A Hierarchical Near-Source Grouping Strategy for Elastic Stream Computing Systems

Minghui Wu, Dawei Sun, Shang Gao, Member, IEEE and Rajkumar Buyya, Fellow, IEEE

***Abstract*—Effective task scheduling in stream computing systems can reduce the latency by minimizing inter-node communication. However, this approach often requires restarting tasks to change their deployment locations, resulting in significant system overhead and making it inadequate especially in dynamically changing data stream environments. To address this issue, we propose Ns-Stream, a hierarchical data scheduler that dynamically adjusts data distribution weights between near-source and off-source tasks. Our solution is discussed as follows: (1) We observe that communication overhead from off-source data processing significantly impacts system latency when tasks' resources are ample. However, as the resources become limited, the computational power required by tasks becomes the key constraint on system performance. (2) During initialization scheduling, we deploy tasks with potential communication to the same node using the graph convolutional network, thus avoiding the need for runtime task scheduling. (3) We dynamically adjust data distribution weights between near-source and off-source tasks based on their computing capabilities, prioritizing local processing of data tuples (within the same worker and node) to optimize resource utilization and reduce data transmission overhead. (4) Experimental results demonstrate significant improvements made by Ns-Stream: reducing maximum system latency by 40% and increasing maximum throughput by 55% compared to existing state-of-the-art works.**

***Index Terms*—Stream computing systems, Data grouping, Task deployment, Initialization scheduling, Weight assignment**

I. Introduction

PROCESSING continuous data streams with low latency has become crucial for applications such as Internet of Things (IoT), traffic monitoring, telecommunications, and health care [1], [2]. In these scenarios, users rely on immediate data feedback for swift decision-making. If the system latency is too high, stream computing may lose its advantage compared to batch processing and disappoint users expecting real-time performance [3]. Furthermore, minimizing data processing latency in stream computing systems accelerates task completion, frees up system resources, and consequently reduces the demand on computing, storage, network, and other resources [4]. Therefore, operating costs can be effectively cut down while resource utilization and energy efficiency are enhanced.

Minimizing inter-node communication within modern stream computing systems has been demonstrated as a highly effective means of enhancing system performance [5], [6]. The underlying principle of this optimization lies in the dynamic monitoring of communication volumes between tasks during runtime, allowing for the strategic placement of communication-intensive tasks on the same worker or compute node. This approach significantly reduces communication overhead, leading to improved system throughput and overall efficiency.

However, when a rescheduling event is triggered to improve communication speed within a stream computing system, some schedulers choose to first terminate the entire topology and then restart it [7], [8]. This might not be the best solution because it can lead to the lose of important information that was being processed. Until it fully restarts and recovers this information, the topology cannot continue processing data, which can significantly slow down its operations. To solve this problem, some schedulers [6], [9]–[11] make small changes to the deployment location of tasks while the topologies are still running. Although this can solve the problem of short-term processing interruptions caused by global scheduling, making these small changes also requires restarting some tasks in the topology, which consumes time and resources. Furthermore, the constant need to trigger rescheduling strategies due to fluctuating data streams can negatively impact the system's performance.

To address the challenges posed by frequent rescheduling and the associated costs of task restarts, data grouping strategies for stream computing systems have been refined by evenly distributing workloads and enhancing computational efficiency among tasks at runtime [12]–[15]. Two commonly used grouping strategies are random grouping and key grouping. Random grouping evenly distributes data stream among tasks using a round-robin approach to balance the workload. Key grouping, on the other hand, emits data based on specific key values, ensuring that related data is processed by the same task. Some studies [16], [17] explore a hybrid approach that combines both strategies to further optimize workload balancing. This hybrid method assigns data with frequently occurring keys (hot keys) and infrequently occurring keys (rare keys) to different tasks using random and key grouping techniques. The goal is to quickly and accurately identify the most frequent keys so that the grouping process can be faster and more effective.

In a distributed stream computing system, tasks are in-

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

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

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

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

stantiated by operators (basic functional units of a streaming application) to process allocated workloads. These tasks of operators are deployed to workers in compute nodes, where each worker executes a portion of data processing logic defined in the streaming application. Balancing workload among tasks in operators through effective data grouping can enhance system performance [13], but the communication cost between tasks is often overlooked. Through experiments, we have observed that, when workers' resources are abundant, the communication cost between tasks tends to be the primary factor affecting system latency. However, in scenarios where the workers' resources are limited, the computational cost of data streams becomes the dominant factor. Therefore, blindly pursuing an even data distribution among tasks is not always the best solution. Instead, if tasks have sufficient computational resources, skewing the workload towards near-source tasks (located closer to the data emitting tasks) can often reduce system latency.

Motivated by the above observations, we have designed a hierarchical near-source grouping strategy, named Ns-Stream. Ns-Stream dynamically reschedules data tuples at runtime based on the relative location between emitting and receiving tasks. It prioritizes dispatching data tuples to near-source tasks to maximize intra-node communication load when worker resources are ample. Otherwise, Ns-Stream sequentially schedules the overloading data tuples to other workers on the same node and then on different nodes. This hierarchical data grouping strategy results in different data distribution sizes at different deployment locations. To achieve this, we construct a resource constraint model to ensure sufficient resource allocation for near-source tasks processing data streams. Furthermore, we develop a graph convolutional network model to continuously aggregate information about tasks to optimize the deployment during the initial scheduling phase, pre-placing tasks with potential communication on the same worker and compute node.

Our contributions are summarized as follows:

(1) We observed that when tasks have ample resources, the communication overhead from off-source data processing is the main factor impacting system latency. However, when the resources are limited, the computational resources of tasks become the key constraint on system performance.
(2) Leveraging a graph convolutional network during the initialization scheduling phase, tasks with potential communication demands are strategically placed to the same worker and node where possible, eliminating the need for runtime task scheduling.
(3) Data distribution weights between near-source and off-source tasks are dynamically adjusted based on their computational resource requirements, prioritizing local tuple processing within the same worker and node to optimize resource usage and minimize data transmission overhead.
(4) The proposed Ns-Stream is integrated into the Apache Storm platform and evaluated on metrics such as system throughput and latency. Experimental results show that Ns-Stream provides notable advancements compared to existing state-of-the-art works.

The rest of this paper is organized as follows. Section III discusses the impact of communication overhead on system latency and our motivation. Section IV introduces the relevant stream computing system models, including the application model, communication model, and data grouping model. Section V formalizes the problems related to initialization scheduling and data stream redirection. Section VI introduces the Ns-Stream strategy and its main algorithms. Section VII evaluates the performance of Ns-Stream. Section II presents related work, and Section VIII concludes our work along with future directions.

II. Related work

In this section, we present a review of cutting-edge research in two relevant fields: task scheduling for stream processing and data stream grouping. A comparison between our work and the relevant research is summarized in Table I.

A. Task scheduling for stream processing

When tasks of a streaming application require data exchange or communication, deploying them on different compute nodes incurs network communication overhead. This not only increases the latency of data transmission but also consumes valuable network resources. By placing communication-intensive tasks on the same node during task scheduling, data transmission between nodes can be effectively reduced.

Communication overhead. P-Schedule [8] models the streaming application as a DAG topology during runtime and partitions the DAG into subgraphs, deploying communication-intensive tasks on the same compute nodes. SP-Ant [6] identifies the most effective operator assignment plan by collocating operators with high communication overhead on the same worker using a bin-packing algorithm, and reschedules only the less communicative operators through an evolutionary ant colony optimization algorithm. However, both methods fail to consider the trade-off between communication cost and computational load, which can result in overloaded nodes when compute-intensive tasks are deployed to the same worker.

Resource efficiency. D-Storm [21] dynamically aligns the resource requirements of streaming applications with the available resources of compute nodes. It formulates the scheduling problem as a bin-packing variant, and introduces a heuristic-driven algorithm to minimize inter-node communication. CE-Storm [18] prioritizes nodes within the cluster based on the associated costs of resource usage, energy consumption, and communication. Tasks are assigned to higher-priority nodes to improve the cost-effectiveness of the Storm cluster. However, both approaches lack the ability for tasks to dynamically adjust the number of tuples they process based on their own processing capabilities, which may lead to inefficiencies under varying workloads.

Migration overhead. CAOM [19] employs a bottleneck operator detection mechanism to identify maximum operator capacities of operators. It avoids repeated migration operations to reduce interruptions and considers varying data generation rates to select optimal migration times. While this approach

TABLE I
RELATED WORK COMPARISON

Aspect	Related Work							
	SP-ant [6]	P-scheduler [8]	Hone [14]	Pstream [16]	CE-Storm [18]	CAOM [19]	FlexD [20]	Our work
Topology redeployment	✔	✔	✗	✗	✔	✔	✗	✗
Task restart	✔	✔	✗	✗	✔	✔	✗	✗
Communication awareness	✔	✔	✗	✗	✔	✗	✗	✔
Data grouping	✗	✗	✔	✔	✗	✗	✔	✔
Inter-task load balancing	✗	✗	✔	✔	✗	✗	✔	✗

effectively reduces migration overhead by selecting optimal timings, the costs associated with task redeployment remain unavoidable.

These aforementioned solutions offer valuable insights into the scheduling problem. However, in distributed stream computing systems, adjusting task deployment locations at runtime typically leads to unavoidable interruptions in data processing. In contrast, Ns-Stream maintains continuous and stable data stream processing by regulating inter-node and inter-worker data flow, thereby enabling dynamic resource allocation within the cluster.

B. Data stream grouping

Data stream grouping serves as a key technique for parallel stream computing. By partitioning data streams into disjoint sub-streams based on grouping keys, messages with the same key are directed to the same sub-stream. This facilitates the concurrent processing of these sub-streams across multiple compute nodes, enabling distributed resource utilization and improved scalability.

Workload balancing. MIXED [13] combine hash-based routing with explicit key-based routing, designating destination worker threads for certain keys while hashing others. Hone [14] effectively schedules tuples to minimize queue backlogs and balance backlog across tasks, mitigating stragglers when workloads exhibit variance. Ms-Stream [22] adopts a hierarchical strategy with lightweight two-level grouping to mitigate stragglers and balance workloads. It also considers cross-node task placement, ensuring that communication- and computation-intensive tasks are appropriately deployed. However, workload balancing strategies such as MIXED [13], Hone [14], and Ms-Stream [22] may introduce unnecessary communication overhead by redistributing tuples to remote tasks, even when local tasks are sufficiently capable of processing the upstream workload.

Hot key identification. Pstream [16] introduces a popularity-aware differentiated stream computing system. This system employs shuffle grouping to allocate hot keys identified by a lightweight probabilistic counting scheme, while using key grouping for rare keys. Dalton [23] provides rewards for hot keys based on reinforcement learning that captures load variations for load balancing among operator tasks, and optimizes the learned model through experience gained by assigning tuples. FlexD [20] utilizes hash grouping to partition low-frequency tuples and uses a progressive splitting method to dynamically update the partitioner based on changes in key frequency for high-frequency tuples. Despite their adaptiveness, these approaches can similarly result in excessive communication overhead by redistributing data solely for the purpose of load balancing, even when local processing capacity is not fully utilized.

In summary, effective data grouping can enhance system performance by evenly distributing task workloads. However, inter-task communication overhead incurred by data transfers is often overlooked. When computational resources are abundant, this communication overhead becomes the primary factor affecting system latency rather than the workload distribution. Consequently, blindly pursuing even data distribution among tasks without considering the communication overhead may not always be an optimal strategy.

III. MOTIVATION

A series of experiments have been conducted on the popular stream computing platform Storm [24] to investigate the impact of inter-task communication overhead on system latency, thus motivating this research. The experimental cluster consists of 3 machines, each equipped with an Intel(R) Xeon(R) X5650 CPU (dual-core, 2.4 GHz), 2 GB of RAM, and a 100 Mbps Ethernet network interface card. Word Count (WCount) is a fundamental and widely-used benchmark for stream computing system performance evaluation and analysis. We employ its common rhombus topology (Fig. 1) as our test case.

To simulate the impact of varying inter-task (i.e., inter-node and intra-node (inter-worker)) communication volumes on system latency, we generate synthetic datasets following the Zipf distribution, similar to [25], [26]. Specifically, we set the Zipf coefficient to -2, -1, -0.5, 0, 0.5, 1, and 2, denoted as Zipf-2, Zipf-1, Zipf-0.5, Zipf0, Zipf0.5, Zipf1, and Zipf2, respectively. The Zipf coefficient governs the skewness

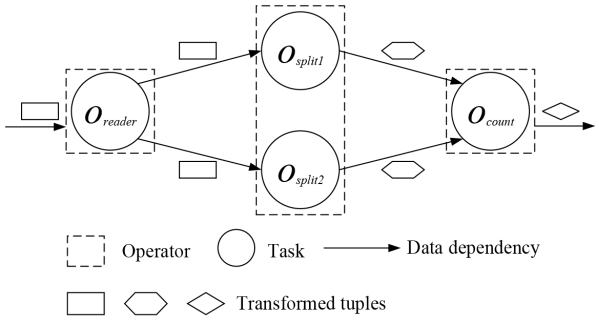


Fig. 1. Instance topology of WCount.

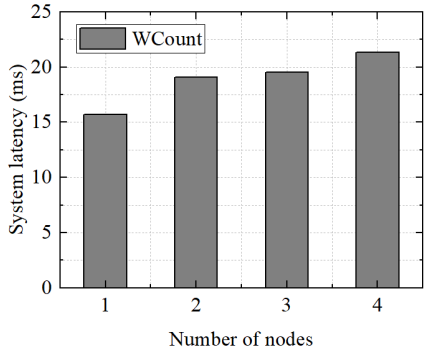


Fig. 2. System latency with different numbers of nodes.

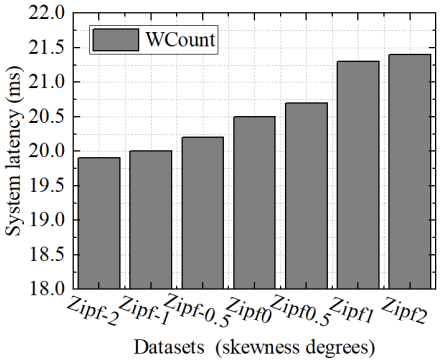


Fig. 3. System latency with workers on the same node and varying skewness degrees.

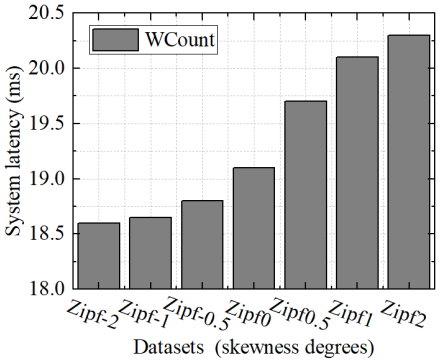


Fig. 4. System latency with workers on different nodes and varying skewness degrees.

degree of these synthetic datasets, wherein a larger coefficient value corresponds to a higher degree of skewness in the data distribution.

A. Observations

By adjusting the number of deployed nodes and the skewness level of data stream, we examine the impact of inter- and intra-node communication on system latency. The following experiments are conducted when the streaming application has relatively ample resources and a stable input stream rate of 100 tuples/s: (1) latency with different numbers of nodes, (2) latency with workers on the same node and varying skewness degrees, (3) latency with workers on different nodes and varying skewness degrees. Moreover, we deploy the streaming application to a single worker to observe the impact of worker's input load on system latency under varying input rates and constrained resources, which is addressed in experiment (4) latency with the same worker and different input rates.

(1) Latency with different numbers of nodes. As depicted in Fig. 2, a positive correlation can be observed between the system latency and the number of nodes. With the node count increasing from 1 to 4, the system latency exhibits a gradual rise, escalating from 15.7 ms to 21.3 ms. This trend suggests that as the number of used nodes scales up, the augmented inter-node communication overhead contributes to the elevated system latency. This finding underscores the importance of minimizing inter-node data transfer to optimize system performance.

(2) Latency with workers on the same node and varying skewness degrees. We deploy the streaming application on a single compute node, with o_{split1} and o_{split2} hosted by different workers. The degree of data skewness is adjusted to modulate the inter-worker communication volume. As shown in Fig. 3, a gradual escalation in system latency can be observed, rising from 19.9 ms for Zipf-2 to 21.1 ms for Zipf2, as the datasets become increasingly skewed (or the inter-worker communication volume increases). This trend illustrates that on the same node, an increase in inter-worker communication incurs a higher overall system latency. The findings underscore the considerable impact of inter-worker communication overhead on the aggregated system latency, particularly when the communication volume (as indicated by skewness degree) is substantial, thereby increasing the latency penalty. Proper control and optimization of inter-worker communication can potentially enhance system performance.

(3) Latency with workers on different nodes and varying skewness degrees. We deploy the streaming application across two compute nodes, with o_{split1} and o_{split2} hosted by workers on different nodes. The degree of data skewness is adjusted to modulate the inter-worker communication volume across nodes. As shown in Fig. 4, a similar rising trend in system latency can be observed, increasing from 18.6 ms for Zipf-2 to 20.3 ms for Zipf2, as the datasets become increasingly skewed. This trend, similar to the impact of communication overhead among workers on the same node, highlights that the increased inter-node communication overhead contributes to higher overall system latency. These findings emphasize the importance of proper control and optimization of inter-node communication as a key approach to further improve the performance of distributed systems.

(4) Latency with the same worker and different input rates. To simulate the impact of resource load on system latency, we deploy the streaming application to a single worker. The Zipf coefficient is set to 0 to maintain a balanced data stream. As shown in Fig. 5, an obvious upward trend in system latency can be observed as the input rate escalates from 100 tuples/s to 400 tuples/s, surging from 16.1 ms to 46.6 ms. Notably, at the high input rate of 400 tuples/s, the

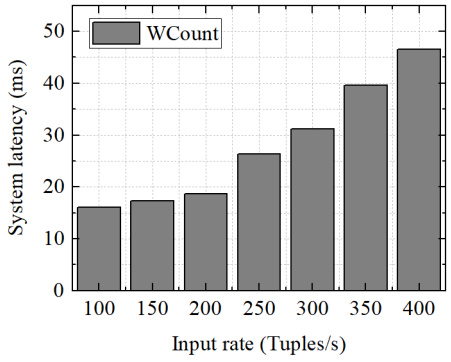


Fig. 5. System latency with the same worker and different data input rates.

latency has reached to 46.6 ms. It suggests that once the input rate exceeds a certain threshold, the single worker's resources become inadequate to meet the data processing demands. The accumulation of substantial input data in the worker leads to a drastic increase in processing latency. These findings highlight the importance of proper input rate control in stream computing systems to ensure low-latency operation. Any input load exceeding the processing capacity of the allocated resources can potentially trigger severe latency degradation.

The above experimental results reveal that when an application's allocated resources are abundant, the overall system latency exhibits a gradual increase as the volumes of inter-node communication and intra-node communication among workers escalate. However, in scenarios where the application's resources are constrained, its resource load becomes the primary factor limiting the system performance.

B. Motivations

Based on the aforementioned observations and analysis, it can be seen that system latency can be affected by several factors, including the volume of intra-node communication (i.e., inter-worker communication within nodes), inter-node communication overhead, and the utilization of worker resources by incoming data streams. It is wise to consider changing the task deployment locations at runtime to minimize these communication costs. However, as discussed earlier, both global and incremental online task scheduling methods incur expensive costs and struggle to maintain the execution information of tasks. To achieve low overhead and latency, a hierarchical near-source data grouping approach may help. Our motivations can be summarized as follows:

(1) Given the statistical information on task processing data, how can we control data distribution weights to minimize the communication overhead between compute nodes?

(2) How can we achieve a trade-off between near-source and off-source data processing to efficiently utilize worker and node resources?

(3) Given the maximum data processing threshold, how can we adjust the input data rates for near-source tasks to enhance the system's low-latency processing capability?

IV. System Model

Before discussing the Ns-Stream strategy and its related algorithms, we first introduce the relevant models, including the streaming application model, the communication model, and the data grouping model.

A. Streaming application model

Each user-submitted streaming application can be represented as a directed acyclic graph (DAG) [27], denoted as $G = (O(G), E(G))$, consisting of an operator set $O(G)$ and a directed edge set $E(G)$. The operator set $O(G) = \{o_i | i \in \{1, \ldots, n\}\}$ comprises a finite number of n operators, where each operator o_i represents an operation with a special function. The edge set $E(G) = \{co_{u,v} | u, v \in \{1, \ldots, n\}, u \neq v\}$ is a finite set of directed edges, with weights assigned to represent the communication costs between operators. Once the user constructs the DAG and submits it to the data center, multiple tasks are instantiated for each operator o_i, and all these tasks execute the same functional logic defined by operator o_i.

Both the DAG and its instantiated tasks compose the streaming application model, which represents the logical structure of a streaming application. Fig. 1 shows a sample logical topology with 3 operators: o_{reader}, o_{split} and o_{count}. Each of o_{reader} and o_{count} has 1 tasks, and o_{split} has 2 task, including o_{split1} and o_{split2}. Tasks of the same operator conduct the same data processing logic. For example, tasks o_{split1} and o_{split2} execute the same function.

B. Communication model

We store the tasks of a streaming application DAG into a set $T = \{o_{1,1}, o_{1,2}, ..., o_{i,k}, ..., o_{n,j}\}$, and construct a matrix E to represent the communication costs among the tasks in the set. This matrix can be described by (1).

$$E = [e_{i,k}] = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,s} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ e_{s,1} & e_{s,1} & \cdots & e_{s,s} \end{bmatrix}, \tag{1}$$

where s denotes the number of all tasks in the streaming application DAG. $e_{i,k}$ denotes the communication cost between the i-th task and the k-th task in the task set T.

As the data stream rate may experience transient fluctuations, to effectively mitigate their impact, $e_{i,k}$ represents the average communication cost over the time interval $[t_d, t_u]$. It can be calculated by Eq. (2).

$$e_{i,k} = \frac{\int_{t_d}^{t_u} e_{i,k}^t dt}{t_u - t_d}, \tag{2}$$

where $e_{i,k}^t$ denotes the communication cost between the i-th task and the k-th task in the task set T at time t, $t \in [t_d, t_u]$.

Communication demand is primarily generated by tasks deployed in different locations, including both intra-node and inter-node communication. For example, if two tasks are deployed on the same node, only intra-node communication will occur. If two tasks are deployed on different nodes, only inter-node communication will occur. In distributed stream computing systems, the key to data processing efficiency lies in how to manage and optimize communication between nodes. To better understand this, we introduce an important theoretical foundation, as described in Theorem 1.

Theorem 1. *In a distributed stream computing system, if the data input rate of a DAG remains stable, its intra-node communication exhibits an inverse relationship with its inter-node communication: maximizing the former minimizes the latter.*

Proof. Under a stable data input rate, a consistent task communication cost matrix E can be obtained. The total

communication volume TCT among all tasks in the DAG can be calculated by Eq. (3).

$$TCT = \sum_{i=1}^{s} \sum_{k=1}^{s} e_{i,k}. \tag{3}$$

We represent the task deployment on a compute node cn_m with vector q_m of length s, $m \in \{1, \ldots, M\}$, where M represents the total number of compute nodes. In q_m, the i-th element is set to 1 if the i-th task in set T is deployed onto this m-th node, to 0 otherwise. For example, Eq. (4) represents a few adjacent tasks in T running on node cn_m.

$$q_m = (\underbrace{0, ..., 0, \overbrace{1, ..., 1}^{cn_m}, 0, ..., 0}_{s})^{\mathrm{T}} \tag{4}$$

The communication cost $c(cn_m, cn_m)$ among tasks within node cn_m can be calculated by Eq. (5).

$$c(cn_m, cn_m) = {q_m}^{\mathrm{T}} \cdot E \cdot q_m, \tag{5}$$

The total communication cost TCN between nodes (or inter-node communication cost) can be obtained by subtracting all the intra-node communication costs from the total communication cost among all tasks of the streaming application, which can be calculated by Eq. (6).

$$TCN = TCT - \sum_{m=1}^{M} c(cn_m, cn_m). \tag{6}$$

In the scenario where the input rate is stable, the total tuple transmission within the system remains stable, i.e., TCT can be considered as a constant, Eq. (6) reveals an inverse relationship between intra-node and inter-node communication. Specifically, an increase in local tuple processing boosts intra-node communication, and lead to a decrease in inter-node communication.

C. Data grouping model

Data grouping is the process of partitioning data tuples from a data stream ds according to a specific logic or condition [28]. We define this specific logic or condition as a grouping function $GF(ds)$. This grouping function determines how the data tuples are divided or clustered into distinct subsets or groups within the overall data stream ds. In a streaming application DAG , it is important to dynamically dispatch data tuples to appropriate tasks of each operator for performance purposes.

As shown in Fig. 6, an upstream task emits data tuples to task set $\{o_{i,1}, o_{i,2}, o_{i,3}, o_{i,4}, o_{i,5}\}$ of operator o_i. Depending on the grouping strategy used, varying numbers of tuples are received by the tasks in $\{o_{i,1}, o_{i,2}, o_{i,3}, o_{i,4}, o_{i,5}\}$. The tasks incur different communication overheads and resource consumption when processing the tuples. These communication overheads and the resource consumption considerably impact the system latency and should not be overlooked.

To better optimize the overheads and resource consumption, we introduce two concepts: near-source data processing and off-source data processing.

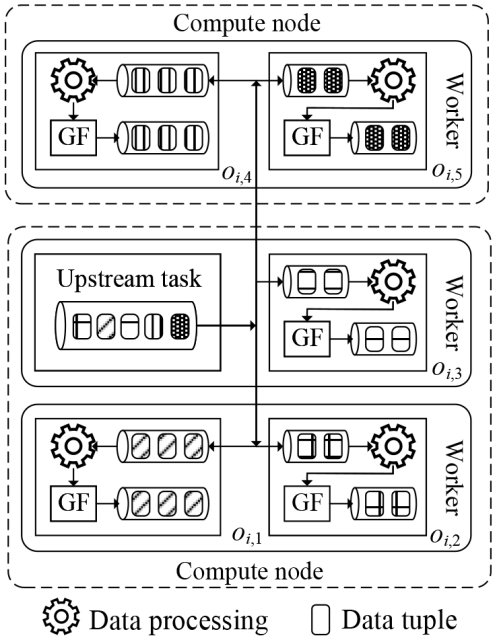


Fig. 6. Data stream grouping among tasks of operator o_i.

Definition 1. *Near-source data processing.* If two communicating tasks are deployed to the same compute node, data tuples transmitted between tasks (in the same worker or different workers) are considered to be processed near-source.

For example, in Fig. (6), the data grouping between the Upstream task and tasks $O_{i,1}$, $O_{i,2}$ and $O_{i,3}$ is near-source. The latency between the Upstream task and $O_{i,3}$ will be lower than those of $O_{i,1}$ and $O_{i,2}$, as the deployment location of $O_{i,3}$ is closer to the Upstream task (in the same worker). This near-source processing can significantly reduce the transmission latency of data tuples.

Definition 2. *Off-source data processing.* If two communicating tasks are deployed to different different nodes, data tuples transmitted between the tasks are considered to be processed off-source.

For example, in Fig. (6), the data grouping between the Upstream task and tasks $O_{i,4}$ and $O_{i,5}$ is off-source. This off-source processing can significantly increase the transmission latency due to cross-node communication.

V. Problem Statement

Drawing from the above models, we formalize the scheduling problems for distributed stream computing, which involve DAG initialization scheduling and data stream grouping.

A. DAG initialization scheduling

When deploying a streaming application DAG across M compute nodes in a cluster, we lack the access to the communication data within the application during the initial scheduling phase. However, we derive data dependency relationships from the DAG, and treat tasks with operator-level dependencies as having potential communication demands. Therefore, we

initialize each element in the communication cost matrix E, which can be calculated by Eq. (7).

$$\forall e_{i,k} \in E, e_{i,k} = \begin{cases} 1, \text{if communication exists between} \\ \quad i\text{-th and } k\text{-th tasks in } T, \\ 0, \text{otherwise.} \end{cases} \tag{7}$$

Let matrix Q be the task deployment solution. q_m is the task deployment vector of compute node m with length s . Q can be described as Eq. (8).

$$Q = (q_1, q_2, ..., q_m, ..., q_M), \tag{8}$$

subject to

$$\begin{cases} \sum_{m=1}^{M} q_{m,i} = 1, i = 1, 2, ..., s, \\ \sum_{i=1}^{s} q_{m,i} \approx \sum_{i=1}^{s} q_{m,j}, i \neq j, m = 1, 2, ..., M. \end{cases} \tag{9}$$

In Eq. (9), $q_{m,i}$ denotes the i-th element (task) in vector q_m. The first constraint ensures that each task (i) can only be deployed on one compute node. The second constraint ensures that the number of tasks deployed on each node should be balanced as much as possible.

Based on the above descriptions, our objective function Z for initial task deployment aims to minimize data dependencies between nodes, i.e., the total communication cost minus the intra-node communication cost, as represented by Eq. (10).

$$Z = \min(\sum_{i=1}^{s}\sum_{k=1}^{s} e_{i,k} - Q^{\mathrm{T}} \cdot E \cdot Q). \tag{10}$$

B. Data stream grouping

Given an upstream operator o_i and its direct downstream operator o_j, data tuples $\{dt_1, dt_2, ...\}$ from data stream ds can be redirected from operator o_i to the k tasks $\{o_{j,1}, o_{j,2}, ..., o_{j,k}\}$ of operator o_j using grouping function GF. The grouping function is a mapping: $GF(ds) = \{dt_1, dt_2, ...\} \rightarrow \{o_{j,1}, o_{j,2}, ..., o_{j,k}\}$.

According to **Theorem 1**, when the input rate is stable, maximizing intra-node communication minimizes inter-node communication. Therefore, data streams should be redirected to the near-source tasks as much as possible. This means that during the initial scheduling phase, tasks with potential communication demands are deployed on the same node.

At runtime, the communication load between tasks can be collected through a monitoring module. This communication load information is constructed into a matrix E. Then, our objective function J for data stream grouping is to maximize intra-node communication (i.e., processing data tuples locally) at runtime, which can be generalized as Eq. (11).

$$J = \max(Q^{\mathrm{T}} \cdot E \cdot Q). \tag{11}$$

VI. Ns-Stream: Architecture and Algorithms

Based on the above discussion, we present Ns-Stream, a lightweight scheduler specifically designed for efficient grouping of data streams. In this section, we first introduce Ns-Stream's architecture, followed by the algorithms for DAG initialization scheduling and near-source grouping.

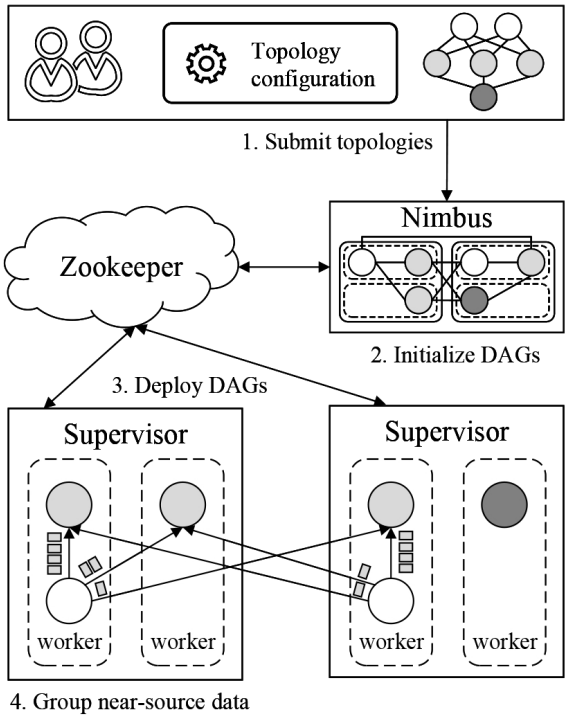


Fig. 7. Ns-Stream architecture.

A. System architecture

Ns-Stream is implemented on the Storm platform. Once a streaming application is submitted, the platform initializes the application's logical topology and deploys it onto compute nodes in the cluster. The topology DAG, deployed across compute nodes, continues running unless manually terminated. Two steps are involved in implementing Ns-Stream: (1) During the topology initialization phase, we strategically deploy tasks with potential communication demands to the same compute node by implementing the `IScheduler` interface in Storm. (2) At runtime, our hierarchical near-source grouping strategy steps in to optimize system latency by routing data tuples to tasks deployed on the same worker or node. This specific grouping strategy is implemented by the `CustomStreamGrouping` interface.

As shown in Fig. 7, a logical topology is first constructed to define internal logic and data dependencies within the streaming application. During the topology construction process, users need to specifically define the Spout components (generating data streams), configure the Bolt components (processing the data streams), establish the connection relationships between these components, and determine the data transmission methods to ensure that the entire topology runs efficiently according to the expected logic. The constructed topologies are submitted to Nimbus by users.

Nimbus deploys the topologies (DAGs) to the compute nodes within the cluster. Ns-Stream prioritizes deploying tasks with data dependencies in one topology on the same compute node. Furthermore, the allocation of these tasks on the same compute node can be fine-grained to minimize the data communication between workers on the same compute node,

enhancing the overall efficiency and performance. The method for task deployment is detailed in Section VI-B.

The Supervisors retrieve task deployment information from Zookeeper and initiate the corresponding Worker processes on the local nodes. The grouping functions for each DAG running on Supervisors are dynamically adjusted to minimize inter-node communication overhead. Subject to resource constraints, data tuples are routed to tasks for processing with the following priority: first to tasks within the same worker, second to tasks on the same compute node, and last to tasks on other nodes. Extensive local processing of data tuples therefore reduces the communication latency between workers and compute nodes. The method for near-source grouping is detailed in Section VI-C.

B. DAG initialization scheduling

At the initial stage, tasks of a DAG are deployed to workers on nodes in the cluster. If task deployment remains unoptimized during this DAG initialization process, communicating tasks might have to be migrated to the same worker or node at runtime, which can be costly. An unoptimized deployment often occurs when the default task deployment uses a round-robin strategy, which tends to place communicating tasks on different nodes. The more tasks that require migration during runtime scheduling, the higher the associated cost. Therefore, during this initialization process, we use graph partitioning to deploy communicating tasks to the same worker or node, eliminating the need for costly runtime scheduling later on.

Deep learning techniques have been applied to address graph partitioning problems [29], [30]. Our proposed solution is implemented using the Generalized Approximate Partitioning (GAP) framework [31]. This GAP framework consists of graph encoding modules and feed-forward neural networks. By iteratively feeding topological structure information into the GAP model and employing the steepest descent method from nonlinear programming, the graph partitioning model's parameters are adjusted along the negative gradient direction of the loss function. Our loss function Lf for graph partitioning is determined by the inter-node communication cost and the balance of the number of tasks deployed on each node. It can be described as (12).

$$\begin{aligned} Lf = & \sum_{reduce-sum} (Y \oslash \Gamma)(1-Y) \odot E \\ & + \sum_{reduce-sum} \left(\mathbf{1}^{\mathrm{T}} Y - \frac{s}{M}\right)^2, \end{aligned} \tag{12}$$

where Y denotes the probability for each task being deployed to one compute node from set $\{cn_1, cn_2, ..., cn_M\}$. M and s denote the number of compute nodes in the cluster and the number of tasks of the streaming application, respectively. $\mathbf{1}$ is an $s \times 1$ column vector, with each element being "1". Operator $\oslash$ represents the elements of matrix Y divided by the corresponding elements of matrix Γ. Operator $\odot$ represents the Hadamard product. The first term of Eq. (12) represents the communication cost between nodes. The second term represents the similarity in the number of tasks deployed on each node. Γ can be calculated by Eq. (13).

$$\Gamma = Y^{\mathrm{T}} X, \tag{13}$$

where X is the vector that represents the number of edges connected to each task.

We use graph convolutional networks to learn the topological information of streaming applications, and then input the learned information into fully connected networks. Our graph partitioning algorithm is described in Algorithm 1.

Algorithm 1: Subgraph partitioning algorithm.

Input: communication cost matrix E, number of subgraphs m;
Output: probability matrix $Y_{s\times m}$;
1 Initialize the user-defined maximum iteration count as $iter\_count$;
2 **while** $iter\_count > 0$ **do**
3 Initialize the graph encoding depth as ed;
4 **for** $l = 1$ *to* ed **do**
5 **for** *each* $e_{i,k}$ *in* E **do**
6 Get the neighboring task set ϑ of edge $e_{i,k}$;
7 Get the edge set ε_i of each task ϑ_i in ϑ ;
8 Calculate the embedding e of edge $e_{i,k}$ based on ε_i information ;
  /* Aggregate edge information. */
9 $e_{i,k} = \sum_{e \in \varepsilon_i} e$;
10 **end**
11 **end**
  /* Feed the encoded matrix E into a neural network consisting of 3 fully-connected layers. */
12 $E^{(2)} \leftarrow$Linear(Linear(Linear(E)));
13 $Y_{s\times m} \leftarrow$softmax($E^{(2)}$) ;
14 Calculate the loss function by Eq. (12) and backpropagate the gradient;
15 $iter\_count = iter\_count - 1$;
16 **end**
17 **return** $Y_{s\times m}$

The input of Algorithm 1 includes the communication cost matrix E between tasks and the number of desired subgraphs m. The output is a probability matrix $Y_{s\times m}$, where its element $Y_{i,j}$ represents the probability of the i-th task belonging to the j-th subgraph. Step 1 and Step 3 initialize the parameters required for the algorithm to run. Steps 4 through 11 encode the graph structure via Graph Convolutional Networks (GCNs) [32]. Steps 12 and 13 compute the probability matrix $Y_{s\times m}$ for the encoded graph. Step 14 updates the model parameters based on the loss function (Eq.(12)). The time complexity of Algorithm 1 is $O(iter\_count \cdot s^2 \cdot ed)$, where s is the number of tasks in the streaming application.

During the initialization phase, the initial task deployment can be fine-grained by following two key steps: (1) the topological information of DAG is fed into Algorithm 1 to

obtain multiple subgraphs. Tasks within each subgraph are deployed to the same node. (2) To minimize intra-node communication, each subgraph is further processed by Algorithm 1 to determine which tasks can be deployed to the same worker on the node.

C. Near-Source grouping

As discussed previously, we have observed that communication overheads between nodes and between workers within nodes can impact system performance. To minimize these overheads, we propose a hierarchical, near-source data grouping strategy to optimize system latency. This strategy prioritizes the deployment of data tuples to tasks deployed on the same worker and compute node. Each near-source task processes these data tuples based on its processing capability PC.

Definition 3. *Task processing capability.* A task's processing capability PC assesses whether the task's processing capability has reached saturation.

PC has a value range of [0, 1]. If its value is closer to 1, it indicates that the task is continuously invoking the execute() function, and its processing capacity is nearly saturated. If its value is closer to 0, it indicates that the task is completely idle and has no data tuples being processed.

A task's PC can be calculated by Eq. (14).

$$PC = \frac{\sum_{i=1}^{\rho_k} et_i}{t_u - t_d}, \tag{14}$$

where t_d and t_u respectively denote the start and end time of a time interval $[t_d, t_u]$. ρ_k denotes the total number of tuples processed by the k-th task during this time interval. et_i denotes the processing time of the tuple dt_i.

In addition, to ensure efficient utilization of worker resources, when the workload of near-source tasks becomes excessive and their processing capacities are approaching saturation, excess data from the data source is redirected to off-source tasks. This redistribution relieves the computational pressure on local resources. We define the threshold for task processing capacity as α. If α is too high, it may overload near-source tasks. Conversely, if α is set too low, it may increase communication cost between nodes. The setting of α value is discussed in Section VII-C.

To calculate the tuple distribution probability, we assume there exists an upstream operator task $o_{i-1,1}$ that emits data tuples to λ tasks $\{o_{i,1}, ..., o_{i,\lambda}\}$ of downstream operator o_i. To demonstrate the near- and off-source processing, we simulate different grouping scenarios by splitting the λ tasks into 3 task sets: $\{o_{i,1}, ..., o_{i,p_1}\}$, $\{o_{i,p_1+1}, ..., o_{i,p_1+p_2}\}$, and $\{o_{i,p_1+p_2+1}, ..., o_{i,\lambda}\}$. These sets are deployed with the upstream source task $o_{i-1,1}$ to the same worker, different workers on the same node, and different nodes, respectively. We define the tuple output rate of task $o_{i-1,1}$ as ro, and the input rates of tasks $\{o_{i,1}, o_{i,2}, ..., o_{i,\lambda}\}$ as $\{io_1, io_2, ..., io_\lambda\}$, respectively.

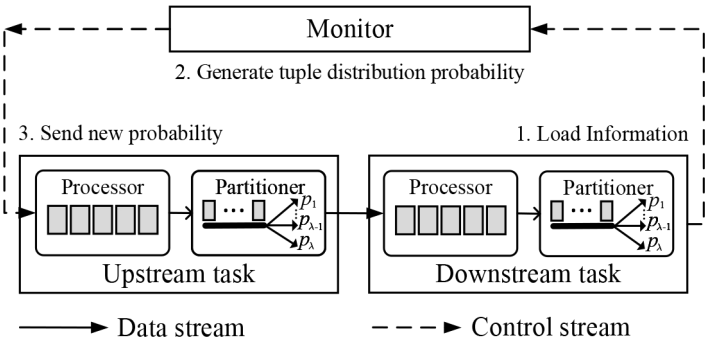


Fig. 8. Near-source grouping workflow.

If a task's PC does not exceed threshold α, the maximum average processing rate $\bar{\rho}_k$ of task $o_{i,k}$, $k \in \{1, ..., \lambda\}$ can be calculated by Eq. (15).

$$\bar{\rho}_k = \frac{\alpha \cdot (t_u - t_d) \cdot \rho_k}{\sum_{i=1}^{\rho_k} et_i}. \tag{15}$$

Based on the above information, the tuple distribution weights $\{w_1, w_2, ..., w_k, ..., w_\lambda\}$ for the data tuples emitted by task $o_{i-1,1}$ to tasks $\{o_{i,1}, o_{i,2}, ..., o_{i,\lambda}\}$ can be calculated by Eq.(16).

In Eq.(16), the first condition indicates that near-source tasks in the same worker can fully process all tuples emitted by the upstream task. The second condition indicates that that near-source tasks in different workers on the same node can process the tuples remaining from near-source tasks in the same worker. The third condition indicates that off-source tasks can process the tuples remaining from the near-source tasks.

The tuple distribution probability $P = \{p_{i,1}, p_{i,2}, ..., p_{i,k}, ..., p_{i,\lambda}\}$ for the data tuples emitted by task $o_{i-1,1}$ to tasks $\{o_{i,1}, o_{i,2}, ...,o_{i,\lambda}\}$ can be calculated based on the distribution weights of these tuples. Specifically, the probability $p_{i,k}$ (for emitting tuples to task $o_{i,k}$) can be calculated by Eq. (17). Then, the probability range for emitting tuples to task $o_{i,k}$ is $[p_{i,k-1}, p_{i,k}]$. A wider probability range means more tuples can be distributed to the corresponding task, while a narrower range results in fewer tuples being distributed.

$$p_{i,k} = \sum_{j=1}^{k} w_j \tag{17}$$

As shown in Fig. 8, the monitor component collects load data from downstream tasks and calculates their maximum processing rates using Eq. (15). Based on task deployment locations, it computes tuple distribution weights and updates the probabilities P using Eq. (17). These new probabilities P are sent to upstream tasks to adjust their partitioners. A higher probability $p_{i,k}$ means a downstream task $o_{i,k}$ receives more tuples. When grouping each data tuple, the partitioner inputs the updated distribution probabilities into Algorithm 2.

The input of Algorithm 2 includes the distribution probability $\{p_{i,1}, p_{i,2}, ..., p_{i,\lambda}\}$ for data tuples emitted by task $o_{i-1,1}$ to tasks $\{o_{i,1}, o_{i,2}, ..., o_{i,\lambda}\}$. The output is the target task to which

$$w_k = \begin{cases} \dfrac{\overline{\rho}_k}{ro}, \text{ if } k \in [1, p_1], \sum_{j=1}^{p_1} \overline{\rho}_j \geqslant ro, \\ \left(1 - \sum_{j=1}^{p_1} \dfrac{\overline{\rho}_j}{ro}\right) \cdot \dfrac{\max\left(0, \overline{\rho}_k - io_k\right)}{ro}, \text{ if } k \in [p_1+1, p_2], \sum_{j=1}^{p_1} \overline{\rho}_j < ro, \\ \left(1 - \dfrac{\sum_{j=1}^{p_1} \overline{\rho}_j + \sum_{g=p_1+1}^{p_2} \max\left(0, \overline{\rho}_g - io_g\right)}{ro}\right) \cdot \dfrac{\max\left(0, \overline{\rho}_k - io_k\right)}{ro}, \text{ if } k \in [p_2+1, p_3], \sum_{j=1}^{p_1} \overline{\rho}_j + \sum_{g=p_1+1}^{p_2} \max\left(0, \overline{\rho}_g - io_g\right) < ro, \\ 0, otherwise. \end{cases} \tag{16}$$

Algorithm 2: Data stream grouping.

Input: distribution probability $P = \{p_{i,1}, p_{i,2}, ..., p_{i,\lambda}\}$;
Output: $TargetTask$

```
Get the target task set {o_{i,1}, o_{i,2}, ..., o_{i,λ}} ;
Generate a random number r, r ∈ [0, 1] ;
for k = 1 to λ do
    if r < p_{i,k} then
        TargetTask ← o_{i,k};
        /* Found the target task to
           receive and process the
           tuple. Quit.                */
        Break;
    end
end
return TargetTask
```

the data tuples will be distributed. Step 1 gets the downstream operator tasks that have data dependencies with the emitting data source $o_{i-1,1}$. Step 2 generates a random number for selecting a target task based on its distribution probabilities. Steps 3 to 8 determine which task's probability range that the random number falls into, and emit the data tuple to that task for processing. The time complexity of Algorithm 2 is $O(\lambda)$, where λ is the number of tasks in operator o_i.

VII. Performance Evaluation

The experimental cluster consists of 16 machines: 3 machines are designated as master nodes running Nimbus, while the remaining 13 machines serve as worker nodes hosting Supervisor processes. Furthermore, a Zookeeper cluster is deployed across 3 machines, which are multiplexed with the Nimbus master nodes. Each Nimbus node is equipped with a GAP model, which consists of two graph encoding modules and a feedforward neural network. Each graph encoder uses a hidden layer size of 64.

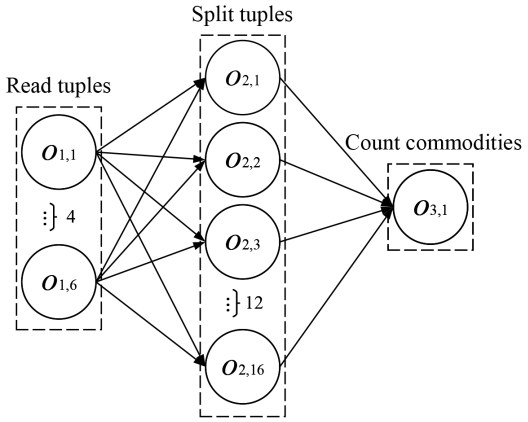
We use the public dataset [33] from Alibaba to simulate a real-world workload for evaluating the performance of the proposed stream computing system. The real-world dataset encompasses the activities of approximately one million random Taobao users over the period from November 25th to December 3rd, 2017. These activities include clicks, purchases, additions to cart, and likes. Each row in the dataset represents a user activity, consisting of a user ID, product ID, product category ID, activity type, and timestamp. This dataset's time-driven, event-intensive, and behaviorally diverse characteristics make it well-suited to benchmark stream computing systems like Ns-Stream. To align the dataset with the data source of the streaming application, we partition the dataset into equally sized subsets by timestamps. The number of sub-datasets corresponds to the number of tasks in the Read operator, and each task only reads its assigned subset.

In addition, we use COMMCount, a widely used benchmark application, as the test streaming application. This COMMCount application creates a more complex scenario by counting the number of commodities browsed by users. Two COMMCount DAGs are designed to simulate different loads between operators and tasks: Topology 1 has fewer tasks (6) for data emitting (Read tuples) and more tasks (16) for data processing (Split tuples); Topology 2 has more tasks (8) for data emitting and fewer tasks (8) for data processing. Their logic graphs are shown in Fig. 9. The two topologies reflect typical structural patterns in real-world applications, such as lightweight upstream vs. heavyweight downstream, and balanced processing pipelines.

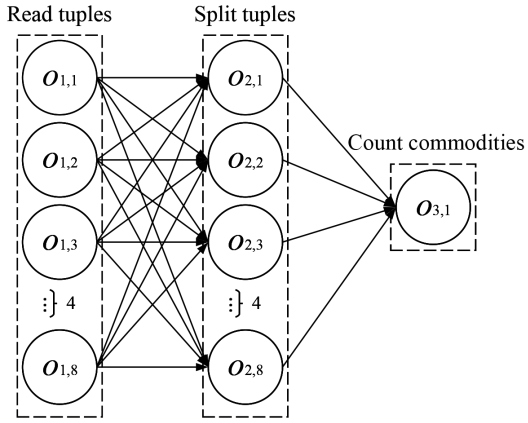
We compare the performance of Ns-Stream with state-of-the-art (SOTA) methods, including the R-Storm [34], TOP-Storm [35], SP-ant [6], and the commonly used Shuffle grouping (SG). Among these works, R-Storm, TOP-Storm and Sp-ant are the most representative in communications awareness and resource management. We collect system latency and bottlenecks through the built-in ACK mechanism in the Storm platform.

A. System latency

System latency is a key performance metric in stream processing that directly impacts the user experience of real-time applications. We define system latency as the time taken for a data stream to enter the topology, be fully processed, and produce the final output. We evaluate system latency under stable, increasing, and fluctuating input rates. The fluctuating stream simulates real-time dynamics, following the approach



(a) Instance topology 1 of COMMCount.



(b) Instance topology 2 of COMMCount

Fig. 9. Two instance topologies of COMMCount.

in RIoTBench [36], where the input load is emulated based on user activity patterns on the platform. For instance, the number of orders on the Taobao platform typically peaks around midday and in the evening, while it declines during standard working hours.

Given a **stable input** rate of 2,000 tuples/s, Ns-Stream significantly reduces the system latency across the test streaming applications compared to the SOTA methods.

As shown in Fig. 10, for instance topology 1, the average system latencies are 25.4 ms, 23.3 ms, 20.4 ms, 19.6 ms, and 15.5 ms for SG, R-Storm, TOP-Storm, SP-ant and Ns-Stream, respectively, when the system stabilizes. Similarly, as shown in Fig. 11, for instance topology 2, the average system latencies are 22.3 ms, 19.5 ms, 15.6 ms, 16.2 ms, and 13.9 ms for SG, R-Storm, TOP-Storm, SP-ant and Ns-Stream, respectively. Compared to the SOTA methods, Ns-Stream reduces the maximum system latency by 38.9% and the minimum by 10.8%.

Given an **increasing rate** and an increment of 1,000 tuples/s, Ns-Stream also significantly reduces the system latency across the test streaming applications compared to the SOTA methods.

As shown in Fig. 12, Ns-Stream consistently shows the lowest latency across all the input rates, followed by SG, R-Storm, TOP-Storm, and finally SP-ant for instance topology 1. This indicates that Ns-Stream achieves higher efficiency and lower latency when processing increasing data streams. As the input rate increases from 1000 tuples/s to 6000 tuples/s, Ns-Stream's latency increases by approximately 49% (from 16.2 ms to 24.2 ms), SG's latency increases by approximately 76% (from 20. 6 ms to 36.4 ms), R-Storm's latency increases by approximately 65% (from 19.6 ms to 32.4 ms), TOP-Storm's latency increases by approximately 63% (from 18.1 ms to 29.5 ms), and SP-ant's latency increases by approximately 52% (from 18.7 ms to 28.5 ms). This suggests that Ns-Stream is relatively more stable in terms of latency growth.

Similarly, Ns-Stream exhibits the lowest latency across all the data stream rates for instance topology 2. As shown in Fig. 13, from an input rate of 1000 tuples/s to 6000 tuples/s, Ns-Stream's latency increases by approximately 56%, SG's by approximately 85%, R-Storm's by approximately 74%, TOP-Storm's by approximately 74%, and SP-ant's by approximately 70%. Compared to the results for topology 1, Ns-Stream's latency growth is more rapid in topology 2.

Given a **fluctuating input** rate (peak: 7697 tuples/s), Ns-Stream also significantly improves system stability and reduces latency across the test streaming applications compared to the SOTA methods.

As shown in Figs. 14 and 15, system latency exhibits significant fluctuations under real input load conditions. SG and R-Storm show noticeable latency spikes, with peaks exceeding 25 ms and frequent oscillations, indicating high sensitivity to changes in data rate. SP-ant and TOP-Storm show moderate variability, performing better than SG and R-Storm, but still exhibiting clear latency fluctuations. In contrast, Ns-Stream consistently maintains low latency with minimal variation, staying around 11 ms even under substantial changes in input rate. This stability highlights Ns-Stream's robustness to input variation.

Experiments on both topologies indicate that Ns-Stream achieves lower latency and maintains relatively stable performance compared to the SOTA methods. This advantage can be attributed to Ns-Stream's dynamic data adjustment to near-source tasks. Although SP-ant and TOP-Storm also optimize system latency to some extent, its approach of minimizing inter-node communication through task scheduling is less effective in optimizing overall system performance due to the stable communication load between tasks.

B. System bottleneck

A system bottleneck occurs when the system's data processing rate reaches its peak given the topology's configuration and available resources. In our tests, we gradually increase the data input rate until it causes downtime in the tasks. This approach allows us to identify the input rate at which the system's performance degrades significantly, leading to failures, and thus pinpoint the bottleneck.

Given an **increasing rate** and an increment of 500 tuples/s, Ns-Stream exhibits significant improvements in system bottleneck compared to the SOTA methods across the test streaming applications.

As shown in Fig. 16, for instance topology 1, the system bottlenecks are 12,500 tuples/s, 8,500 tuples/s, 9,000 tuples/s,

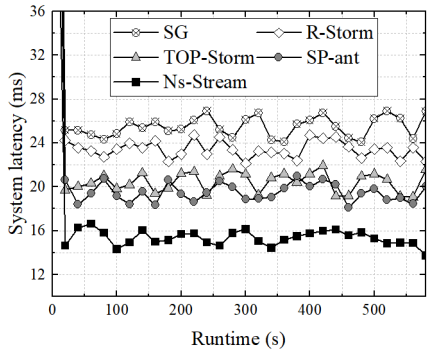


Fig. 10. System latency of topology 1 under stable data rate of 2000 tuples/s.

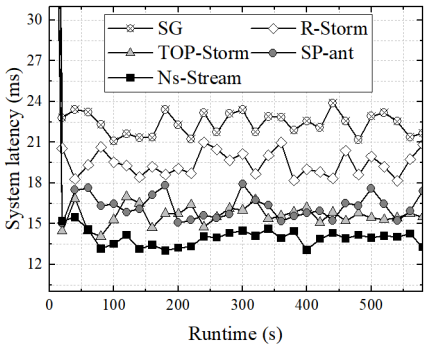


Fig. 11. System latency of topology 2 under stable data rate of 2000 tuples/s.

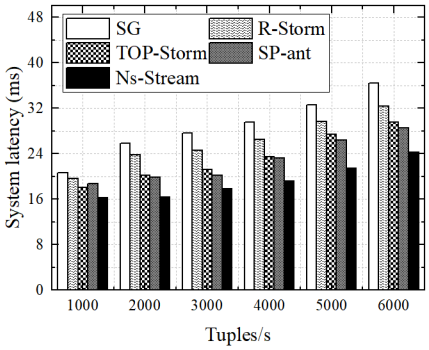


Fig. 12. System latency of topology 1 under increasing data rates.

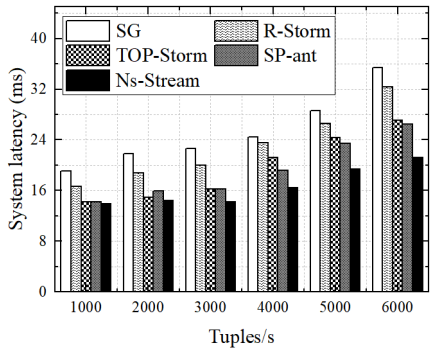


Fig. 13. System latency of topology 2 under increasing data rate.

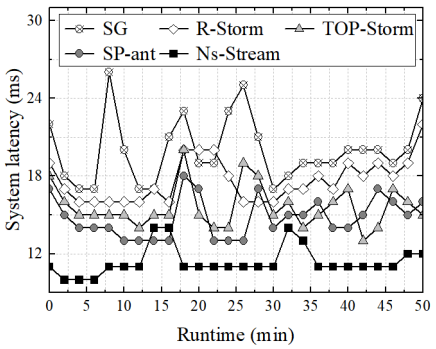


Fig. 14. System latency of topology 1 under fluctuating data rates.

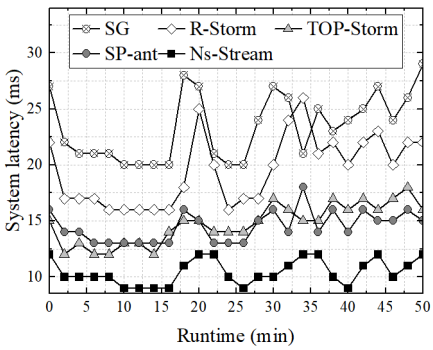


Fig. 15. System latency of topology 2 under fluctuating data rates.

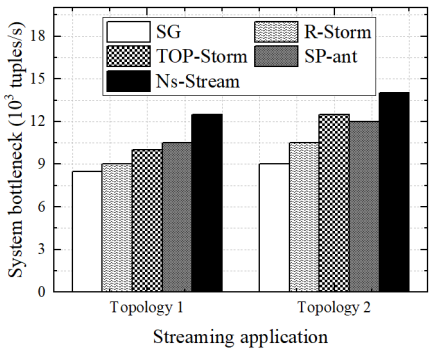


Fig. 16. System bottleneck of topologies 1 and 2.

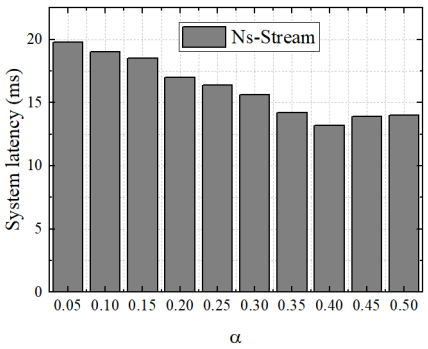


Fig. 17. System latency of topology 1 with different α.

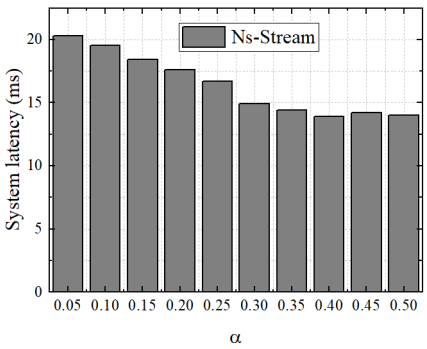


Fig. 18. System latency of topology 2 with different α.

10,000 tuples/s, and 10,500 tuples/s for Ns-Stream, SG, R-Storm, TOP-Storm, and SP-ant, respectively, when the system stabilizes. Similarly, for instance topology 2, the system bottlenecks are 14,000 tuples/s for Ns-Stream, 9,000 tuples/s for SG, 10,500 tuples/s for R-Storm, 12,500 tuples/s for TOP-Storm, and 12,000 tuples/s for SP-ant. Compared to the most advanced SP-ant, Ns-Stream enhances the average system bottleneck by 15.1%. It is evident that the average bottleneck of Ns-Stream surpasses those of SG and SP-ant when the input rate increases steadily.

Ns-Stream has a higher system bottleneck compared to the other two SOTA methods. This is because, when the resource load of a task is insufficient, Ns-Stream can dispatch some data to other tasks for processing, thereby enhancing the system's resource utilization efficiency.

C. System parameter settings

Proper system parameter settings enable streaming applications to process data at their best, which is crucial for enhancing the performance of distributed stream computing systems. In Ns-Stream, threshold α is important as it determines the prioritization of near-source or off-source tasks for data processing. We conduct experiments to evaluate its impact on system performance (both latency and bottleneck) by setting different α values.

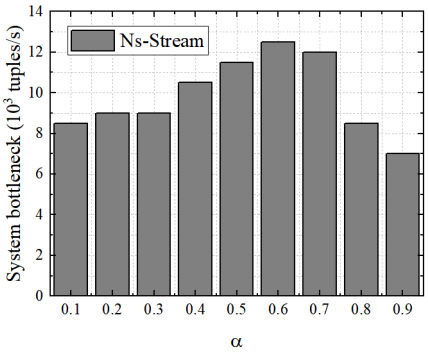


Fig. 19. System bottleneck of topology 1 with different α.

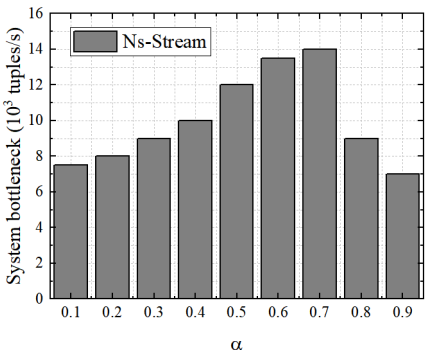


Fig. 20. System bottleneck of topology 2 with different α.

Given a stable input rate of 3,000 tuples/s, we compare the **system latency** under different α values for the two topologies.

As shown in Fig. 17, for instance topology 1, the system latency gradually decreases as the value of α increases up to 0.35. However, beyond 0.35, the system latency stabilizes. Similarly, in Fig. 18, for instance topology 2, the system latency steadily decreases with the increase of α up to 0.30. Beyond this point, the latency remains consistent.

A smaller value of α can affect the system latency by prioritizing off-source tasks for data processing, leading to increased communication overhead between nodes. However, when α reaches a certain threshold, near-source tasks become fully capable of handling the data tuples emitted by upstream tasks, thereby maintaining a stable latency.

Given an increasing input rate and an increment of 500 tuples/s, we compare the **system bottleneck** under different α values.

As shown in Fig. 19, for instance topology 1, the system bottleneck gradually increases as the value of α approaches 0.6. However, beyond 0.6, the system bottleneck begins to gradually decrease. Similarly, in Fig. 20, for instance topology 2, the system bottleneck progressively rises with the increase of α up to 0.7. Beyond that point, the system bottleneck starts to gradually decline.

The parameter α governs the trade-off between near-source and off-source task processing. A very low α can result in excessive inter-node communication, while a very high α may risk overloading local resources. As shown in Figs. 19 and 20, setting α within the range of 0.5 to 0.7 yields favorable bottleneck performance by balancing data transmission overhead and local resource utilization. This trade-off is generally applicable across a broad range of DAGs, as it supports both local efficiency and overall system flexibility.

VIII. Conclusions and Future Work

In this study, we observe that when resources are abundant, communication overhead between compute nodes is the primary factor affecting system latency. However, in resource-constrained scenarios, the computational demands of tasks become the critical factor limiting system performance. To overcome these limitations, we introduce Ns-Stream, a data tuple scheduler designed to dynamically adjust weight assignments between near-source and off-source tasks. Ns-Stream prioritizes local processing of data tuples based on the computing capabilities of near-source tasks, aiming to optimize resource utilization and reduce data transmission overhead. To achieve this, a graph convolutional network is employed to deploy tasks with potential communication to the same compute node in advance during the initialization scheduling. Ns-Stream has been implemented on the Apache Storm platform. Experimental results demonstrate its advantage over existing solutions, exhibiting notable improvements in both system throughput and latency.

In our future work, we aim to integrate an auto-scaling operator parallelism mechanism into Ns-Stream. This integration will enable Ns-Stream to dynamically adjust the number of tasks in operators, further boosting the overall performance of systems.

References

[1] Z. Wen, R. Yang, B. Qian, Y. Xuan, L. Lu, Z. Wang, H. Peng, J. Xu, A. Y. Zomaya, and R. Ranjan, "Janus: Latency-aware traffic scheduling for iot data streaming in edge environments," *IEEE Transactions on Services Computing*, vol. 16, no. 6, pp. 4302–4316, 2023.

[2] M. Barika, S. Garg, A. Chan, and R. N. Calheiros, "Scheduling algorithms for efficient execution of stream workflow applications in multicloud environments," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 860–875, 2022.

[3] H. Ji, S. Jiang, Y. Zhao, G. Wu, G. Wang, and G. Y. Yuan, "Bs-join: A novel and efficient mixed batch-stream join method for spatiotemporal data management in flink," *Future Generation Computer Systems*, vol. 141, pp. 67–80, 2023.

[4] A. Brown, S. Garg, J. Montgomery, and U. KC, "Resource scheduling and provisioning for processing of dynamic stream workflows under latency constraints," *Future Generation Computer Systems*, vol. 131, pp. 166–182, 2022.

[5] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, "Efficient operator placement for distributed data stream processing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1753–1767, 2019.

[6] M. Farrokh, H. Hadian, M. Sharifi, and A. Jafari, "Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters," *Expert Systems with Applications*, vol. 191, pp. 1–11, 2022.

[7] J. Rho, T. Azumi, M. Nakagawa, K. Sato, and N. Nishio, "Scheduling parallel and distributed processing for automotive data stream management system," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 286–300, 2017.

[8] L. Eskandari, Z. Huang, and D. Eyers, "P-scheduler: adaptive hierarchical scheduling in apache storm," in *Proceedings of the Australasian Computer Science Week Multiconference*, 2016, pp. 1–10.

[9] H. Li, J. Xia, W. Luo, and H. Fang, "Cost-efficient scheduling of streaming applications in apache flink on cloud," *IEEE Transactions on Big Data*, vol. 9, no. 4, pp. 1086–1101, 2023.

[10] M. Asif and M. Aleem, "Ban-storm: A bandwidth-aware scheduling mechanism for stream jobs," *Journal of Grid Computing*, vol. 19, no. 3, pp. 1–16, 2021.
[11] A. Al-Sinayyid and M. Zhu, "Job scheduler for streaming applications in heterogeneous distributed processing systems," *The Journal of Super computing*, vol. 76, pp. 9609–9628, 2020.
[12] X. Huang, Z. Shao, and Y. Yang, "Potus: Predictive online tuple scheduling for data stream processing systems," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2863–2875, 2022.
[13] J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou, and X. Zhou, "Distributed stream rebalance for stateful operator under workload variance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2223–2240, 2018.
[14] W. Li, D. Liu, K. Chen, K. Li, and H. Qi, "Hone: Mitigating stragglers in distributed stream processing with tuple scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2021–2034, 2021.
[15] S. Ding, L. Yang, J. Cao, W. Cai, M. Tan, and Z. Wang., "Partitioning stateful data stream applications in dynamic edge cloud environments," *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2368–2381, 2022.
[16] H. Chen, F. Zhang, and H. Jin, "Pstream: A popularity-aware differentiated distributed stream processing system," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1582–1597, 2021.
[17] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2018, pp. 741–756.
[18] H. Li, H. Fang, H. Dai, T. Zhou, W. Shi, J. Wang, and C. Xu, "A cost-efficient scheduling algorithm for streaming processing applications on cloud," *Cluster Computing*, vol. 25, pp. 781–803, 2022.
[19] J. Tan, Z. Tang, W. Cai, W. J. Tan, X. Xiao, J. Zhang, Y. Gao, and K. Li, "A cost-aware operator migration approach for distributed stream processing system," *IEEE Transactions on Cloud Computing*, vol. 13, no. 1, pp. 441–454, 2025.
[20] A. C. Z. Gang Liu, Zeting Wang and R. Mao, "Adaptive key partitioning in distributed stream processing," *CCF Transactions on High Performance Computing*, vol. 6, no. 3, p. 164–178, 2023.
[21] X. Liu, Y. Lin, and R. Buyya, "Dynamic resource-efficient scheduling in data stream management systems deployed on computing clouds," in *New Frontiers in Cloud Computing and Internet of Things*. Switzerland: Springer, October 2022, pp. 133–163.
[22] M. Wu, D. Sun, S. Gao, and R. Buyya, "Straggler mitigation via hierarchical scheduling in elastic stream computing systems," *Future Generation Computer Systems*, vol. 166, pp. 1–15, 2025.
[23] E. Zapridou, I. Mytilinis, and A. Ailamaki, "Dalton: Learned partitioning for distributed data streams," *Proceedings of the VLDB Endowment*, vol. 16, no. 3, p. 491–504, 2022.
[24] "Apache storm," 2021. [Online]. Available: https://storm.apache.org/2021/10/11/storm124-released.html
[25] Q. Wang, D. Zuo, Z. Zhang, S. Chen, and T. Liu, "An adaptive non-migrating load-balanced distributed stream window join system," *The Journal of Supercomputing*, vol. 79, pp. 8236–8264, 2023.
[26] S. Zhou, F. Zhang, H. Chen, H. Jin, and B. B. Zhou, "Fastjoin: A skewness-aware distributed stream join system," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 1042–1052.
[27] H. Xu, P. Liu, S. T. Ahmed, D. Da Silva, and L. Hu, "Adaptive fragment-based parallel state recovery for stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2464–2478, 2023.
[28] Y. He, W. Wu, Y. Lei, M. Liu, and C. Lao, "A generic service to provide in-network aggregation for key-value streams," in *ASPLOS 2023: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 2, 2023, pp. 33–47.
[29] Y. Yang and D. Li, "Nenn: Incorporate node and edge features in graph neural networks," in *Proceedings of Machine Learning Research*, vol. 129, 2020, pp. 593–608.
[30] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *34th International Conference on Machine Learning, ICML 2017*, vol. 70, 2017, pp. 3748–3757.
[31] A. Nazi, W. Hang, A. Goldie, S. Ravi, and A. Mirhoseini, "Gap: Generalizable approximate graph partitioning framework," *arXiv*, 2019. [Online]. Available: https://arxiv.org/abs/1903.00614
[32] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR*, 2017, pp. 1–14.
[33] Aliyun, "tianchi," https://tianchi.aliyun.com/dataset/.
[34] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, 2015, p. 149–161.
[35] A. Muhammad, M. Aleem, and M. A. Islam, "Top-storm: A topology-based resource-aware scheduler for stream processing engine," *Cluster Computing*, vol. 24, p. 417–431, 2021.
[36] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, pp. 1–22, 2017.

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

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

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

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