Dynamic Redirection of Real-Time Data Streams for Elastic Stream Computing

Dawei Sun\textsuperscript{a, e}, Shang Gao\textsuperscript{b}, Xunyun Liu\textsuperscript{c}, Xindong You\textsuperscript{d}, and Rajkumar Buymä\textsuperscript{c}

\textsuperscript{a} School of Information Engineering, China University of Geosciences, Beijing, 100083, P.R. China
\textsuperscript{b} School of Information Technology, Deakin University, Waurn Ponds, Victoria 3216, Australia
\textsuperscript{c} Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia
\textsuperscript{d} Beijing Key Laboratory of Internet Culture and Digital Dissemination Research, Beijing Information Science & Technology University, Beijing, China, 100101
\textsuperscript{e} Polytechnic Center for Territory Spatial Big-data, MNR of China

E-mail: sundaweicn@cugb.edu.cn, shang.gao@deakin.edu.au, xunyunliu@gmail.com, youxindong@bistu.edu.cn, rbuyya@unimelb.edu.au

Corresponding Author: Dawei Sun

HIGHLIGHTS

\begin{itemize}
  \item A stream application model, data stream model and data stream grouping model in stream computing environments
  \item Formalization of load balancing optimization and data stream redirection for elastic stream computing.
  \item Reduction of data streams among multiple instances to improve load balancing of data center.
  \item Logical ring based state management for stateful operators to improve accuracy of data stream processing.
  \item Development and integration of Dr-Stream into Apache Storm.
\end{itemize}

ARTICLE INFO

Article history:
Received 3 Sept. 2019
Received in revised form 10 Feb. 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 performance 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, data stream model and data stream grouping model, as well as formulating the problem of load balancing optimization and data stream redirection. (2) Redirecting data streams among multiple instances of an operator at runtime time by a lightweight load balancing strategy to improve the load balancing of a data center at the vertex level. Managing system states, especially the state 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 during the runtime of a stream application. (4) Evaluating the fulfillment of low latency, high throughput, and load balancing objectives in a real-world distributed stream computing environment. Experimental results show that Dr-Stream reduces the average system latency and load balancing of data center by more than 20% and 15%, respectively. It also improves the average system stability by more than 15% and avoids over-utilization of computing nodes, as compared to the existing strategies in Storm.

I. 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 Figure 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 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 stateful operators.

![Real-time data stream](image)

Figure 1. Real-time data stream with different input rates of data stream.

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 optimizing 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 optimizing from the perspective of data stream redirection at the vertex level, which suits elastic stream computing systems and provides a scalable way to deal with fluctuating data streams.

### A. 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, data stream model and 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 time with a lightweight load balancing strategy that improves load balancing performance at the vertex level; managing the system state, especially the state 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 to 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 systems, called Dr-Stream, and test its performance. Experimental results conclusively demonstrate that the proposed Dr-Stream provides significant performance improvements on system latency, throughput and load balancing metrics.

### B. Paper Organization

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

II. 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 receives the same number of data tuples. As shown in Figure 2, all those output data tuples of vertex \( v_i \) are stored in an output queue, and will be partitioned to vertices \( v_{j1}, v_{j2}, v_{j3} \) consecutively. Shuffle grouping is one of the most common grouping strategies for real-time data streams partition, as 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 Figure 3, all those output data tuples of vertex \( v_i \) are stored in an output queue, and will be partitioned to vertices \( v_{j1}, v_{j2}, v_{j3} \) according to its fields’ value. \( v_{j1}, v_{j2}, v_{j3} \) will process data tuples with different fields’ values, respectively. Fields grouping is also one of the most common grouping strategies, as 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 special 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 Figure 4, all those output data tuples of vertex \( v_i \) are stored in an output queue, and will be partitioned to vertices \( v_{j1}, v_{j2}, v_{j3} \) 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 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 Figure 5, all those output data tuples of vertex \( v_i \) are stored in an output queue, and will be replicated to vertices \( v_{j1}, v_{j2}, v_{j3} \).
\(v_{ji}, v_{j2}\) 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 Figure 6, all the output data tuples of \(v_i\) and \(v_k\) are stored in their output queue, respectively, and will be only sent to vertices \(v_{ji}\) for further processing. No output data tuples will be partitioned to vertices \(v_{j2}\), 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 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 regard to 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. In this paper, we focus on dynamically redirecting real-time data streams on Storm 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.

III. 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 stream application model, data stream model, and data stream grouping model.

A. 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 \(V(G) = \{v_i | i \in \mathbb{I}\}\) is a finite set of \(n\) vertices, vertex \(v_i \in V(G)\) is an operator with a specific function, \(E(G) = \{e_{v_i,v_j} | v_i, v_j \in V(G)\}\) 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 vertex 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.

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 of downstream vertex set \(D(v_i)\). As shown in Figure 7, \(D(v_i) = \{v_j,v_m,v_n\}\) is the downstream vertex set of \(v_i\), and \(n_{D(v_i)} = 3\).
Vertex \( v_j \) receives its input data stream from its upstream vertices set \( U(v_j) \). \( n_{v_j} = |U(v_j)| \) is the number of upstream vertices of \( v_j \), and \( n_{v_j} \geq 0 \). If \( n_{v_j} = 0 \), then \( v_j \) is source vertex of stream application \( G \), which means it has no upstream vertex in \( G \). Each vertex of \( U(v_j) \) outputs a data stream to \( v_j \). All data streams aggregate as one single input data stream to \( v_j \). As shown in Figure 8, \( U(v_j) = \{v_j, v_w, v_r\} \) is the upstream vertex set of \( v_j \), and \( n_{v_j} = 3 \).

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

### B. Data Stream Model

A data stream \( ds \) is composed of a sequence of data tuples in the form of continuous stream, denoted as \( ds = \{dt_1, dt_2, \cdots \} \). The \( i \)th 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 \( i \)th data tuple \( dt_i \), respectively. Each vertex has at least one input data stream and/or one output data stream. As shown in Figure 9, \( ds_{\delta(j)} \) and \( ds_{\delta(i)} \) are the input data stream and output data stream of vertex \( v_j \), 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 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 heterogeneity [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 data streams continuously fluctuate over time. As multi-source data streams aggregate at vertex level, the input rate \( ir_{v_j} \) of vertex \( v_j \) is the sum of all input data stream rates of vertex \( v_j \). It can be described as (1).

\[
ir_{v_j} = \sum_{i=1}^{n} r_{v_{iA(i)}},
\]

where \( r_{v_{iA(i)}} \) is the \( i \)th input data stream rates, \( n \) is the number of input data stream of vertex \( v_j \).

### C. 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 the data tuple is replicated, a grouping strategy can be roughly categorized as replication grouping strategy or a non-replication grouping strategy.

For replication grouping strategy, each data tuple in the output data stream \( ds_{\delta(i)} \) of vertex \( v_j \) is replicated at the replication point, which can be set to any time point after the data tuple leaves \( v_j \) and before its transmission downwards.

After replication at the replication point, data tuples forms \( n \) output data streams \( ds_{\delta(i)}(1), ds_{\delta(i)}(2), \cdots, ds_{\delta(i)}(n) \), where \( n \) is determined by the downstream vertices of \( v_j \), and \( n = n_{\delta(v_j)} \). The data tuples are replica of the output data stream \( ds_{\delta(i)} \); that is \( \forall dt \in ds_{\delta(i)}(k), k \in \{1,2,\cdots,n\} \), then \( \exists dt \in ds_{\delta(i)}(j), j \in \{1,2,\cdots,n\} \) and \( \exists ds_{\delta(i)}(k) = ds_{\delta(i)}(j) \).

As shown in Figure 10, each data tuple of vertex \( v_j \) is replicated 3 times by partition function \( f(ds_{\delta(i)}) = r(dt(n=3)) \),
and forms 3 output data streams for each downstream vertex of $v_i$. On Storm platform, all grouping strategy is a replication group strategy.

![Figure 10. Replication based grouping model.](image)

For a non-replication grouping strategy, each data tuple of in the output data stream $ds_{i(0)}$ of vertex $v_i$ is emitted to one of downstream vertices of $v_i$ by partition function $f(ds_{i(0)})$, which is implemented according to specific grouping semantics. The output data stream $ds_{i(0)}$ of vertex $v_i$ is partitioned into $n$ input data streams given $n$ downstream vertices of $v_i$. That is if $\forall dt_i \in ds_{i(0)}(k), k \in \{1,2,\ldots\}$, then $\exists dt_i \in ds_{i(0)}$, and $ds_{i(0)} = ds_{i(0)}(1) \cup \ldots \cup \ldots$, 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 $\forall dt_i \in ds_{i(0)}(k)$, then $\exists dt_i \in ds_{i(0)}(j), k \neq j$, and $ds_{i(0)}(k) \cap \ldots = \emptyset, k \neq j$. It can be described as (3).

$$ds_{i(0)} = \bigcup_{k=1}^{n} \{1,2,\ldots\}$$  \hspace{1cm} (2)

$$\bigcup_{k=1}^{\gamma} \gamma, k \in \{1,2,\ldots\}$$  \hspace{1cm} (3)

As shown in Figure 11, the output data stream $ds_{i(0)}$ of vertex $v_i$ is partitioned into three sub-streams by the partition function $f(ds_{i(0)})$. One data tuple of output data stream $ds_{i(0)}$ 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.

![Figure 11. Non-replication based grouping model.](image)

### IV. PROBLEM FORMALIZATION

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

#### A. Load Balancing Problem of a Data Center

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

$$l_{cn[t_i,t_e]} = \sum_{v \in V_{cn[t_i,t_e]}} n_{v,cn[t_i,t_e]}.$$  \hspace{1cm} (4)

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

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

$$lr_{cn[t_i,t_e]} = \frac{l_{cn[t_i,t_e]}}{\text{length}_{cn}} = \frac{\sum_{v \in V_{cn[t_i,t_e]}} n_{v,cn[t_i,t_e]}}{\text{length}_{cn}},$$  \hspace{1cm} (5)

where $\text{length}_{cn}$ is the length of CPU queue of computing node $cn$.

The value of load ratio $lr_{cn[t_i,t_e]}$ of computing node $cn$ during $[t_i, t_e]$ should be greater than 0. If $lr_{cn[t_i,t_e]} > 1$, then computing node $cn$ during $[t_i, t_e]$ is overloaded, which means the CPU queue is full, and the new incoming data tuples will be discarded. If $lr_{cn[t_i,t_e]} \in [0,1]$, the load of $cn$ during $[t_i, t_e]$ is acceptable, and the larger the load ratio $lr_{cn[t_i,t_e]}$, the more fully the node $cn$ is used.
Let $DC = \{cn_1, cn_2, \cdots \}$ be a data center with $num$ computing nodes. Its load balancing $lb_{DC[k_i]}$ metric during $[t_i, t_i]$ can be evaluated by (6).

$$lb_{DC[k_i]} = \frac{1}{num} \sum_{k=1}^{num} |r_{cn_k[i], t_i} - \overline{l}_{[i], t_i}|,$$

(6)

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

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

**B. 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 throughput and minimize response time of system, optimizing resource use and load balancing metric $lb_{DC[k_i]}$ of $DC$ during $[t, t]$ without overloading any single computing node, satisfying user’s specified SLA constraints on system latency and throughput.

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

$$\min \left( \max_{G_i \in G_m} I(G_i) \right),$$

(7)

subject to

$$0 \leq I(G_i) \leq I_{\text{max}}(G_i), \forall G_i \in G_m,$$

and,

$$\min lb_{DC[k_i]},$$

(9)

subject to

$$0 < lb_{\text{min}, DC} \leq lb_{DC[k_i]} \leq lb_{\text{max}, DC} < 1, \forall cn_k \in DC,$$

(10)

in which $I(G_i)$ is the latency of the $i$th stream application in the set of $G_m$. $I_{\text{max}}(G_i)$ is the maximum latency of the $i$th application. $lb_{\text{min}, DC}$ and $lb_{\text{max}, DC}$ are the minimum and maximum load ratio of the $i$th computing node $cn_k$, respectively. Both are user-specified SLAs constraints.

**C. Data Stream Redirection**

Multiple data tuples $\{dt_1, dt_2, \cdots \}$ of data stream $ds_{(o)}$ can be redirected from vertex $v_j$ to $n$ instances $\{v_j, v_j, \cdots \}$ 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 $fr: ds_{(o)} = \{dt_1, dt_2, \cdots \} \rightarrow \{dt_1, dt_2, \cdots \}$. Usually, redirection function $fr(ds_{(o)}, lb_{DC[k_i]})$ can be described as (11).

$$fr(ds_{(o)}, lb_{DC[k_i]}) = \begin{cases} f(ds_{(o)}), & \text{if } lb_{DC[k_i]} \in [lb_{\text{min}, DC}, lb_{\text{max}, DC}], \\ f(lb_{DC[k_i]}), & \text{otherwise}. \end{cases}$$

(11)

where $f(ds_{(o)})$ is partition function and employed for data stream grouping, and $f(lb_{DC[k_i]})$ is a load based redirection function among multiple instances of a vertex for load balancing improvement.

As shown in Figure 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.
V. 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.

A. System Architecture

The system architecture of Dr-Stream includes four stages: topological construction, instantiation, scheduling, and redirection and rescheduling, as shown in Figure 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 vertex 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 analyzing its computational complexity, and 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 for dynamic redirection of real-time data stream [18]. On Storm platform, the specific stream grouping modes can be customized by implementing the Custom-StreamGrouping 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 configurations file Storm.yaml in which deployment strategy is employed.

With the fluctuating rate of data streams 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 cause 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 deployment on the computing nodes. In addition, the number of instances for each vertex can be further adjusted as needed.

As shown in Figure 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.
B. Lightweight Load Balancing Based Redirection Among Instances

In order to improve the load balancing of data center DC, redirection of data stream \( ds_{(i)} \) 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 \( v_j \), each data tuple \( dt \) in \( ds_{(i)} \) is emitted to one of the \( n \) instances of \( v_j \) during \([t_s, t_e]\) by load based redirection function \( f(lb_{v_j, [t_s, t_e]})\). For the \( k \)th instance, the load-based redirection function \( f(lb_{v_j, [t_s, t_e]})\) is defined by (12).

\[
f(lb_{v_j, [t_s, t_e]}) = \frac{\text{length}_{v_j, k}}{\sum_{j=1}^{n} \text{length}_{v_j, j}}
\]

where \( \text{length}_{v_j, k} \) is the available queue length of the \( k \)th 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 being 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 \( ds_{(i)} \) 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 \( ds_{(i)} \).
5. **return** input data tuple assigned to each instance of \( v_j \).
6. **end if**
7. **for** each computing node \( cn \), running instance **do**
8. **Set** configuration parameters minimum load ratio \( h_{min, cn} \) and maximum load ratio \( h_{max, cn} \).
9. **end for**
10. **while** new data tuple is produced by \( v_j \) **do**
11. **for** each instance \( v_j \) 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 \( v_j \) of \( v_j \) **do**
16. **if** the queue of the \( k \)th instance \( v_j \) is empty **then**
17. **Emit** the new data tuple to \( v_j \).
18. **end if**
19. **if** the queue of the \( k \)th instance \( v_j \) is full **then**
20. **Skip** \( v_j \).
21. **end if**
22. **end for**
23. **Sort** all \( n \) instances by available queue length in descending.
24. **Calculate** the load-based redirection function \( f(lb_{v_j, [t_s, t_e]}) \) 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
The input of this algorithm includes the output data stream \( ds_{(i)} \) 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 \( l_{\text{min},c} \) and maximum load ratio \( l_{\text{max},c} \) 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 \( v_j \).

C. Logical Ring-Based State Management

Besides data stream redirection, vertex statement management also affects the computing performance. For example, if a stateful vertex \( v_i \) fails, its state will be lost, undermining the accuracy of data tuple processing [21] [22]. In order to maintain the state of stateful vertices, checkpoint mechanism has been employed in Storm for state management [8]. However, checkpoint data is stored in external storage, causing longer fault tolerance time to restore the system state. It is also possible that part of status data was 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.

For a stateful vertex \( v_i \), 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 Figure 15, the \( k \)th instance \( v_{sk} \) of \( v_i \) manages its own state, simultaneously synchronizing its own state with the \((k+1)\)th instance \( v_{sk+1} \) and the \((k-1)\)th instance \( v_{sk-1} \).

![Figure 15. State backup for a stateful vertex.](image)

If an instance node fails, its state still exists and does not stop the system from further processing. As shown in Figure 16, when the \( k \)th 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 \( v_{sk+1} \) take over the work of \( v_{sk} \), and receive synchronization state information from the \((k-1)\)th instance \( v_{sk-1} \). If \( v_{sk+1} \) fails, synchronization state information is received from instance \( v_{sk} \). Another option is to create a new instance \( v_{sk} \), recovering the state of \( v_{sk} \) from \( v_{sk-1} \) and \( v_{sk+1} \), and to take over the work of the failed instance \( v_{sk} \).

If \( v_{sk} \) fails, recovering the state from \( v_{sk-1} \).

![Figure 16. State recovery for a failed stateful vertex.](image)

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 \( v_i \), input data tuple of \( v_i \), and state of computing node for each instance.
2. **Output**: state dependency of all instances of \( v_i \).
3. if the number of instances is 1 then
4. Create one instance and make the number of instances of \( v_i \) meet the condition \( n \geq 2 \).
5. end if
6. Built a logical ring for the \( n \) instances of \( v_i \) 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 \( v_i \) 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 \((k+1)\)th instance \( v_{sk+1} \) of \( v_i \).
11. end if
12. if the \( k \)th instance \( v_{sk} \) failed then
13. if the number of instances of \( v_i \) is larger than 1 then
14. Select a new computing node, create a new
instance $v_{d\_t}$, recovery the state of $v_{d\_t}$ from $v_{d\_t+1}$, and take over the work of the failed instance $v_{d\_t}$.

15. **else**
16. Select the $(k+1)th$ instance $v_{d\_t(k+1)}$ to take over the work of $v_{d\_t}$, and make $v_{d\_t(k+1)}$ synchronize status from the $(k\_1)th$ instance $v_{d\_t(k-1)}$.
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 $v_{d\_t}$ to the $(k+1)th$ instance $v_{d\_t(k+1)}$, according to the condition that synchronization interval of the $kth$ instance $v_{d\_t}$ is greater than the minimum synchronization period $7$ and the state of $v_{d\_t}$ 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.

D. Deployment Optimization

Deployment optimization is the key to accomplish computing node-level data stream redirection, as well as to optimize resource usage, minimize the load balancing of data center $DC$, and to improve throughput and response time of running 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 vertices 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.

**Algorithm 3**: Deployment optimization algorithm.

1. **Input**: $m$ stream application set $G_{sa} = \{G_1, G_2, \ldots \}$, $num$ available computing nodes in data center $DC = \{cn_1, cn_2, \ldots \}$, and input data stream $\{a_0(G_1), r_0(G_2), \ldots \}$ for each stream application.
2. **Output**: Deployment of stream applications on available computing nodes.
3. **if** stream application set $G_{sa}$ or available computing nodes is null **then**
4. **return** null.
5. **end if**
6. 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 $l_{\min, cn_i}$ and maximum load ratio $l_{\max, cn_i}$ for each computing node in $DC$.
7. Set the initial number of instances per vertex for each stream application, and the incremental proportion $p_{lin} \%$ of the number of instances for each vertex.
8. **for** each stream application in $G_{sa}$ **do**
9. Sort all computing nodes by available CPU in descending order.
10. Deploy each vertex of the stream application to an available compute node in a topologically ordered manner by the first fit strategy.
11. **if** the number of instances of a vertex is larger than one **then**
12. Keep all the instances of any vertex deployed to different compute nodes.
13. **end if**
14. Update the available CPU of the deployed computing nodes.
15. **end for**
16. **while** $lb_{DC,[j,i]}$ is out of the range $[lb_{\min, DC}, lb_{\max, DC}]$ or $l_{\min, cn_i}$ is out of $[l_{\min, cn_i}, l_{\max, cn_i}]$ 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**
17. **if** $lb_{DC,[j,i]}$ is out of the range $[lb_{\min, DC}, lb_{\max, DC}]$ **then**
18. **for** each stream application in $G_{sa}$ **do**
19. Get the output data stream $ds_{(j,i)}$ of upstream $v_j$ and instances of downstream $v_j$.
20. Redirect input data tuple for each instance of $v_j$ by Algorithm 1.
21. **end for**
22. **end if**
23. **if** $l_{\min, cn_i}$ is out of the range $[l_{\min, cn_i}, l_{\max, cn_i}]$
then
24. for each instance \( v_j \) on \( cn \) do
25. Get the output data stream \( ds_{s(o)} \) of upstream
26. \( v_j \) and instances of downstream \( v_j \).
27. Redirect input data tuple for each instance of
28. \( v_j \) by Algorithm 1.
29. if any latency of application \( G_j \) is greater than the
30. maximum latency \( l_{\text{max}}(G) \) then
31. Sort all computing nodes by available CPU in
32. descending order.
33. Pre-deploy each vertex in the stream application
34. to an available compute node in a topologically
35. ordered manner by the first fit strategy.
36. Calculate the difference between existing deployment
37. and pre-deployment, and redeploy the vertices accordingly.
38. end if
39. if the throughput of any application \( G_j \) is less
40. than the minimum throughput \( T_{\text{min}}(G) \) then
41. Increase the number of instances by \( p_{\text{inc}} \) % for
42. each vertex.
43. Sort all computing nodes by available CPU in
44. descending order.
45. Deploy new instances in the stream application
46. to an available compute node in a topologically or-
47. dered by the first-fit strategy.
48. end if
49. for each stream application \( G_j \) in \( G_{\infty} \) do
50. for each stateful vertex \( v_i \) in \( G_j \) do
51. Build state dependency of all instances of \( v_i \)
52. by Algorithm 2.
53. end for
54. end for
55. Update state of stream application set \( G_{\infty} \), available
56. computing nodes in \( DC \).
57. Monitor input data stream \( \{r_{d_i}(G_i), r_{d_i}(G_j), \ldots \} \) for each stream application.
58. end while
59. return deployment of stream applications on avail-
60. able computing nodes.

The input of this algorithm includes \( m \) stream applications set \( G_{\infty} = \{G_1, G_2, \ldots \} \), \( num \) available computing nodes in data center \( DC = \{cn_1, cn_2, \ldots \} \), input data stream \( \{r_{d_i}(G_i), r_{d_i}(G_j), \ldots \} \) 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 \( b_{dc}[t.i] \) of the data center during \( [t_i,t_f] \) by redirecting the input data tuple at instance level with Algorithm 1. Step 23 to step 28 keep load ratio \( b_{cn}[t,i] \) of computing node \( cn \) during \( [t_i,t_f] \) within range \( [b_{\text{min}}, b_{\text{max}}] \) 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 \( p_{\text{inc}} \) % 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.

VI. 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.

A. 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 moni-

tor 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 working 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. Moreover, two types of stream applications \( \text{Top}_N \text{ and WordCount} \) are submitted to the data center.

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

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

![Logical graph of Top-N](image1)

**Figure 17. Logical graph of Top-N**

**Table 1 Function of vertex in the logic graph of Top-N**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_{\text{reader}})</td>
<td>read word from data stream</td>
</tr>
<tr>
<td>(v_{\text{count}})</td>
<td>count word</td>
</tr>
<tr>
<td>(v_{\text{rank}})</td>
<td>rank word by count</td>
</tr>
<tr>
<td>(v_{\text{merge}})</td>
<td>merge all ranks from upstream</td>
</tr>
</tbody>
</table>

![Logical graph of WordCount](image2)

**Figure 18. Logical graph of WordCount**

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

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_{\text{reader}})</td>
<td>read sentence from data stream</td>
</tr>
<tr>
<td>(v_{\text{split}})</td>
<td>split word of sentence</td>
</tr>
<tr>
<td>(v_{\text{count}})</td>
<td>count word</td>
</tr>
</tbody>
</table>

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

**Table 3 Parameter settings**

<table>
<thead>
<tr>
<th>Notations</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l_{\text{min,DC}})</td>
<td>0</td>
<td>min load balancing of DC</td>
</tr>
<tr>
<td>(l_{\text{max,DC}})</td>
<td>0.3</td>
<td>Max load balancing of DC</td>
</tr>
<tr>
<td>(l_{\text{min,cn}})</td>
<td>0</td>
<td>min load ratio of computing node (cn_i)</td>
</tr>
<tr>
<td>(l_{\text{max,cn}})</td>
<td>0.99</td>
<td>max load ratio of computing node (cn_i)</td>
</tr>
<tr>
<td>(p_{\text{inc}})</td>
<td>40</td>
<td>incremental % of the number of instances</td>
</tr>
</tbody>
</table>

**B. Performance Results**

The experimental setting contains four evaluation parameters: average latency \(AL\), average throughput \(AT\), average load balancing \(l_{\text{avg,DC}}\) of data center \(DC\), and average load ratio \(l_{\text{avg,cn}}\) 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.

![Real-time latency at different input rates](image3)

**Figure 19. Real-time latency at different input rates.**

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 Figure 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 milliseconds and 52 milliseconds, respectively, reducing 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 milliseconds and 149 milliseconds, respectively, reducing 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, and it is reduced by more than 20%.

![Average latency at different input rates](image4)

**Figure 20. Average latency at different input rates.**

With the increase of input rates, the average latency increases under both deployment strategies. For a given input rate, Dr-Stream has a shorter average than the default Storm strategy. As shown in Figure 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 milliseconds and 98 milliseconds, respectively, reducing 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 milliseconds and 595 milliseconds, respectively, reducing it by 37%. The higher the data rate is, 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 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 Figure 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, improving it by 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, improving 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, and it 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 rates, the difference between the two strategies becomes apparent. The higher the data rate is, the more significantly this difference is noted. As shown in Figure 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, improve 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, improving it by 75%. The difference becomes rather significant.

(3) Average load balancing $lb_{\text{avg, DC}}$ of data center DC. The average load balancing $lb_{\text{avg, DC}}$ 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 $lb_{\text{avg, DC}}$ 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 Figure 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, reducing it by 36%, respectively. When the rate of input data stream change to 5000 tuples/s, the average load balancing become 0.3 and 0.54, respectively, reducing 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, and it 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 \( lb_{\text{max,}DC} \) of data center DC by dynamically redirecting of real-time data streams. As shown in Figure 24, when the rate of input data stream is 1500 tuples/s, the average load balancing of Dr-Stream and that of the default Storm deployment strategy are 0.16 and 0.27, respectively, reducing 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 \( lr_{\text{avg,}cn} \). The average load ratio \( lr_{\text{avg,}cn} \) of a computing node \( cn \) reflects the loading state of its CPU queue. If \( lr_{\text{avg,}cn} \geq 1 \), then computing node \( cn \) is overloaded, lowering the system performance. It is better to have the average load ratio \( lr_{\text{avg,}cn} \) kept within the range of (0, 1). Given the average load ratio \( lr_{\text{avg,}cn} \in (0, 1) \), the larger the average load ratio \( lr_{\text{avg,}cn} \), 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 \( lr_{\text{max,}cn} \) of computing node. As shown in Figure 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 any computing node with excessive load 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.

VII. 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.

A. 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 have 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 differ-
ent 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 well 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.

B. 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 modelled as Mixed-Integer Linear Program, LP solver was used to improve the load distribution in computing cluster.

In order to improve the load balancing for stateful applications in Storm system, a locality-aware routing strategy was proposed in [37], improving the data stream locality for stateful stream processing applications. Data migration is 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 is 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 works. However, most of the them achieve 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 also 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.

C. 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 estimates the resources utilization was proposed and implemented on Storm platform.

In response to the uncertainty and complexity of stream 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 are 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 Storm platform from both the resource optimization perspective and the data stream redirection perspective. It provides significant performance improvements on metrics such as system latency, throughput, and load balancing.

VIII. 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 elastic stream computing system is 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 stream 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, 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, data stream model and 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 time; managing the state 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.

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 under Grant No. 2652018081; Australian Research Council (ARC) Discovery Project; Promoting the Developing University Intension–Disciplinary Cluster No. 5211910940, Qin Xin Talents Cultivation Program, and Beijing Information Science & Technology University No. XQTCP B201908.

REFERENCES


| **Shang Gao** received her Ph.D. degree in computer science from Northeastern University, 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. 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 currently holds 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 with Hangzhou Dianzi University. She received his Ph.D. degree in computer science from Northeastern University, China in 2007. Her current research areas include Distributed Computing, 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 128, 85,600+ 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. |