategy.
  if the number of instances of a vertex is larger than 1 **then**
    Keep all the instances of any vertex deployed to different compute nodes.
  end if
  Update the available CPU of the deployed computing nodes.
end for
while $lb_{DC,[t_s,t_e]}$ is out of the range $[lb_{\min,DC},lb_{\max,DC}]$ or $lr_{cn_i,[t_s,t_e]}$ is out of $[lr_{\min,cn_i},lr_{\max,cn_i}]$ or the latency of any application G_i is greater than the maximum latency $l_{\max}(G)$, or the throughput of any application G_i is less than the minimum throughput $T_{\min}(G)$ **do**
  if $lb_{DC,[t_s,t_e]}$ is out of the range $[lb_{\min,DC},lb_{\max,DC}]$ **then**
    for each stream application in G_{sa} **do**
      Get the output data stream $ds_{i(O)}$ of upstream v_i and instances of downstream v_j.
      Redirect input data tuple for each instance of v_j by Algorithm 1.
    end for
  end if
  if $lr_{cn_i,[t_s,t_e]}$ is out of the range $[lr_{\min,cn_i},lr_{\max,cn_i}]$ **then**
    for each instance v_j on cn_i **do**
      Get the output data stream $ds_{i(O)}$ of upstream v_i and instances of downstream v_j.
      Redirect input data tuple for each instance of v_j by Algorithm 1.
    end for
  end if
  if any latency of application G_i is greater than the maximum latency $l_{\max}(G)$ **then**
    Sort all computing nodes by available CPU in descending order.
    Pre-deploy each vertex in the stream application to an available compute node in a topologically ordered manner by the first fit strategy.
    Calculate the difference between existing deployment and pre-deployment, and redeploy the vertices accordingly.
  end if
  if the throughput of any application G_i is less than the minimum throughput $T_{\min}(G)$ **then**
    Increase the number of instances by p_{ins}% for each vertex.
    Sort all computing nodes by available CPU in descending order.
    Deploy new instances in the stream application to an available compute node in a topologically order by the first-fit strategy.
  end if
  for each stream application G_i in G_{sa} **do**
    for each stateful vertex v_s in G_i **do**
      Build state dependency of all instances of v_s by Algorithm 2.
    end for
  end for
  Update state of stream application set G_{sa}, available computing nodes in DC.
  Monitor input data stream $\{r_{ds}(G_1),r_{ds}(G_2),\cdots,r_{ds}(G_m)\}$ for each stream application.
end while
return deployment of stream applications on available computing nodes.

Moreover, two types of stream applications Top_N and WordCount are submitted to the data center.

As shown in Fig. 17, the topology of TOP_N consists of v_{reader}, v_{count}, v_{rank} and v_{merge}. The function of each vertex is shown in Table 1. The data stream grouping strategy from v_{reader} to v_{count}, from v_{count} to v_{rank}, and from v_{rank} to v_{merge} are fields grouping, fields grouping, and globalGrouping grouping, respectively. The initial number of instances for v_{reader}, v_{count}, v_{rank} and v_{merge} are 20, 20, 16, 8.

As shown in Fig. 18, the topology of WordCount consists of v_{reader}, v_{split} and v_{count}. The function of each vertex is shown in Table 2. The data stream grouping strategy from v_{reader} to v_{split} and from v_{split} to v_{count} are shuffleGrouping and fields grouping, respectively. The initial number of instances for v_{reader}, v_{split} and v_{count} are 30, 30, 20, respectively.

The parameter settings of the experiment are shown in Table 3.

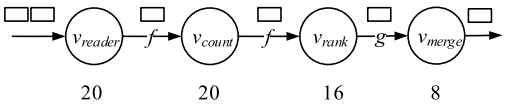

Fig. 17. Logical graph of Top_N.

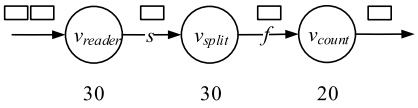

Fig. 18. Logical graph of WordCount.

Table 1
Function of vertex in the logic graph of Top_N.

Vertex	Function
v_{reader}	Read words from data stream
v_{count}	Count words
v_{rank}	Rank words by count
v_{merge}	Merge all ranks from upstream

Table 2
Function of vertex in the logic graph of WordCount.

Vertex	Function
v_{reader}	Read sentence from data stream
v_{split}	Split words of sentence
v_{count}	Count words

Table 3
Parameter settings.

Notations	Value	Description
$lb_{min,DC}$	0	Min load balancing of *DC*
$lb_{max,DC}$	0.3	Max load balancing of *DC*
lr_{min,cn_i}	0	Min load ratio of computing node cn_i
lr_{max,cn_i}	0.99	Max load ratio of computing node cn_i
p_{ins}	40	Incremental % of the number of instances

6.2. Performance results

The experimental settings contain four evaluation parameters: average latency *AL*, average throughput *AT*, average load balancing $lb_{avg,DC}$ of data center *DC*, and average load ratio $lr_{avg,cn}$ of a computing node.

(1) *Average Latency*. The average latency *AL* or average response time of applications is one of the most important performance indicators that reflect the overall responsiveness of an elastic stream computing system. On Storm platform, *AL* can be retrieved through the Storm UI. The shorter the average latency *AL*, the better the real-time performance.

The real-time latency of both Dr-Stream and the default deployment strategies can reach a relatively stable state. In the stable phase, Dr-Stream has a shorter real-time latency as compared to the default strategy on Storm. As shown in Fig. 19, when the rate of input data stream is 1000 tuples/s, the average latency of Dr-Stream and that of the default are gauged at 41 ms and 52 ms, respectively. Dr-Stream reduces it by 21%. When the rate of input is 2000 tuples/s, the average latency of Dr-Stream and that of the default Storm deployment strategy are gauged at 109 ms and 149 ms, respectively. Dr-Stream reduces it by 26%. For the two general stream applications, it is apparent that the average latency of Dr-Stream is shorter than that of the default Storm scheduling strategy in the stable phase. The average latency is reduced by more than 20%.

With the increase of input rate, the average latency increases under both deployment strategies. For a given input rate, Dr-Stream has a shorter average than the default Storm strategy.

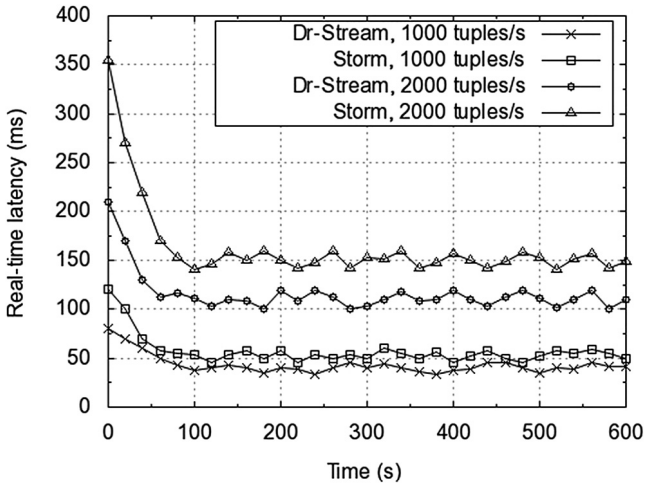

Fig. 19. Real-time latency at different input rates.

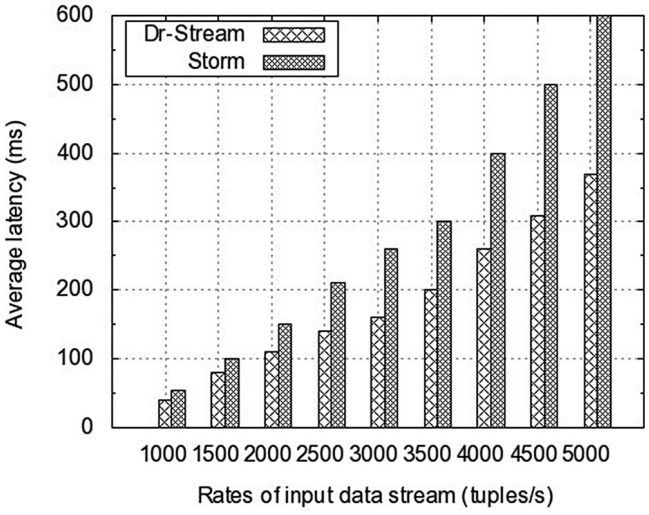

Fig. 20. Average latency at different input rates.

As shown in Fig. 20, when the rate of input data stream is 1500 tuples/s, the average latency of Dr-Stream and that of the default Storm deployment strategy change to 78 ms and 98 ms, respectively. Dr-Stream reduces it by 20%. However, when the rate increases to 5000 tuples/s, the average latency of Dr-Stream and that of the default Storm deployment strategy climb to 372 ms and 595 ms, respectively. Dr-Stream reduces it by 37%. The higher the data rate, the more significant this difference is noted.

(2) *Average throughput*. The average throughput *AT* is the average data rates delivered by an output vertex of a stream application. It can be evaluated by the number of output tuples per second produced by each stream application. Average throughput is also one of the most important performance indicators that reflect the overall processing capability. The greater the system throughput, the stronger the data processing capability of the stream computing system.

The real-time throughput of both deployment strategies can reach a relatively stable state. In the stable phase, Dr-Stream has a higher real-time throughput as compared to the default Storm strategy. As shown in Fig. 21, when the rate of input data stream is 1000 tuples/s, the average throughput of Dr-Stream and that of the default Storm deployment strategy are gauged at 530 tuples/s and 451 tuples/s, respectively. Dr-Stream improves it by

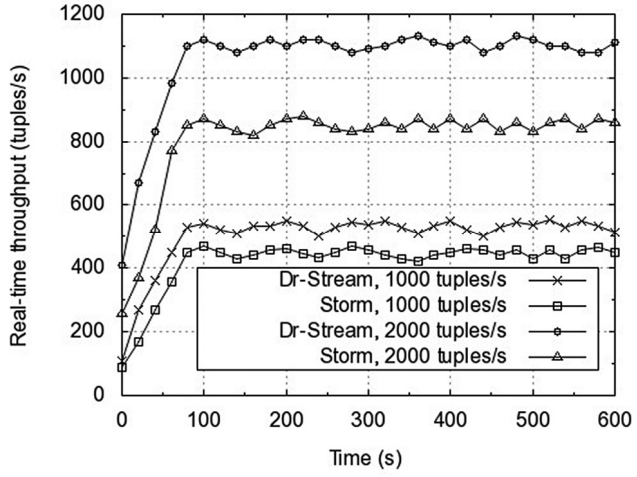

Fig. 21. Real-time throughput at different input rates.

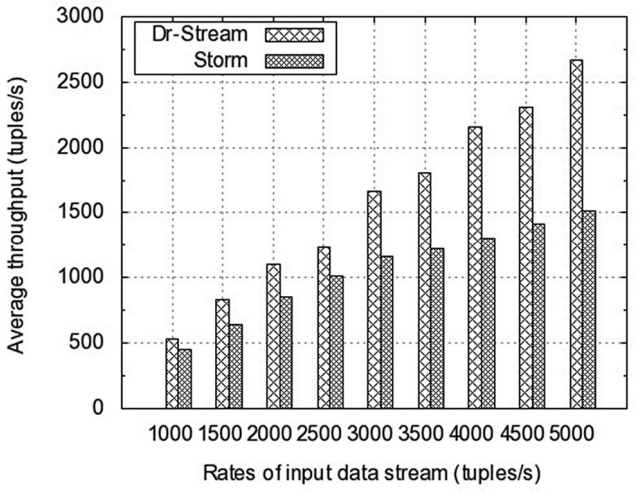

Fig. 22. Average throughput at different input rates.

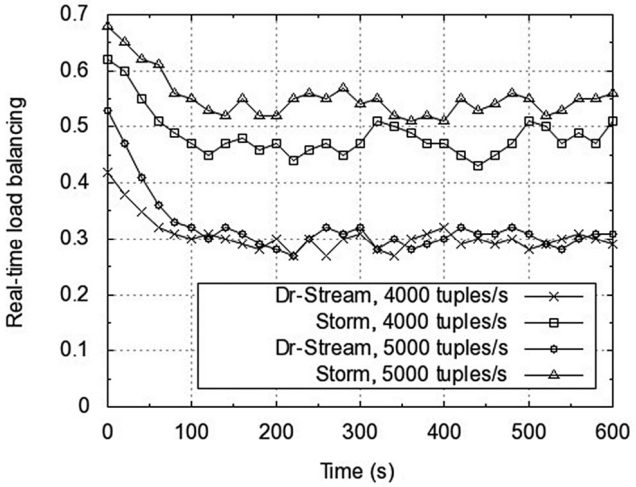

Fig. 23. Real-time load balancing at different input rates.

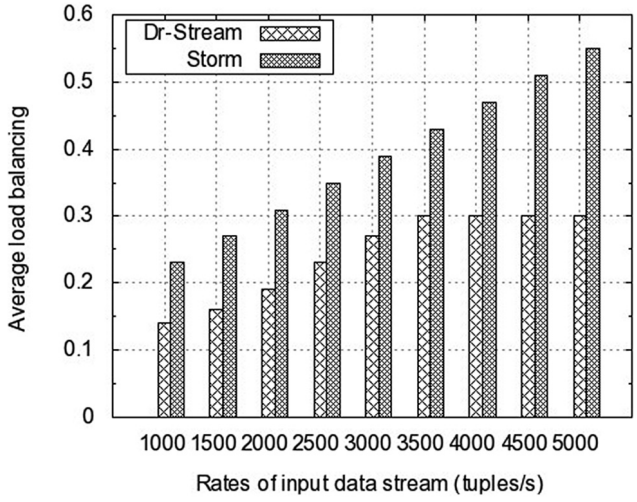

Fig. 24. Average load balancing at different input rates.

17%. When the rate is 2000 tuples/s, the average throughput of Dr-Stream and that of the default change to 1103 tuples/s and 851 tuples/s, respectively. Dr-Stream improves it by 29%. For the given general stream applications, it is apparent that the average throughput by Dr-Stream is higher than that of the default at the stable phase. The average throughput is improved by more than 15%.

At each input rate, Dr-Stream beats the default Storm strategy with a higher average throughput. With the increase of rate, the difference between the two strategies becomes apparent. The higher the data rate, the more significantly this difference is noted. As shown in Fig. 22, when the rate is 1500 tuples/s, the average throughput of Dr-Stream and that of the default Storm deployment strategy are 832 tuples/s and 641 tuples/s, respectively. Dr-Stream improves the average throughput by 29%. The difference between the two is small. However, when the rate increases to 5000 tuples/s, the average throughput of the two become 2673 tuples/s and 1521 tuples/s, respectively. Dr-Stream improves it by 75%. The difference becomes rather significant.

(3) *Average load balancing $lb_{avg,DC}$ of data center DC*. The average load balancing $lb_{avg,DC}$ of data center *DC* reflects the load balancing metric of CPU queues on computing nodes during a period of time. If the average load balancing $lb_{avg,DC}$ of *DC* is small, the better load status is observed, and it demonstrates how well the stream computing system adapts to data stream fluctuation.

The real-time load balancing of both deployment strategies can reach a relatively stable state. In the stable phase, Dr-Stream has a less real-time load balancing as compared to the default Storm strategy. As shown in Fig. 23, when the rate of input data stream is 4000 tuples/s, the average load balancing of Dr-Stream and that of the default Storm deployment strategy are 0.3 and 0.47, respectively. Dr-Stream reduces it by 36%. When the rate of input data stream change to 5000 tuples/s, the average load balancing values become 0.3 and 0.54, respectively. Dr-Stream reduces it by 44%. For the given general examples of stream applications, it is apparent that the average load balancing by Dr-Stream is less than that of the default Storm scheduling strategy in the stable phase. The average load balancing is reduced by more than 35%.

At each input rate, Dr-Stream beats the default Storm strategy with a less average load balancing value. When the rate increases to a certain extent, Dr-Stream can stabilize the average load balancing at the set maximum load balancing $lb_{\max,DC}$ of data center *DC* by dynamically redirecting of real-time data streams. As shown in Fig. 24, when the rate of input data stream is 1500 tuples/s, the average load balancing of Dr-Stream and that of

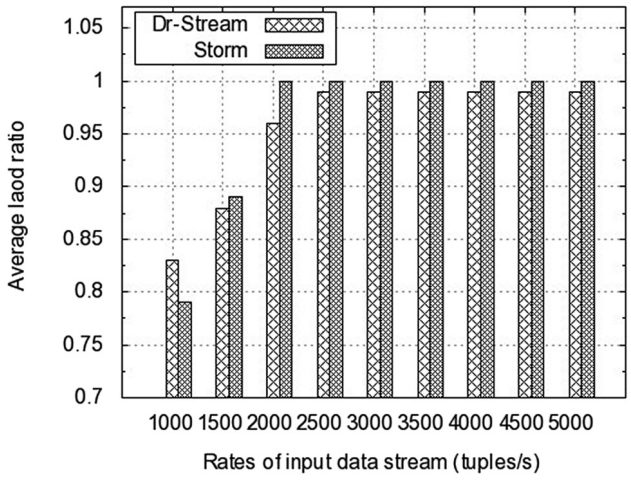

Fig. 25. Average load ratio at different input rates.

the default Storm deployment strategy are 0.16 and 0.27, respectively. Dr-Stream reduces it by 40%. When the rate is greater than 3500 tuples/s, the average load balancing of Dr-Stream is stabilized at 0.3. However, the average load balancing of the default continues to increase along with the increasing data stream rate. When the rate is greater than 5000 tuples/s, the average load balancing of the default reaches 0.54.

(4) *Average load ratio* $lr_{avg,cn}$. The average load ratio $lr_{avg,cn}$ of a computing node *cn* reflects the loading state of its CPU queue. If $lr_{cn,[t_s,t_e]} \geq 1$, then computing node *cn* is overloaded, lowering the system performance. It is better to have the average load ratio $lr_{avg,cn}$ kept within the range of (0, 1). Given the average load ratio $lr_{avg,cn} \in (0, 1)$, the larger the average load ratio $lr_{avg,cn}$, the higher the effective utilization of the compute node.

With the increase of rate, the average load ratio of computing nodes increases under both deployment strategies. When the rate increases to a certain extent, Dr-Stream can stabilize the average load ratio at the set maximum load ratio $lr_{\max,cn_i}$ of computing node. As shown in Fig. 25, when the rate is 1000 tuples/s, the average load ratio of the two are 0.83 and 0.79, respectively. This is because our scheduling strategy reduces the number of computing nodes used without affecting the performance of the stream computing system when the data steam rate is low. When the rate increases to 2500 tuples/s, the average load ratio climbs to 0.99 and 1, respectively, avoiding overloading computing node and ensuring the system performance. When the rate is greater than 2500 tuples/s, the load ratio by Dr-Stream can be controlled at a high but efficient level, however, the ratio produced by the default is always in an overload state, which directly lowers the system performance.

7. Related work

In this section, we review the three broad categories of related work: big data stream computing, load balancing optimization for elastic stream computing systems and stream application deployment on Storm platform.

7.1. Big data stream computing

Stream computing and batch computing [24] are the two most important forms of big data computing paradigms, and they can meet the computing needs of most big data scenarios. In general, big data batch computing is used for large-scale data processing in batches, where the processing results must be highly accurate, and higher data processing delay is tolerable, even if it could take as long as hours or days. On the other hand, an increasing number of application scenarios have stringent real-time requirements. The timely processing of data has become more prominent, and the accuracy of processing results is no longer the primary goal.

To address this need, a new generation of big data streaming computing architecture has been proposed and widely used, such as Storm [8], Heron [25], and Samza [26]. Stream computing and batch computing are not opposing alternatives but interactants. They can be combined with each other to meet various goals of data processing at different stages. At the initial stage when fresh data is generated, new values need to be mined promptly. Stream computing can meet the data processing needs at this stage because of its stringent timeliness. When analyzing the intrinsic values and the regularity of data at a later stage, the batch calculation can meet the needs of data processing quite well because of its high accuracy.

Stream computing also provides new opportunities for many computing scenarios. Generally, security issues [27,28] are one of the most significant challenges in cloud and fog computing environments. Timeliness is crucial for the inspection of security threats [29]. Through a streaming computing platform, we can analyze data streams in real time to identify propagation sources [30] and detect threats [31]. In addition, stream computing can also be applied in the fields of precision advertising [32] and smart transportation [32], etc.

7.2. Load balancing optimization for elastic stream computing systems

Load balancing optimization plays an important role [33,34] to meet the need of low system latency and high system throughput for an elastic stream computing system as the rate of real-time data stream fluctuates over time. In recent years, there is great interest in improving load balancing in a distributed environment. However, it is challenging [35,36] to achieve load balancing in stream computing systems due to a series of factors, such as the unbounded volume of data streams, the fluctuating arrival rate, and the lack of global consistency.

In [12], the authors proposed a framework that integrated the optimization of load balancing, operator instance collocations and horizontal scaling. Load balancing and horizontal scaling were modeled as Mixed-Integer Linear Program, LP solver was used to improve the load distribution in computing cluster.

To improve the load balancing for stateful applications in Storm system, a locality-aware routing strategy was proposed in [37], to improve the data stream locality for stateful stream processing applications. Data migration was considered in the routing processing.

Focusing on the need to incorporate aggregation cost in the partitioning model, a cost model for stream partitioning was introduced in [38], where the imbalance and aggregation cost on the window of a stateful vertex was considered, and a stream partitioning strategy was proposed to minimize the latency and throughput of stream computing systems.

To improve throughputs and communication cost, in [39], a scalable scheme partitioning model was proposed for stream join operators. To find the migration plan with minimal data communication cost, a lightweight computation model was also presented.

Focusing on the dynamic parallelism configuration of partitioning tasks on Spark platform, an analytical model for expressing the running time was proposed in [40], where an algorithms for configuring dynamic partitioning was given to optimize the resource of data center and the running time of Spark tasks.

To improve the throughput of a stream computing system, in [41], a partitioning function for stateful data parallelism was

proposed. Various desirable properties for the function was given, such as balance properties, structural properties, adaptation properties.

To summarize, load balancing optimization for elastic stream computing systems has been studied in many research work. However, most of them achieved load balancing by redirecting data stream among multiple instances at the instance level only. Extensive experiments in this paper showed that such implementation of load balancing is limited. It is necessary to achieve a larger range and a higher level of load balancing at vertex level through node rescheduling. In our work, we improved the system load balancing by simultaneously redirecting real-time data stream at the instance level and by optimizing vertex deployment at the vertex level.

7.3. Stream application deployment on storm platform

Stream applications deployment is important on Storm platform to satisfy its performance requirements, such as low system latency, high system throughput, and acceptable load balancing. It is hard to find an optimal deployment as the problem of Stream applications deployment is NP-hard [42,43], which has also been extensively studied in recent years.

To implement an elastic stream computing system, in [13], the authors tried to scale a stream computing system from two dimensions: operator parallelism and resource scaling. A fine-grained model and an elastic scaling framework that estimated the resources utilization was proposed and implemented on Storm platform.

In response to the uncertainty and complexity of streaming data, in [44], a model-based scheduling scheme for stream processing systems was proposed. It captured the system behavior and provided an optimal allocation strategy to adapt to the changing work conditions on Storm platform.

Due to the unpredictability of data sources, and the fact that these stream applications often operate in a dynamic environment, the streaming applications require the support of elastically scaling in response to workload variations. In [45], an optimal operator deployment and replication strategy was proposed to achieve elastic distributed data stream processing. The deployment and runtime decisions were made by solving a suitable integer linear programming problem, with an objective function capturing the relative importance between QoS goals and reconfiguration costs.

Graph partitioning is a NP-hard complexity problem in computer science. In [46], graph partitioning was employed to partition the work-load of stream programs, to improve the deployment of applications, and to optimize the throughput on heterogeneous distributed stream computing platforms.

A single bottleneck of an application (congested link or an overloaded operator) can drastically throttle the system throughput. In [47], two techniques were proposed to address the bottleneck problems on stream computing platforms, which were network-aware routing for fine-grained control of streams and dynamic overlay generation for optimizing performance of group communication operations.

Stream application deployment has been studied extensively from resource optimization perspective recently with various goals. However, the problem of continuous fluctuation in real-time data streams has not been fully considered from the data stream redirection perspective, which is another important factor that affects the performance of streaming computing systems. In this paper, we studied the application deployment strategy on Strom platform from both the resource optimization perspective and the data stream redirection perspective. It provided significant performance improvements on metrics such as system latency, throughput, and load balancing.

8. Conclusions and future work

Low system latency, high system throughput, and acceptable load balancing are critical performance requirements for an elastic stream computing system. One of the major challenges to realize an elastic stream computing system is how to continuously and adaptively adjust resource allocation for streaming applications. A rescheduling scheme with efficient performance optimization does not always work due to the relatively long decision time. The computing environment might have undergone fundamental changes due to data steam fluctuation. In addition, the loss of vertex state information and load shedding of data tuples during operator rescheduling have not been fully considered. To address these problems, we try to optimize system performance from the perspective of data stream redirection by proposing a dynamic redirection framework Dr-Stream. It processes fluctuating data streams in a scalable and elastic manner, optimizing system performance and adjusting the load to resources without causing data or state loss. Meanwhile, the fine-grained resource optimization is also incorporated to further improve system performance, such as lower system latency, higher system throughput, and acceptable load balancing.

Our contributions are summarized as follows:

(1) Provided a general stream application model, a data stream model and a data stream grouping model, as well as formalization of the load balancing optimization and data stream redirection problems;

(2) Redirected data streams among multiple instances of an operator at runtime; managing the states of stateful operators by a logical ring-based strategy;

(3) Determined the number of instances for each operator, and deployed the instances to computing nodes;

(4) Evaluated the fulfillment of low latency, high throughput, and acceptable load balancing objectives;

(5) Implemented a prototype and tested the performance of the proposed Dr-Stream;

Our future work will be focusing on the following directions:

(1) To integrate the state migration as a part of Dr-Stream, considering fault tolerance strategies to improve system reliability.

(2) To apply the Dr-Stream in real big data stream computing application scenarios, such as urban intelligent transportation and geological disaster real-time warning.

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, Writing - review & editing. **Xindong You:** Data curation, Funding acquisition. **Rajkumar Buyya:** 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. 2652018081; Australian Research Council (ARC) Discovery Project; Promoting the Developing University Intension–Disciplin ary Cluster, China No. 5211910940, Qin Xin Talents Cultivation Program, China, and Beijing Information Science & Technology University, China No. QXTCP B201908.

References

[1] 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.

[2] A. Shukla, Y. Simmhan, Model-driven scheduling for distributed stream processing systems, J. Parallel Distrib. Comput. 117 (2018) 98–114.

[3] Z. Zvara, P.G.N. Szabó, B. Balázs, A. Benczúr, Optimizing distributed data stream processing by tracing, Future Gener. Comput. Syst. 90 (2019) 578–591.

[4] A. Rezaeian, M. Naghibzadeh, D.H.J. Epema, Fair multiple-workflow scheduling with different quality-of-service goals, J. Supercomput. 75 (2) (2019) 746–769.

[5] 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.

[6] 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.

[7] 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.

[8] Storm, http://storm.apache.org.

[9] 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.

[10] Q.C. To, J. Soto, V. Mark, A survey of state management in big data processing systems, VLDB J. 27 (6) (2018) 847–872.

[11] 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.

[12] K.G.S. Madsen, Y.L. Zhou, J.N. Cao, Integrative dynamic reconfiguration in a parallel stream processing engine, in: Proc. 2017 IEEE 33rd International Conference on Data Engineering, ICDE 2017, IEEE Press, 2017, pp. 227–230.

[13] 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.

[14] J. Fang, P. Chao, R. Zhang, X. Zhou, Integrating workload balancing and fault tolerance in distributed stream processing systems, World Wide Web (2019) http://dx.doi.org/10.1007/s11280-018-0656-0.

[15] S. Schneider, J. Wolf, K. Hildrum, R. Khandekar, K.L. Wu, Dynamic load balancing for ordered data-parallel regions in distributed streaming systems, in: Proc. 17th International Middleware Conference, Middleware 2016, Dec. 2016, p. a21.

[16] K. Vasiliki, L. John, H. Moritz, D. Desislava, F. Matthew, Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows, in: Proc. 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Oct. 2018, pp. 783-798.

[17] M. Bilal, M. Canini, Towards automatic parameter tuning of stream processing systems, in: Proc. 2017 Symposium on Cloud Computing, SoCC 2017, ACM Press, 2017, pp. 189–200.

[18] M.A.U. Nasir, G.D.F. Morales, N. Kourtellis, M. Serafini, When two choices are not enough: Balancing at scale in distributed stream processing, in: Proc. 2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016, IEEE Press, 2016, pp. 589–600.

[19] 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.

[20] D. Millot, C. Parrot, Optimization of the processing of data streams on roughly characterized distributed resources, IEEE Trans. Parallel Distrib. Syst. 27 (5) (2016) 1415–1429.

[21] C. Mariluz, B.V. Pablo, S.F. Luis, T-hoarder: A framework to process twitter data streams, J. Netw. Comput. Appl. 83 (2017) 28–39.

[22] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, M. Mock, Issues in complex event processing: Status and prospects in the big data era, J. Syst. Softw. 127 (2017) 217–236.

[23] 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.

[24] Y.P. Wen, Z.B. Wang, Y. Zhang, J.X. Liu, B.Q. Cao, J.J. Chen, Energy and cost aware scheduling with batch processing for instance-intensive IoT workflows in clouds, Future Gener. Comput. Syst. 101 (2019) 39–50.

[25] 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.

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

[27] S. Ivan, S. Wen, X. Huang, H. Luan, An overview of fog computing and its security issues, Concurr. Comput.: Pract. Exper. 28 (10) (2015) 2991–3005.

[28] S. Moin, A. Karim, Z. Safdar, K. Safdar, E. Ahmed, M. Imran, Securing IoTs in distributed blockchain: Analysis, requirements and open issues, Future Gener. Comput. Syst. 100 (2019) 325–343.

[29] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, K. Ren, Android HIV: A study of repackaging malware for evading machine-learning detection, IEEE Trans. Inf. Forensics Secur. 15 (1) (2020) 987–1001.

[30] J. Jiang, S. Wen, S. Yu, Y. Xiang, W. Zhou, Identifying propagation sources in networks: State-of-the-Art and comparative studies, IEEE Commun. Surv. Tutor. 19 (1) (2017) 465–481.

[31] T. Wu, S. Wen, Y. Xiang, W. Zhou, Twitter spam detection: Survey of new approaches and comparative study, Comput. Secur. 76 (2018) 265–284.

[32] T.R. Rao, P. Mitra, R. Bhatt, A. Goswami, The big data system, components, tools, and technologies: a survey, Knowl. Inf. Syst. 60 (3) (2019) 1165–1245.

[33] B.V. Pablo, F.G. Norberto, S.F. Luis, A.F. Jesus, Patterns for distributed real-time stream processing, IEEE Trans. Parallel Distrib. Syst. 28 (11) (2017) 3243–3257.

[34] N. Hidalgo, D. Wladdimiro, E. Rosas, Self-adaptive processing graph with operator fission for elastic stream processing, J. Syst. Softw. 127 (2017) 205–216.

[35] M.A.U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, M. Serafini, The power of both choices: Practical load balancing for distributed stream processing engines, in: IEEE 31st International Conference on Data Engineering, ICDE 2015, IEEE Press, 2015, pp. 137–148.

[36] G. Jon, S. Malte, B. Jonathan, T.A. Lara, E. Martin, K. Eddie, K.M. Frans, M. Robert, Noria: dynamic, partially-stateful data-flow for high-performance web applications, in: Proc. 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Oct. 2018, pp. 213–231.

[37] M. Caneill, A. El Rheddane, V. Leroy, N. De Palma, Locality-aware routing in stateful streaming applications, in: Proc. 17th International Middleware Conference, Middleware 2016, IEEE Press, 2016, p. a4.

[38] N.R. Katsipoulakis, A. Labrinidis, P.K. Chrysanthis, A holistic view of stream partitioning costs, in: Proc. 43rd International Conference on Very Large Data Bases, VLDB 2017, Aug. 2017, pp. 1286–1297.

[39] J. Fang, R. Zhang, X. Wang, A. Zhou, Distributed stream join under workload variance, World Wide Web 20 (5) (2017) 1089–1110.

[40] A. Gounaris, G. Kougka, R. Tous, C.T. Montes, J. Torres, Dynamic configuration of partitioning in spark applications, IEEE Trans. Parallel Distrib. Syst. 28 (7) (2017) 1891–1904.

[41] B. Gedik, Partitioning functions for stateful data parallelism in stream processing, VLDB J. 23 (4) (2014) 517–539.

[42] M. Nardelli, V. Cardellini, V. Grassi, F. Lo Presti, Efficient operator placement for distributed data stream processing applications, IEEE Trans. Parallel Distrib. Syst. (2019) http://dx.doi.org/10.1109/TPDS.2019.2896115.

[43] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, S. Pallickara, Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams, IEEE Trans. Parallel Distrib. Syst. 28 (12) (2017) 3553–3569.

[44] Y. Wang, Z. Tari, M.R.H. Farahabady, A.Y. Zomaya, Model-based scheduling for stream processing systems, in: Proc. 2017 IEEE 19th Intl Conference on High Performance Computing and Communications, HPCC 2017, IEEE Press, 2017, pp. 215–222.

[45] V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, Optimal operator deployment and replication for elastic distributed data stream processing, Concurr. Comput.: Pract. Exper. 30 (9) (2018) e4334(1-20).

[46] V.T.N. Nguyen, R. Kirner, Throughput-driven partitioning of stream programs on heterogeneous distributed systems, IEEE Trans. Parallel Distrib. Syst. 27 (3) (2016) 913–926.

[47] N. Rapolu, S. Chakradhar, A. Grama, VAYU: Accelerating stream processing applications through dynamic network-aware topology re-optimization, J. Parallel Distrib. Comput. 111 (2018) 13–23.

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. 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, 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.

Xindong You is currently an Associate Professor of Department of Computer Science at Beijing Information Science & Technology University, China. She was a post-doctoral position with the Beijing Institute of Graphic Communication, Tsinghua University from 2016 to 2018. Before as a post-doctoral, she is an Associate Professor at Hangzhou Dianzi University. She received her Ph.D. degree in computer science from Northeastern University, China in 2007. Her current research areas include Distributed Computing and Cloud Storage, etc.

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 132, 93,200+ 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.