Ls-Stream: Lightening Stragglers in Join Operators for Skewed Data Stream Processing

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

***Abstract*—Load imbalance can lead to the emergence of stragglers, i.e., join instances that significantly lag behind others in processing data streams. Currently, state-of-the-art solutions are capable of balancing the load between join instances to mitigate stragglers by managing hot keys and random partitioning. However, these solutions rely on either complicated routing strategies or resource-inefficient processing structures, making them susceptible to frequent changes in load between instances. Therefore, we present Ls-Stream, a data stream scheduler that aims to support dynamic workload assignment for join instances to lighten stragglers. This paper outlines our solution from the following aspects: (1) The models for partitioning, communication, matrix, and resource are developed, formalizing problems like imbalanced load between join instances and state migration costs. (2) Ls-Stream employs a two-level routing strategy for workload allocation by combining hash-based and key-based data partitioning, specifying the destination join instances for data tuples. (3) Ls-Stream also constructs a fine-grained model for minimizing the state migration cost. This allows us to make trade-offs between data transfer overhead and migration benefits. (4) Experimental results demonstrate significant improvements made by Ls-Stream: reducing maximum system latency by 49.3% and increasing maximum throughput by more than 2x compared to existing state-of-the-art works.**

***Index Terms*—Distributed stream computing, stream join, straggler instances, load balancing, state migration.**

I. Introduction

STREAM join is one of the most critical and resource-intensive operators in stream processing systems [1]. It finds widespread use in various domains, including finance, e-commerce, transportation, and healthcare [2], [3], [4]. In comparison to traditional database join operations, stream join is more challenging due to the continuous, high-speed, and real-time features of data streams [5]. Stream joins must merge data from two sources for complex data analysis, placing a significant demand on system resources [6]. Therefore, achieving efficient stream joins is pivotal for enhancing system performance [7].

Efficient stream joins must meet the following fundamental requirements: 1) Real-time: The system must quickly respond and reflect the data's value in a short period of time; 2) Resource efficiency: The system must effectively leverage available computing resources within the cluster; 3) Completeness: The system must be capable of joining any pair of tuples from two streams and producing the result exactly once.

To achieve these goals, distributed stream join systems have explored efficient data stream schedulers to execute complex multi-stream join procedures in parallel [8]. Based on various tuple partitioning strategies, stream schedulers can be categorized into two types: random partitioning and hash partitioning.

The random partitioning strategy evenly assigns data tuples to join instances in self-stream and broadcasts all data tuples to join instances in other streams. Consequently, this method balances the system load by distributing data tuples evenly between join instances. However, due to the broadcast, this approach leads to data tuple replication, resulting in increased memory and communication overhead [9].

In contrast, hash partitioning strategy assigns data tuples to join instances based on their key values. Data tuples with the same key value are directed to the same join instance, effectively mitigating the overhead caused by random partitioning. However, real-world business scenarios often exhibit data skew [10], leading to stragglers in hash partitioning strategies [11], where some join instances take significantly longer than others to process data streams.

In a ride-sharing service scenario, the stream-joining application efficiently assigns passenger orders to nearby taxis. Within the join instances, the passenger stream queries the taxi driver stream to dispatch orders to taxis. Using the real-world DiDi Chuxing datasets [12], which includes both passenger order stream and taxi stream, the data is distributed across 100 join instances using a hash partitioning strategy. Fig. 2 shows the proportion of data processed by join instances, highlighting uneven distribution. Fig. 2 ranks join instances by their processed data size, revealing a highly skewed distribution of data tuples. For example, the top 10% of join instances process

Received 7 December 2023; revised 15 December 2024; accepted 27 May 2025. Date of publication 3 June 2025; date of current version 11 July 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62372419, in part by the Fundamental Research Funds for the Central Universities under Grant 265QZ2021001, and in part by Melbourne Chindia Cloud Computing (MC3) Research Network. Recommended for acceptance by R. Canal. *(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, VIC 3216, Australia (e-mail: shang.gao@deakin.edu.au).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Rajkumar Buyya is with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, University of Melbourne, Victoria 3010, Australia (e-mail: rbuyya@unimelb.edu.au).

Digital Object Identifier 10.1109/TC.2025.3575917

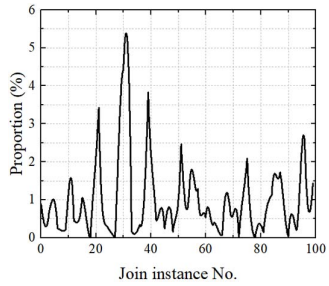

Fig. 1. Proportion of data sizes processed by join instance.

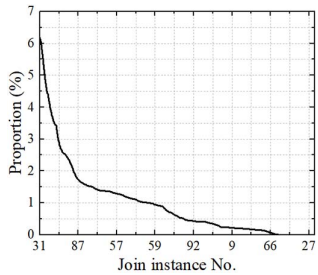

Fig. 2. Join instances ranked by their processed data size.

39.6% of the data tuples, while the bottom 10% process none. This skew occurs because large volumes of data features aggregate within the same instances, resulting in straggler issues [13]. Even with manual adjustments to data distribution for balance, stragglers may still occur over time.

To mitigate stragglers, an adaptive load balancing strategy for distributed stream join systems is always needed [14], [15]. It is expected to dynamically adjust the resource allocation between join instances at runtime, while considering the objectives of high performance, low cost, and high stability [16]. However, most state-of-the-art solutions [17], [18], [19] achieve load balancing in stream join systems by optimizing hot keys and employing random partitioning, incurring costly overhead when handling frequently fluctuating load changes.

A notable approach [17] introduced a scalable distributed stream join system to handle skewed loads. It identifies tuples with heavy workloads within the stream join system and evenly partitions them using a shuffling strategy, while other tuples are partitioned using hashing. Enabling shuffle policies in the stream join system necessitates broadcasting high-workload tuples from the store stream and evenly partitioning them in the join stream, resulting in unavoidable memory overhead due to the massive replicas in storage.

The most recent work [18] implemented a non-migrating load-balancing method for stream window join systems. It primarily transforms some tuples of the store stream in stragglers to low-load join instances using a storage routing table, enabling tuples with the same key in the store stream to be distributed across different join instances. To ensure join result completeness, the join routing table schedules corresponding tuples to join instances that store the same key of tuples in a dispersed manner. Although this approach effectively avoids memory overhead generated by tuple replicas, it introduces additional communication overhead. Moreover, both routing tables become challenging to maintain and incur increased system overhead with frequently triggered load balancing strategies.

Another recent work [19] combined hotspot detection and range routing strategies to mitigate stragglers and achieve balanced load, dynamically adjusting partitioning rules based on the unstable system load. By sampling and analyzing data streams within the current window, it can predict the distribution of data streams in the next window and adjust partitioning rules accordingly. However, guaranteeing join result completeness becomes challenging as it disregards late data tuples.

As such, our aim is to address the aforementioned issues using a lightweight data tuple scheduler. This scheduler can determine when and how to reschedule the data tuples at runtime based on the skewed data stream and the load difference between stragglers and other join instances, and minimize migration cost via a fine-grained model that adjusts the workload allocation of stragglers.

In an attempt to achieve these objectives, we propose Ls-Stream, a lightweight data tuple scheduler. It first collects system information, including partition communication loads, memory consumption, and computational demands. Then, Ls-Stream reschedules data tuples based on an analysis of this data. It enables one single join instance to oversee multiple partitions using a two-level routing strategy, which can effectively reduce the complexity and scale of conventional routing systems. Additionally, Ls-Stream makes trade-offs between state transfer overhead and migration benefits through a fine-grained model to determine the optimal partitions for migration. Through these strategies, Ls-Stream strives to attain real-time processing, resource efficiency, and completeness of join results to a significant extent.

A. Contributions

Our Ls-Stream is designed to mitigate stragglers and enhance the throughput and latency of distributed stream join systems. Our contributions can be summarized as follows:

(1) Provide models for partitioning, communication, matrix, and resource, along with the formalization of problems including imbalanced load between join instances and costs of state migration.
(2) Construct a two-level routing strategy for workload allocation by combining hash-based and key-based data partitioning. It allows for the redistribution of partitions from stragglers to other join instances, achieving load balance.
(3) Implement a fine-grained model for estimating the state migration cost resulting from workload adjustments for stragglers. It facilitates trade-offs between data transfer costs and migration benefits.
(4) Conduct experiments on DiDi Chuxing dataset. Through comprehensive evaluation, the proposed strategy provides promising improvements on throughput and latency, compared to existing state-of-the-art works.

B. Paper Organization

The rest of the paper is organized as follows: Section II discusses related work. Section III introduces the system model, including the partitioning model, communication model, matrix model, and resource model, as well as the system assumptions. Section IV formalizes the problems of imbalanced load between join instances and the state migration costs. Section V explains the framework of Ls-Stream and introduces the optimization methods to address the problems identified in Section IV. Section VI evaluates the performance of Ls-Stream. Section VII concludes our work and presents directions for future work.

II. Related Work

Distributed stream joins can be categorized into two main types: stream-static join and stream-stream join [20]. A stream-static join involves joining tuples from a real-time data stream with pre-existing, static data records. Relevant studies primarily focus on optimizing system computing efficiency. For example, a stratified-like sampling method [20] was used to select well-balanced representative geospatial data stream samples for emission to join instances. Additionally, a custom spatial data locality-aware partitioning method [21] was implemented to balance load while preserving spatial data locality.

In contrast, a stream-stream join involves two dynamically changing real-time data streams. Both streams continuously generate new data tuples with uncertain arrival times, making stream-stream joins more challenging than stream-static joins. Methods designed for stream-static joins are typically not applicable to stream-stream joins due to the incomplete views of both streams during the joining process.

To enhance the scalability of distributed stream-stream joins, certain studies [7], [9] introduced the Join-Matrix model. This model routes and stores tuples from one stream to randomly selected join instances while broadcasting and storing tuples from the other stream to all join instances. However, the Join-Matrix model increases system overhead due to redundant tuple storage. To address this issue, the join-biclique model was proposed [22]. Building on this model, BiStream [22], a scalable stream join system, and BiStream-ContRand [22], a content-aware router, were developed. BiStream uses hash partitioning to direct tuples to appropriate join instances for storage or joining. BiStream-ContRand uses a hybrid strategy: it hashes stored tuples to a subgroup and randomly selects a unit in the subgroup, while routing join tuples to all units in that subgroup.

To address load imbalances caused by skewed data streams in stream-stream join instances, a dynamic load balancing strategy named FastJoin was proposed [23]. When load imbalance occurs, this approach migrates tuples that cause heavy load skewness to low-load join instances. The keys of migrated tuples and their corresponding join instance are stored in a routing table. However, frequent strategy triggering complicates routing table maintenance, potentially requiring the storage of all data tuples in the worst-case scenario.

An adaptive range partitioning strategy named Nereus was developed for distributed stream-stream band join systems [24]. Nereus ensures controllable partition numbers and load balancing at minimal cost. It designs a dynamic routing table to partition tuples in data streams, treating partitions as processing units for storing and joining tuples. A migration benefit model further facilitates efficient adaption to skewed data streams. However, its effectiveness is limited for hash partitions, restricting its broader application.

In summary, these solutions provide valuable insights for optimizing distributed stream join systems. However, novel approaches are needed to better balance load among join instances, accommodate skewed data streams, and address the unique characteristics of distributed stream join systems.

TABLE I
Description of Primary Symbols Used in This Paper

Symbol	Description
j_n	Any of join instances
E_n	Average input rate of tuple for join instance j_n
j_h	Join instance with the heaviest load (i.e., the straggler)
μ_n	Average tuple processing rate of the join instance j_n
j_l	Join instance with the lowest load
LI	Load imbalance degree between join instances
p_o	One partition in join instance
W	A time window (set by the user)
R_{p_o}	Resources consumption of partition p_o
AST_{j_n}	Average sojourn time of tuples in instance j_n
R_{j_n}	Resource consumption of instance j_n
mif	Migration impact factor
α	Threshold of load imbalance
L	Average queue length in join instance j_n

III. System Models and Assumptions

Before addressing the issues of load imbalance between join instances and state migration cost and introducing our proposal, we first explain the partitioning model, communication model, matrix model, and resource model in distribute stream join systems. For enhanced clarity, Table I summarizes the primary notations used throughout the paper.

A. Partitioning Model

Data stream partitioning is a technique used in distributed systems to process data streams. It involves dividing these streams into multiple sub-streams that can be processed by multiple nodes in a cluster [25], [26]. Its primary goal is to enhance the system's processing capacity and improve its reliability. Let's consider a data stream $ds_k = \{dt_0, dt_1, ..., dt_i, ...\}$ in a distributed stream join system, where tuple dt_i in the data stream ds_k is mapped to join instances using a partitioning function $F(ds_k)$. The partitioning function $F(ds_k)$ assigns tuples to the appropriate join instances based on the tuples' characteristics. As a result, the data stream ds_k can be divided into multiple sub-streams $\{ds_{k0}, ds_{k1}, ..., ds_{kn}, ..., ds_{k(b-1)}\}$, and each divided sub-stream, such as ds_{kn}, is directed to join instance j_n. This relationship between ds_k and ds_{kn} can be described as follows (1).

$$ds_k = \bigcup_{n=0}^{b-1} ds_{kn}, \tag{1}$$

where b denotes the number of join instances.

Efficient data tuple partitioning is to assign a tuple dt to a specific sub-stream ds_{kn} and forward it to the join instance j_n. Therefore, the relationship between sub-streams can be described as (2).

$$\bigcap_{n=0}^{b-1} ds_{kn} = \emptyset. \tag{2}$$

B. Communication Model

In the distributed stream join system, assume it receives two streams, R and S, where their data tuples are forwarded to join instances for processing through a router. This router comprises shuffle instances and dispatcher instances. The load of this router remains balanced as shuffle instances employ a polling strategy to distribute tuples [27]. The dispatcher instances direct data tuples from streams R and S to join instances $J = \{j_0, j_1, ..., j_n, ..., j_{(b-1)}\}$. Let the communication load between dispatcher instance d_u and join instance j_n be denoted as e_{un}, then the input rate of join instance j_n can be calculated by (3).

$$Ir_{j_n} = \sum_{m=0}^{u-1} e_{mn}, \tag{3}$$

where u denotes the number of dispatcher instances.

Given the potential for transient fluctuations in the tuple arrival rate, we calculate the average of Ir_{j_n} by (4) to ease the impact of sudden fluctuations.

$$E_n = \frac{\int_{t_s}^{t_e} Ir_{j_n} dt - \max(Ir_{j_n}) - \min(Ir_{j_n})}{t_e - t_s}, \tag{4}$$

where E_n represents the average input rate of join instance j_n within the time interval $[t_s, t_e]$, excluding the maximum and minimum values of the input rate within this period. t_s and t_e denote the start and end times of the specified short period, which can be defined by users.

C. Matrix Model

The join operations between two input streams can be conceptualized as a join matrix. In this matrix, the X and Y axes represent tuples from the respective two data streams. The collection of tuples arranged within this join matrix is described as a tuple matrix, denoted as M. This tuple matrix includes all possible pairs of tuples from both streams, and these tuples are organized into partitions. Tuples from different streams residing in the same partition undertake the Cartesian product of R and S for the join operation. Let $m_{i,j}$ represent the tuple pair at the i^{th} row and j^{th} column of the M matrix, and pa_o denote the set of $m_{i,j}$ covered by partition p_o. The $pa_o \in Pa$ is represented by $m_{i,j}, m_{i',j'}$, where $m_{i,j}$ and $m_{i',j'}$ denote the upper left corner and lower right corner of the partition, respectively.

Specifically, the tuple matrix M has these characteristics for performing join operation [7]: interval-free, overlap-free and results completion.

(1) Interval-free. This requires that the complete set of partitions cover all input tuples from both R and S streams, which can be described by (5).

$$\bigcup_{m=0}^{o-1} pa_m^r = R, \bigcup_{m=0}^{o-1} pa_m^s = S \tag{5}$$

where o denotes the number of partitions, pa_m^r denotes tuples from stream R in partition pa_m, and pa_m^s denotes tuples from stream S in partition pa_m.

(2) Overlap-free. This requires that tuple pair $m_{i,j}$ from R and S streams is covered by the complete set of partitions $Pa = \{pa_0, pa_1, ..., pa_{o-1}\}$ at most once. If $\forall pa_i \in Pa$, $\forall pa_j \in Pa$ and $i \neq j$, then $pa_i \cap pa_j = \emptyset$.

(3) Results completion. This requires that all join results in matrix M are covered by partitions Pa, which can be described by (6).

$$\bigcup_{m=0}^{o-1} pa_m = M' \tag{6}$$

where M' denotes the area encompassing all tuples from both streams in matrix M.

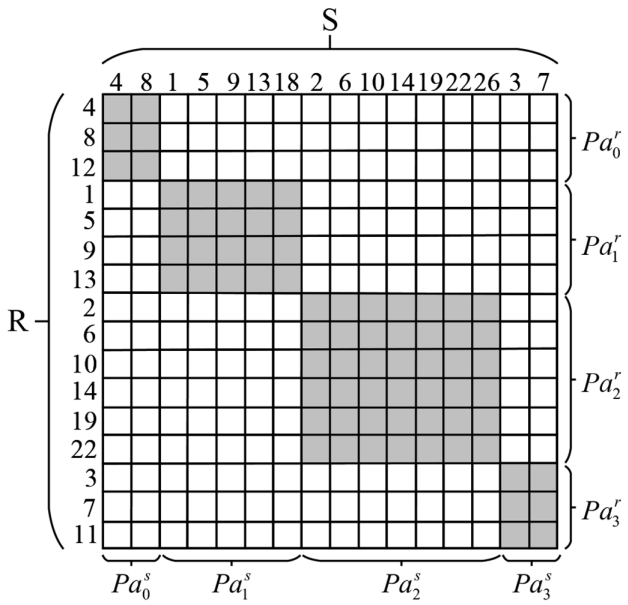

Fig. 3. Example of stream join where multiple partitions are managed by one join instance.

As shown in Fig. 3, data tuples from two streams are subjected to join operations using the matrix model. In this case, there are 12 tuples in each data stream, and these tuples are directed to their respective partitions by the Hash function. The X-axis of the matrix represents the key values of tuples from the S stream, while the Y-axis represents the key values of tuples from the R stream. The join condition for a tuple requires the key value of a tuple from the S stream matches that of a tuple from the R stream, and the computation area for join results in the matrix is shaded in grey. If a tuple from one stream does not find a matching tuple from the other stream, it will be cached for a certain period of time until it times out and is removed. From Fig. 3, we can observe that the tuple matrix satisfies all the three

aforementioned characteristics. Each partition is independent and all data tuples are covered.

D. Resource Model

The load of join instances primarily includes the resources used for storing and joining data tuples from different streams. Therefore, the resource model can be constructed based on the instance's consumption of memory and computing resources. For the distributed stream join system with two input data streams R and S, let's define R stream as the join stream and S stream as the store stream.

(1) Memory resource. Considering a time window W for each tuple in both streams within join instance j_n, memory consumption is evaluated based on the number of stored tuples from the S stream and the queue length of R stream tuples waiting to join. The memory consumption $R^m_{p_o}$ for a partition p_o can be calculated by (7).

$$R^m_{p_o} = ST^s_{p_o} + QL^r_{p_o} \tag{7}$$

Where $ST^s_{p_o}$ denotes the the number of tuples stored from the S stream within partition p_o, and $QL^r_{p_o}$ denotes the queue length of tuples from the R stream within partition p_o waiting to join. Furthermore, as the number of tuples stored in partition p_o increases, the memory load of this partition also increases. Hence, we use (7) to evaluate the resource consumption of partition p_o.

In Ls-Stream system, there may be multiple partitions in one instance. Therefore, the memory consumption $R^m_{j_n}$ of instance j_n can be calculated by (8).

$$\begin{aligned} R^m_{j_n} &= \sum_{p_o \in SP} (ST^s_{p_o} + QL^r_{p_o}) \\ &= E_n \cdot W \end{aligned} \tag{8}$$

where SP denotes the set of partitions managed by the instance j_n.

(2) Computing resource. When a data tuple dt from the R stream is emitted to the join instance j_n, it is compared with all stored tuples of the S stream within join instance j_n. Following this, it is joined with all tuples with the same key in the S stream. Consequently, the computing load $R^c_{p_o}$ of partition p_o can be calculated by multiplying the number of tuples from the S stream stored in partition p_o with the queue length of tuples from the R stream in partition p_o, as described in (9).

$$R^c_{p_o} = ST^s_{p_o} \cdot QL^r_{p_o} \tag{9}$$

Therefore, the computing load $R^c_{j_n}$ of instance j_n can be calculated by (10).

$$R^c_{j_n} = \sum_{p_o \in SP} (ST^s_{p_o} \cdot QL^r_{p_o}) \tag{10}$$

Traditional distributed stream join systems [17], [18], [19], [23] typically employ only one partition for each join instance. In contrast, each instance in Ls-Stream manages multiple partitions, which significantly reduces the computing area. Let's consider that partitions p_0, p_1, p_2 and p_3 are managed by a single join instance j_n. As depicted in Fig. 3, the computing area pa_0 of the join result matrix represents the computing resources consumed by partition p_0. The larger the computing area pa_0, the more computing resources partition p_0 consumes. From this Fig. 3, we can observe that Pa is much smaller than the full matrix area.

Based on the above description, the resources consumed by partition p_o during the time window W, denoted as R_{p_o}, can be calculated by (11).

$$R_{p_o} = \gamma \cdot R^m_{p_o} + (1-\gamma) \cdot R^c_{p_o}, \tag{11}$$

where γ is a weighting factor of memory and computing resource consumed by partition p_o and $0 < \gamma < 1$.

Then, the resource consumption R_{j_n} of instance j_n can be calculated by (12).

$$R_{j_n} = \sum_{p_o \in SP} R_{p_o} \tag{12}$$

E. Assumptions

In distributed stream join systems, the arrival times of data are unpredictable. The latest data tuples emitted by the data sources may arrive late at join instances. Therefore, we consider using sliding windows to join two data streams. We compare the timestamp of the latest tuple emitted by the data source with the timestamp in the stored stream, always maintaining the window size set by the user.

We assume that: (1) The threshold for load imbalance (explained in Section IV, with a default value of 1.0) is set by the user before submitting a stream application. (2) Data streams are dynamically changing, with their degree of skewness fluctuating over time. (3) All processors in the distributed environment have the same computing power, though the proposed model and algorithm also support settings with heterogeneous processors.

IV. Problem Formalization

In this section, we formalize the data stream scheduling problem in distributed stream join systems, which mainly includes the imbalanced load between join instances and the state migration costs.

A. Imbalanced Load Between Join Instances

The skewed data stream means that a massive number of data tuples are centrally distributed to one or more join instances for processing in parallel [28], which results in significantly longer data processing times for these instances compared to the average time [29]. Consequently, stragglers often occur in distributed stream join systems due to load imbalance between join instances. Furthermore, unbalanced load can become a bottleneck for processing data streams and affect the performance of systems [30]. Unbalanced load can pose the following risks: 1) Low resource utilization. Most data tuple processing is concentrated on a few join instances, causing the computing resources of others to remain idle. 2) Join instance downtime. The unbalanced load results in a large volume of data sets being

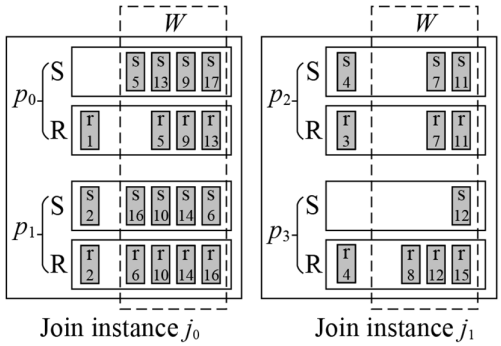

Fig. 4. An example of load imbalance degree *LI*.

distributed to only a few join instances, which can lead to data processing exceeding the capacity of those instances, ultimately causing join instance downtime.

To measure how skewed the data stream is, we introduce load imbalance degree. It mainly prevents the resources of join instances from becoming underutilized or overloaded by keeping it within a proper range.

Definition 1 (*Load imbalance degree*)*:* The load imbalance degree quantifies the deviation in resource utilization among join instances within a distributed stream processing system. It serves as a metric to evaluate the disparity or skewness in the distribution of data workload across these instances. The larger the load imbalance degree *LI*, the more skewed the data stream becomes.

The load imbalance degree *LI* is defined as the greater of the two relative differences from the mean to the maximum and minimum values in resource loads R_J of join instances and can be calculated by (13).

$$LI = \max\left(\frac{\max(R_J) - \bar{R}}{\bar{R}}, \frac{\bar{R} - \min(R_J)}{\bar{R}}\right) \tag{13}$$

where

$$\bar{R} = \frac{1}{size(J)} \cdot \sum_{n=0}^{size(J)-1} R_{j_n} \tag{14}$$

As shown in Fig. 4, there are two join instances j_0, j_1, where j_0 manages partitions p_0 and p_1, and j_1 manages partitions p_2 and p_3. In window W, the number of tuples in the S stream for partitions p_0, p_1, p_2, and p_3 are 4, 4, 2, and 1, respectively. The number of tuples in the R stream for partitions p_0, p_1, p_2, and p_3 are 3, 4, 2, and 3, respectively. According to Eq. 8, the memory resources for j_0 and j_1 are 15 and 8. According to Eq. 10, the computation resources for j_0 and j_1 are 28 and 7. The weighting factor γ is set to 0.5 by default, meaning memory resource and computing resource are equally important for system performance. Based on Eq. 12, the resource consumptions for j_0 and j_1 are 43 and 15, respectively. Given this information, the load imbalance degree *LI* can be determined as 0.48 using Eq. 13.

Theorem 1: In a distributed stream join system, there exists a positive mathematical relationship between the sojourn time of tuples and resource consumption of join instances.

Proof: If the average input rate of the join instance j_n is E_n and the average tuple processing rate of the join instance j_n is μ_n, the average sojourn time AST_{j_n} of j_n can be calculated by (15) based on the Little's law [31].

$$AST_{j_n} = \frac{L}{E_n} \tag{15}$$

where L denotes the average queue length in the join instance j_n, and can be calculated by (16).

$$L = \sum_{\lambda=0}^{\infty} \lambda \cdot pr_\lambda \tag{16}$$

where pr_λ denotes the probability of the instance j_n's existing λ data tuples when the data stream input rate is stable. A stable data stream input rate ensures that the number of tuples received by each join instance per unit of time does not exhibit significant variation, thus maintaining system stability. Once the system stabilizes, the tuple input rate should equal the system's tuple processing rate. Based on the equilibrium equation of M/M/1 model [31], [32], pr_λ can be calculated by (17).

$$pr_\lambda = \left(\frac{E_n}{\mu_n}\right)^\lambda \cdot pr_0 \tag{17}$$

where pr_0 denotes the probability that no data tuple exists in the instance and can be calculated by (18) based on the probability distribution condition [31].

$$\begin{aligned} pr_0 &= \frac{1}{1 + \sum_{\lambda=0}^{\infty} \left(\frac{E_n}{\mu_n}\right)^\lambda} \\ &= 1 - \frac{E_n}{\mu_n} \end{aligned} \tag{18}$$

Taking equations (17) and (18) into (16), it can get the average queue length L of instance j_n by (19).

$$\begin{aligned} L &= \sum_{\lambda=0}^{\infty} \lambda \cdot \left(\frac{E_n}{\mu_n}\right)^\lambda \cdot \left(1 - \frac{E_n}{\mu_n}\right) \\ &= \frac{E_n}{\mu_n - E_n} \end{aligned} \tag{19}$$

Taking (19) into (15), it can get the average sojourn time AST_{j_n} by (20).

$$AST_{j_n} = \frac{1}{\mu_n - E_n} \tag{20}$$

Taking (8) into (20), it can get the relationship between the sojourn time AST_{j_n} and the resources load by (21).

$$AST_{j_n} = \frac{W}{W \cdot \mu_n - R^m_{j_n}} \tag{21}$$

There is a negative mathematical relationship between resource consumption R_{j_n} and the data processing rate μ_n of instance j_n, with W as a constant. Therefore, the sojourn time AST_{j_n} of the instances will increase as resource consumption grows. Excessive resource consumption may cause the tuple processing rate of the instance to be lower than the input rate, i.e., $\mu_n < E_n$, leading to an elongation of instance j_n's sojourn time until it eventually becomes unresponsive due to the accumulation of a large number of data tuples within the instance. Consequently, it can be inferred

from this theorem that stragglers in join instances take longer time to process data streams.

The reason for stragglers consuming more resources is that a substantial number of tuples with the same characteristics are concentrated within the same instance, which leads to longer system latency and lower throughput. Therefore, the system's performance depends on instances with the heaviest load, i.e., stragglers. Based on **Theorem 1**, the objective function for this problem is $LI \leq \alpha$, where α represents the threshold for load imbalance. α ensures that the load difference between join instances remains within an acceptable range, mitigating the impact of stragglers on system performance while optimizing resource utilization in instances with low loads.

B. State Migration Costs

The state of instance j_n is a temporary cache for certain data tuples from R and S streams, and these tuples will be removed after performing the join operation. According to the resource model, it is evident that the state size of join instance j_n is closely correlated with its resource load. Therefore, when dealing with load imbalances between instances, we can migrate some of the state from the instance j_h with the heaviest load (i.e., straggler j_h) to the instance j_l with the lowest load. This migration involves selecting a subset SK from the full set $FS = \{dt_0, dt_1, ..., dt_{c-1}\}$ of data tuples in the straggler j_h, with the aim of minimizing the load difference between instances, where $SK \subset FS$. However, finding an optimal solution within a finite time is challenging because the selection of key values is an NP-complete problem [23].

Definition 2 (*Resource benefit*)*:* The resource benefit is the impact on load distribution caused by migrating certain tuples from the straggler instance to the lest-loaded instance.

The resource benefit ΔR can be calculated by (22).

$$\begin{aligned}\Delta R &= (R_{j_h} - R_{j_l}) - (R'_{j_h} - R'_{j_l}) \\ &= (R_{j_h} - R'_{j_h}) + (R'_{j_l} - R_{j_l})\end{aligned} \tag{22}$$

where R'_{j_h} and R'_{j_l} respectively denote the load of straggler j_h and instance j_l after performing the migration operation.

Stream join operation is carried out within each partition. Consequently, the load of each instance consists of the resources consumed by multiple partitions. When a migration operation is executed, an optimal combination of partitions is determined to achieve load balance. Then, ΔR can be simplified as (23).

$$\Delta R = 2 \cdot (R_{j_h} - R'_{j_h}) \tag{23}$$

Definition 3 (*Latency benefit*)*:* The latency benefit is determined by the difference between the latency increment and the latency decrement, which respectively represent the changes in processing time for data tuples in straggler and least-loaded instance before and after migration.

Based on the equation (19), the latency benefit lb can be calculated by (24).

$$\begin{aligned}lb &= (AST_{j_h} - AST'_{j_h}) + (AST_{j_l} - AST'_{j_l}) \\ &= \frac{\mu'_h - \mu_h + E_h - E'_h}{(\mu_h - E_h) \cdot (\mu'_h - E'_h)} + \frac{\mu'_l - \mu_l + E_l - E'_l}{(\mu_l - E_l) \cdot (\mu'_l - E'_l)}\end{aligned} \tag{24}$$

where AST', μ' and E' respectively denote the average sojourn time of tuples, the average tuple processing rate and the average input rate of the instance after performing the migration operation.

Definition 4 (*Migration impact factor*)*:* The migration impact factor measures the benefit derived from performing the migration operation and should aim to minimize the number of data tuples migrated.

The migration impact factor *mif* can be described by (25).

$$mif = \frac{\Delta R + lb}{card(SK)} \tag{25}$$

where $card(SK)$ denotes the cardinal number of the set SK. It is evident that a larger migration impact factor *mif* results in lower migration costs and a more balanced load between instances.

So, the objective function of state migration costs can be described as (26).

$$\min(mif) \tag{26}$$

subject to

$$\begin{cases} LI < \alpha \\ R'_{j_h} \geq R'_{j_l} \end{cases} \tag{27}$$

where the first condition is that the migrated data volume must maintain resource load imbalance within a tolerable range. The second condition is that the migrated data volume must ensure the j_h instance's resource load remains higher than the j_l instance's.

V. Ls-Stream: Architecture and Algorithms

Based on the above analysis, we propose a lightweight data tuple scheduler, called Ls-Stream, for stream join systems. Implemented on top of Apache Storm platform, Ls-Stream inherits all the features of Apache Storm and is capable of supporting one-time stream processing. In this section, we introduce Ls-Stream's architecture and algorithms for balancing load between join instances and minimizing migration costs.

A. System Architecture

As shown in Fig. 5, the system architecture of Ls-Stream consists of three main components: dispatcher, join instances and controller.

The dispatcher component plays a pivotal role in achieving load balancing between instances. It can be customized by implementing the CustomStreamGrouping interface on Apache Storm [23], [24]. Its primary function is to dispatch data tuples to partitions within join instances using the routing table. Skewed data streams can result in substantial load disparities among partitions, leading to an unbalanced load between instances. To rectify this imbalance, the deployment of partitions needs adjustment. When the load between instances becomes uneven, the routing table can be updated by the controller component. The dispatcher identifies the partitions to be migrated and blocks the data tuples within them. Subsequently, the

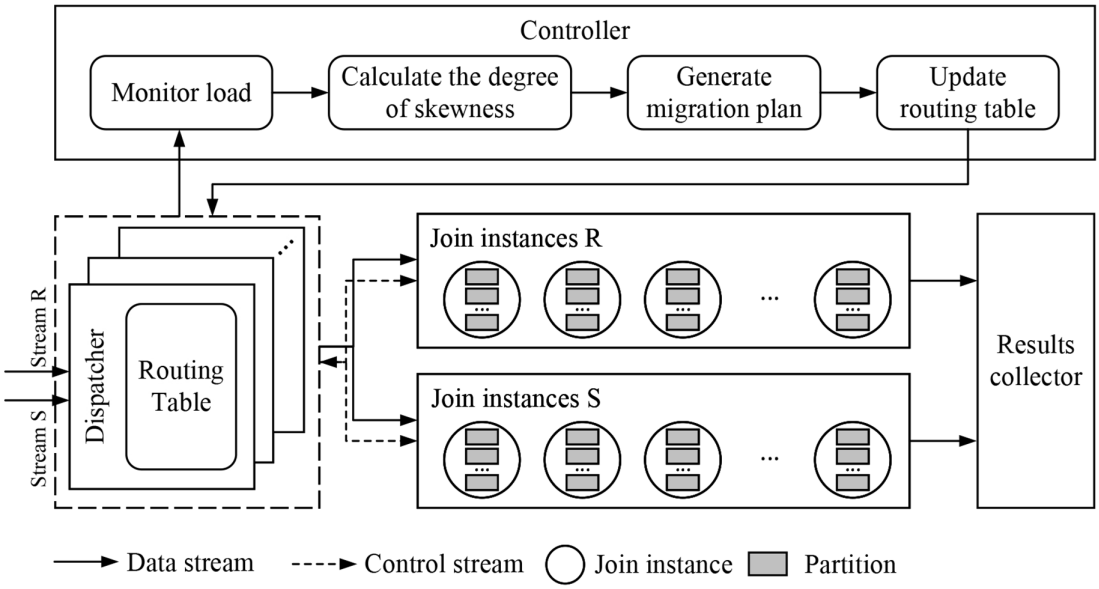

Fig. 5. Architecture of Ls-Stream.

dispatcher transmits migration information to the join instances via the control stream.

Join instances are composed of multiple partitions, with each partition serving as a processing unit for storing and joining data tuples. This partition approach effectively reduces the computing area required for join operations and accelerates the retrieval of stored data tuples from the S stream. Join instances receive two primary types of data streams: data stream containing tuples from R and S streams, and control stream conveying information about migrating partitions from straggler j_h to least-loaded instance j_l. When straggler j_h receives migration information, it initiates the migration of these specified partitions, guided by the migration benefits computed by the controller component. During data migrations, we reuse the data transmission channels provided by Apache Storm.

The controller is primarily responsible for collecting load information and conducting data analysis based on this information. It consists of the following four main stages: (1) Load Monitoring. It periodically tracks the tuple input rates of partitions from both R and S streams within instances using a counter. (2) Skewness Calculation. The degree of skewness between instances is computed based on the load information. If an imbalance is detected, the process proceeds to the next step. (3) Migration Plan Generation. Considering the uneven load, the controller determines which partitions should be migrated from straggler j_h to the least-loaded instance j_l to minimize migration costs. (4) Routing Table Update. Based on the migration plan, the controller generates a new routing table and synchronizes the dispatcher's router with the updated table. The controller does not significantly impact the performance of the stream join system, as it operates in isolation from the join instances.

In comparison to traditional distributed stream join systems, Ls-Stream offers several advantages: (1) Traditional solutions construct a routing table to redirect migrated data tuples, which can lead to scalability issues as the routing table size grows with the number of migrated tuples. Ls-Stream, however, constructs its routing table based on partitions, ensuring its size remains independent of the number of migrated tuples. (2) In traditional solutions, each instance maintains the state data for a single partition. Consequently, when tuples from R streams engage in join operations, querying globally stored tuples within the instance becomes necessary. In contrast, Ls-Stream allows each instance to manage state data for multiple partitions, effectively reducing the required computing area for joining tuples.

B. Balance Load Between Join Instances

Unbalanced load distribution between join instances can lead to certain instances becoming overloaded and acting as system bottlenecks, while others remain underutilized or even idle, resulting in inefficient resource allocation. Therefore, in the face of the challenge posed by massive data, a well-balanced load strategy is crucial for enhancing the performance of distributed stream join systems. This strategy aims to distribute data evenly across multiple join instances, reducing system latency and enhancing system availability. It should exhibit the following characteristics: (1) High concurrency. It should evenly distribute the workload among join instances and maximize system throughput. (2) Lightweight. Its load balancing algorithm should not consume excessive system resources, avoiding constraints on system performance. (3) High reliability. Data tuples should be routed to the correct join instance for processing, ensuring the completeness and reliability of data state.

Assume there exist join instances $J = \{j_0, j_1, ..., j_n, ..., j_{(b-1)}\}$ in the distribution stream join system. To balance load between these join instances, the data tuples $\{dt_0, dt_1, ..., dt_i, ...\}$ from upstream instances will be grouped according to the dynamic routing table. The table mainly maps the relationship $F(dt_i)$ between partitions $P = \{p_0, p_1, ..., p_m, ..., p_{(t-1)}\}$ and instances J, which can be described by (28).

$$F(dt_i) = p_m \rightarrow j_n \tag{28}$$

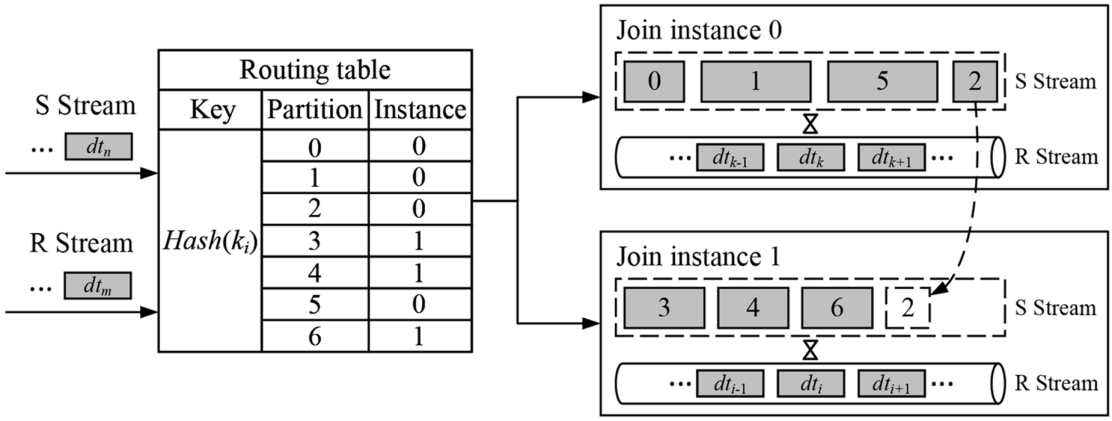

Fig. 6. Routing table of Ls-Stream.

where $F(dt_i)$ denotes that the tuple dt_i will be emitted to the partition p_m in instance j_n, and p_m can be calculated by (29).

$$p_m = Hash(dt_i(Key))\%card(P), \tag{29}$$

where $dt_i(Key)$ denotes the key of tuple dt_i and $card(P)$ denotes the cardinal number of the partition set P.

Based on the above description, emitting a data tuple from an upstream instance to a join instance mainly includes two steps: First, calculate the partition number to which the tuple will be emitted using (29). Second, consult the routing table to identify the corresponding join instance for the given partition number. As shown in Fig. 6, the routing table primarily comprises partition numbers and instance numbers. Each join instance manages the data state of multiple partitions from the S stream. During join operations on data tuples from the R stream, only the state of the relevant partition in the S stream is accessed, significantly narrowing the search scope.

In cases where the load disparity between join instances fails to meet the condition $LI \leq \alpha$, a significant load imbalance occurs within the distributed stream join system. Consequently, it becomes necessary to dynamically adjust the deployment of partitions. As depicted in Fig. 6, partition 2 is migrated from join instance 0 to join instance 1 to rebalance the load between these two instances. Migration costs will be discussed in the next subsection. The process of mitigating the load imbalance between join instance 0 and join instance 1 encompasses the following four steps: (1) Notify the routing table to block the emission of tuples from partition 2. (2) Migrate partition 2 from join instance 0 to instance 1. (3) Modify the instance number corresponding to partition 2 in the routing table from 0 to 1. (4) Release the previously blocked data tuples.

To balance the load between join instances, it is highly likely that the join instance hosting a partition may change, leading to the reallocation of online partitions. Notably, the size of Ls-Stream's routing table remains constant, regardless of the increasing number of migrated tuples, as the number of partitions remains unchanged.

C. Minimize Migration Costs

To achieve the balanced load between join instances, it is necessary to emigrate some partitions from stragglers. However, the migration of partitions comes at costs. To reduce these migration costs, we aim to minimize the migration impact factor by modeling the selection of partitions within a straggler as a combinatorial optimization problem. Solving combinatorial optimization problems, often NP problems, is typically better suited to heuristic algorithms. In this paper, we use the simulated annealing algorithm [33] to find an optimal solution.

There are several partitions $SP = \{p_0, p_1, ..., p_{u-1}\}$ within the straggler j_h, where $SP \subseteq P$, P is the set of all partitions, and u represents the number of partitions managed by the straggler j_h. We mark migrated partitions as 1 and non-migrated partitions as 0, and initialize certain partition markers randomly as the initial solution. Moreover, new solutions are generated iteratively within this algorithm, and must satisfy conditions (30) and (31). Otherwise, the algorithm proceeds to the next iteration.

$$\begin{aligned} &\frac{R'_{j_h} - \bar{R}}{\bar{R}} < \alpha \\ &\Rightarrow \frac{R_{j_h} - \sum_{p_o \in MP} R_{p_o} - \bar{R}}{\bar{R}} < \alpha \\ &\Rightarrow R_{j_h} - \sum_{p_o \in MP} R_{p_o} < \bar{R} \cdot (1 + \alpha) \end{aligned} \tag{30}$$

$$\begin{aligned} &R'_{j_h} - R'_{j_l} > 0 \\ &\Rightarrow R_{j_h} - \sum_{p_o \in MP} R_{p_o} - \left(R_{j_l} + \sum_{p_o \in MP} R_{p_o} \right) > 0 \\ &\Rightarrow R_{j_h} - R_{j_l} > 2 \cdot \sum_{p_o \in MP} R_{p_o} \end{aligned} \tag{31}$$

where MP denotes the set of partitions to be emigrated and $MP \subseteq SP$.

Next, we need to compare the migration impact factor *mif* of the new solution with the old one. If the new solution has a higher *mif* than the old one, the new solution is more valuable for migration. Otherwise, a probability equation $Pe(mif_{new}, mif_{old}, T)$ will be used to decide whether the new solution replaces the old one. $Pe(mif_{new}, mif_{old}, T)$ can be calculated by (32).

$$Pe(mif_{new}, mif_{old}, T) = e^{\frac{mif_{new} - mif_{old}}{T}} \tag{32}$$

Algorithm 1: Minimize Migration Costs.

Input: R_{j_h},R_{j_l},$\overline{R}$,SP;
Output: MP_{old};
Initialize the temperature T and the minimum temperature T_{min};
Initialize empty sets MP_{old} and MP_{new};
Initialize the temperature drop rate η;
for *each* p_o *in* SP **do**
  $p_o(flag) \leftarrow$ Random number from $\{0, 1\}$;
  if $p_o(flag) = 1$ **then**
    $MP_{old}.add(p_o)$;
  end
  if $R_{j_h} - \sum_{p_o \in MP_{old}} R_{p_o} > \overline{R} \cdot (1 + \alpha)$ || $R_{j_h} - R_{j_l} < 2 \cdot \sum_{p_o \in MP_{old}} R_{p_o}$ **then**
    $MP_{old}.delete(p_o)$;
    $p_o(flag) = 0$;
    Break;
  end
end
while $T > T_{min}$ **do**
  for $i = 0$ *to* $card(SP) - 1$ **do**
    Randomly select a partition p_j from SP_{old};
    $p_j(flag) = 1 - p_j(flag)$;
    Generate the new MP_{new} and SP_{new};
    if $R_{j_h} - \sum_{p_o \in MP_{new}} R_{p_o} < \overline{R} \cdot (1 + \alpha)$ || $R_{j_h} - R_{j_l} > 2 \cdot \sum_{p_o \in MP_{new}} R_{p_o}$ **then**
      $mif_{new} \leftarrow$ calculate the migration impact factor of MP_{new};
      if $mif_{new} > mif_{old}$ —— $random(0.0, 1.0) < Pe(mif_{new}, mif_{old}, T)$ **then**
        $MP_{old} \leftarrow MP_{new}$;
        $mif_{old} \leftarrow mif_{new}$;
        $SP_{old} \leftarrow SP_{new}$
      end
    end
  end
  $T = T \cdot \eta$;
end
return MP_{old}

where T denotes the temperature in the simulated annealing algorithm. This temperature parameter is to calculate the transition probability. A higher initial temperature increases the the probability of obtaining a high-quality solution, but it also prolongs the time taken for computation.

The algorithm for minimizing migration costs is described in Algorithm 1.

The input of algorithm 1 includes the load R_{j_h} of join instance j_h with high resource consumption, the load R_{j_l} of join instance j_l with low resource consumption, the average load $\bar{R}$ of join instances and the partitions set SP managed by instance j_h. The output of algorithm 1 is the set of partitions to be emigrated, denoted as MP_{old}. Step 1 to step 3 initializes the data required by this algorithm. Step 4 to step 14 generates initial solution. Step 17 to step 19 generates new solution. Step 21 calculates the migration impact factor of the new solution. Step 22 to step 26 generate acceptance probability according to the current temperature and the difference between the new and old solutions, and update the old solution. The time complexity of algorithm 1 is $O(\log T \cdot card(SP))$, where $card(SP)$ is the size of the set SP.

In algorithm 1, to minimize migration costs, the join instance with highest load can determine which partitions will be emigrated by combinatorial optimization. This algorithm enables the system to maximize migration benefits and minimize migration size to achieve load balancing between join instances.

TABLE II
SOFTWARE CONFIGURATIONS OF LS-STREAM

SoftWare	Version
Ubuntu	Ubuntu 20.04 64bit
Storm	Apache-Storm-1.2.4
Zookeeper	Zookeeper-3.5.7
JDK	Jdk1.8
Python	Python 2.7.2
MySql	MySql 5.6

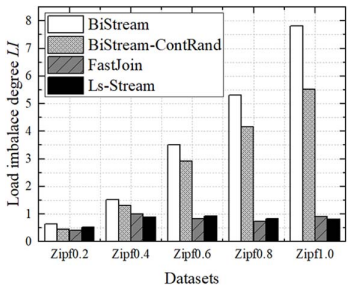

Fig. 7. Load imbalance degree on various datasets.

VI. PERFORMANCE EVALUATION

In this section, we evaluate the performance of Ls-Stream system. The experimental setup is first discussed, followed by the analysis of impact of skewed data streams on system performance, results and parameter settings.

A. Experimental Setup

The Ls-Stream framework is implemented on the top of Apache Storm and deployed on Ubuntu 20.04 operating system. The cluster consists of 20 computers with 2 as nimbus nodes and 18 as supervisor nodes. Each computer is equipped with an Intel(R) Xeon(R) X5650 CPU (dual-core, 2.4 GHz), 2GB of RAM, and a 100 Mbps Ethernet interface card. The zookeeper cluster is deployed on three computers which are multiplexed with the supervisor. Detailed software configurations are shown in Table II.

We compare BiStream, BiStream-ContRand [22], and FastJoin [23] under identical experimental conditions, including the dataset, data stream rate, number of instances per component, number of resources, and other relevant factors. To evaluate the

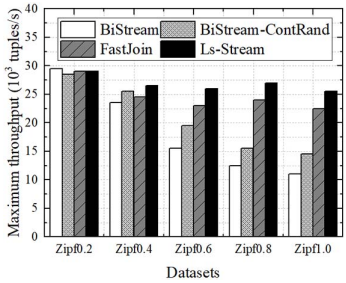

Fig. 8. System bottlenecks on various skewed datasets.

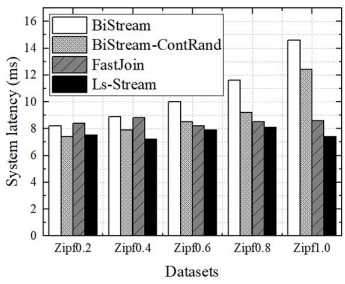

Fig. 9. System latency on various skewed datasets.

performance of Ls-Stream system, we utilize a large-scale real-world dataset provided by DiDi Chuxing GAIIA Initiative [12] and synthetic datasets [23] following the Zipf distribution. The DiDi Chuxing real-world dataset includes two kinds of data streams: order stream for passengers and driving track stream for taxis. The passenger order stream includes order ids (String type), timestamps (Long type), and GPS locations (Double type, Double type), while the driving track stream includes taxi ids (String type), GPS locations (Double type, Double type), and timestamps (Long type). The query workload involves matching every tuple in the passenger order stream with all tuples in the driving track stream. These data were collected by DiDi in Chengdu, China, during November 2016. The objective of the stream join application is to efficiently dispatch passenger orders to the nearby taxis. The aggregation logic of the join instance is to query all taxis nearby the passenger. However, finding an optimal dispatch of passenger orders is challenging. To simplify this problem, we utilize the Hilbert curve to convert GPS locations of passengers and taxis into one-dimensional data, and adopt this one-dimensional value as the key for stream join.

To generate varying degrees of skewness in the experimental datasets, we set the Zipf coefficient [18] of synthetic datasets to 0.2, 0.4, 0.6, 0.8, and 1.0, denoted as Zipf0.2, Zipf0.4, Zipf0.6, Zipf0.8, and Zipf1.0, respectively. The Zipf coefficient determines the skewness level of the synthetic datasets. A larger Zipf coefficient indicates more skewed datasets. These synthetic datasets are generated in a similar way to the approach described in [18], [23].

We design three distinct experiments, focusing on load imbalance, system performance, and parameter configurations. Firstly, we employ synthetic datasets to present the impact of load imbalance across join instances on system performance under stable data streams. Secondly, real-world datasets are applied to evaluate system performance and latency under varying computing resources and different numbers of join instances. Thirdly, we revisit synthetic datasets to investigate how adjusting the load imbalance thresholds affects system efficiency.

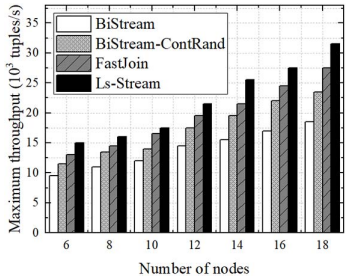

Fig. 10. System bottlenecks with different number of nodes.

B. Load Imbalance Degree

Load imbalance degree *LI* measures the skewness of a data stream. In this experiment, the impact of different *LI* values on system performance is evaluated using various synthetic datasets. 20 join instances are evenly deployed across the cluster, with the trigger load balancing factor α set to 1.0. This setting triggers the balancing load strategy when the imbalance degree between join instances exceeds 1.0.

Given a stable input rate of 5,000 tuples/s, Ls-Stream effectively balances the load among join instances when the system reaches a stable state. As shown in Fig. 10, the load imbalance degree remains stable for Ls-Stream and FastJoin across different skewed data streams. However, for BiStream and BiStream-ContRand, the load becomes increasingly unbalanced as the data stream skewness intensifies. Ls-Stream and FastJoin outperform BiStream and BiStream-ContRand in load balancing by effectively adapting to skewed input streams and evenly distributing resources among join instances. BiStream-ContRand exhibits a lower load imbalance degree compared to BiStream because it divides join instances into subgroups and balances the load within these subgroups. However, it still demonstrates a higher load imbalance degree than Ls-Stream due to its inability to address the load imbalance between these subgroups.

Given an increasing data stream rate and an increment of 500 tuples/s, the system bottleneck (i.e., maximum throughput) may be affected by skewed data streams. As shown in Fig. 10, the maximum bottleneck of BiStream and BiStream-ContRand is observed at 29,500 tuples/s and 28,500 tuples/s on Zipf0.2, respectively. The minimum bottleneck of BiStream and BiStream-ContRand is noted at 11,000 tuples/s and 14,500 tuples/s on Zipf1.0, respectively. Moreover, the bottleneck of both decreases with increasing Zipf coefficients. However, Ls-Stream and FastJoin maintain a stable bottleneck across varying data stream skewness. Although Ls-Stream and FastJoin exhibit a similar load imbalance in Fig. 10, Ls-Stream's system bottleneck is superior to that of FastJoin, attributed to Ls-Stream's lightweight router design. Moreover, this experiment also demonstrates that Ls-Stream and FastJoin exhibit a bottleneck similar to BiStream and BiStream-ContRand on Zipf0.2 because the load imbalance degree remains below the threshold α, preventing Ls-Stream and FastJoin from triggering the load balancing strategy.

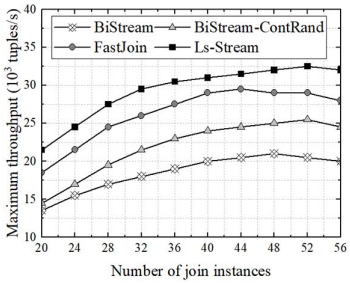

Fig. 11. System bottlenecks with different number of join instances.

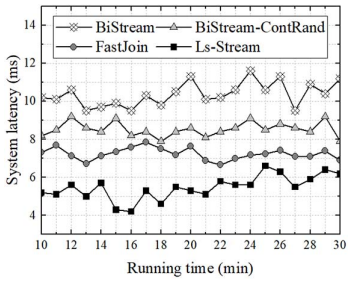

Fig. 12. System latency under stable data rate.

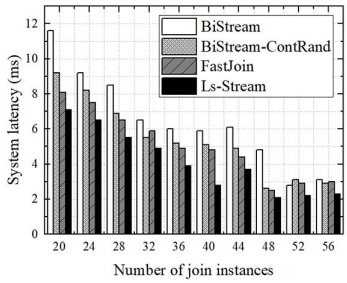

Fig. 13. System latency with different number of join instances.

Given a stable input rate of 5,000 tuples/s, the system latency can be affected by data streams with different degrees of skewness. As shown in Fig. 10, the latency of both BiStream and BiStream-ContRand systems increases as the Zipf coefficients rise. In contrast, Ls-Stream and FastJoin maintains a stable latency despite varying degrees of skewed data streams. In this experiment, Ls-Stream achieves a reduction in maximum system latency by 49.3% and 39.2% compared with BiStream and BiStream-ContRand, respectively, indicating its more efficient load balancing across join instances. The latency difference between Ls-Stream and BiStream-ContRand widens as data streams become more skewed. In addition, Ls-Stream achieves a reduction in maximum system latency by 18.1% compared with FastJoin. The lightweight router and the coarse-grained partition management of Join instances contribute to Ls-Stream's lower latency.

In summary, the above experiments reveal two points: (1) The performance of system worsens as the data stream skewness intensifies. It is primarily due to the concentration of most tuples on a single join instance for processing. (2) Ls-Stream demonstrates a significant improvement on the system performance.

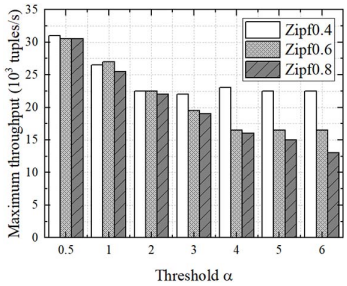

Fig. 14. System bottlenecks with different load imbalance thresholds.

This improvement can be attributed to Ls-Stream's dynamic load balancing across join instances through the lightweight routing table at runtime, thereby minimizing system latency and maximizing throughput.

C. Performance Results

The experiments evaluate Ls-Stream's performance on the real-world dataset, focusing on two metrics: system throughput and system latency.

(1) System throughput. Throughput assesses the system's resistance to load. It represents the number of tuples processed by the system per second, serving as an important metric for measuring system performance. A higher system throughput signifies better data processing capabilities. The experiments will evaluate the system bottleneck (i.e., maximum throughput) considering the resource count and parallelism of join instances.

When using 28 join instances and varying the count of compute nodes, Ls-Stream consistently exhibits higher maximum throughput compared to BiStream, BiStream-ContRand and FastJoin. This distinction becomes increasingly apparent with a larger deployment of nodes for join instances. As shown in Fig. 10, when deploying join instances across 6 nodes, the maximum throughputs of BiStream, BiStream-ContRand, FastJoin and Ls-Stream stand at 9,500 tuples/s, 11,500 tuples/s, 13,000 tuples/s and 15,000 tuples/s, respectively. This results in differences of 5,500 tuples/s, 3,500 tuples/s and 2,000 tuples/s, respectively. However, with 14 nodes deploying join instances, their respective maximum throughputs reach 15,500 tuples/s, 19,500 tuples/s, 21,500 tuples/s, and 25,500 tuples/s, showing discrepancies of 10,000 tuples/s, 6,000 tuples and 4,000 tuples/s, with Ls-Stream outperforming BiStream, BiStream-ContRand and FastJoin.

With the number of nodes set to 14 and varying counts of join instances, Ls-Stream consistently outperforms BiStream, BiStream-ContRand and FastJoin in terms of maximum throughput. More specifically, Ls-Stream has a better bottleneck improvement across any join instance counts. As shown in Fig. 14, when the number of join instances is under 48, the throughput of BiStream, BiStream-ContRand, FastJoin and Ls-Stream increases along with the number of join instances. At this stage, Ls-Stream improves system throughput by more than 52%, 27% and 6%, respectively, compared to BiStream, BiStream-ContRand and FastJoin. However, with a fixed number of node resources and an increase in join instances

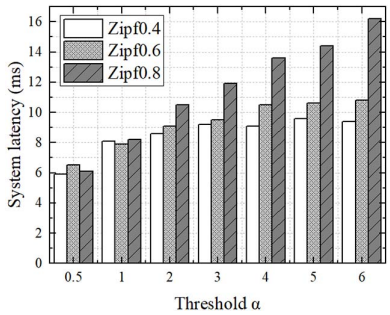

Fig. 15. System latency with different load imbalance thresholds.

exceeding 48, their throughput declines due to the resource consumption incurred by join instances.

In summary, compared to BiStream, BiStream-ContRand and FastJoin, Ls-Stream exhibits higher throughput and greater improvements under identical resource and join instance conditions. This is attributed to Ls-Stream's lightweight design to effectively balance the load between join instances, maximizing the system's throughput utilization.

(2) System latency. System latency refers to the time interval from the input of data tuples into the system to their complete processing, serving as an important metric for evaluating system performance. Low latency implies faster responses to user requests, thereby enhancing user experience. This experiment assesses system latency under a consistent data stream rate across varying number of join instances.

Given a stable input rate of 4,500 tuples/s, Ls-Stream exhibits shorter latency compared to BiStream, BiStream-ContRand and FastJoin. As shown in Fig. 14, the average latencies for BiStream, BiStream-ContRand, FastJoin, and Ls-Stream are 10.3 ms, 8.5 ms, 7.2 ms, 5.4 ms, respectively, after the system stablizes. Ls-Stream respectively reduces the average latency by 47.5%, 36.4%, and 25%, compared with BiStream, BiStream-ContRand and FastJoin. This experiment clearly shows that Ls-Stream maintains lower average latency than BiStream, BiStream-ContRand and FastJoin under stable input rate.

Given the data input rate of 3,000 tuples/s, for different number of join instances, the average latency of Ls-Stream is shorter than that of BiStream, BiStream-ContRand and FastJoin. As the number of join instances increases, the latency of BiStream, BiStream-ContRand, FastJoin, and Ls-Stream continues to decrease. However, Ls-Stream exhibits better latency improvements under any number of join instances. As shown in Fig. 14, Ls-Stream reduces the average operator latency by 35.8%, 23.8% and 18%, respectively, compared to BiStream, BiStream-ContRand and FastJoin.

In summary, based on the above experiments, it's evident that similar to system throughput, Ls-Stream also exhibits shorter average latency and greater improvement under different number of join instances. This is attributed to Ls-Stream's ability to balance the load between connected instances through lightweight routers and to accelerate tuple processing rates through coarse-grained partition management of Join instances.

D. Parameter Setting

A proper parameter enables join instances to process data at their best, which is important for improving the performance of distributed stream join systems. This experiment aims to assess the impact of load imbalance threshold α on system performance using different synthetic datasets.

Given an increasing data stream rate with increment of 500 tuples/s, the system bottleneck can be affected by varying threshold values of α. When the skewness of data stream is greater than the threshold α, Ls-Stream triggers the balancing strategy to improve the system bottleneck. However, when the load imbalance degree of data stream is less than threshold α, Ls-Stream degrades to standard hash partitioning. As shown in Fig. 14, When the Zipf coefficient of the synthetic dataset is 0.4, the maximum throughput of system stabilizes after threshold 2. This stabilization occurs because the load imbalance degree of Zipf0.4 (1.5) remains below 2. When the Zipf coefficient of the synthetic dataset is 0.8, the maximum throughput of system continues to decline, as the load imbalance degree of Zipf0.8 (5.3) consistently exceeds threshold α.

Given a data input rate of 3,000 tuples/s, system latency can be affected by different threshold values. If threshold α is less than the load imbalance degree of data streams, the system latency increases with rising threshold value. However, if threshold α is greater than the load imbalance degree of data streams, the system latency remains stable. As shown in Fig. 15, When the Zipf coefficient of the synthetic dataset is 0.6, the system latency increases until reaching threshold 4, after which it stabilizes.

These scenarios highlight that if the threshold setting is higher than the load imbalance degree of data streams, the load balancing strategy of Ls-Stream will not be effective. Therefore, selecting an appropriate parameter setting is important for enhancing system performance. From the experimental results above, it can be seen that the threshold should not be excessively high. Optimal adjustment of the threshold α within the [1], [2] range proves beneficial as it ensures a balanced sensitivity to skewness, effectively minimizing unnecessary and frequent activation of load rebalancing strategies.

VII. Conclusion and Future Work

Skewed data streams often lead to uneven loads among join instances, which can adversely affect system performance. Existing state-of-the-art solutions typically rely on complex routing strategies or resource-inefficient processing structures, making them vulnerable to dynamically skewed data distributions. To address these challenges, we introduced Ls-Stream, a data tuple scheduler designed to mitigate stragglers in distributed stream join systems. Our proposed strategy evenly distributes workloads across join instances, adapting to skewed data streams, while balancing data transfer costs and migration benefits. We have implemented Ls-Stream on top of the Apache Storm platform. Experiments demonstrate excellent performance improvement across various skewness levels of both

synthetic and real-world datasets, and present huge advantage over existing solutions in both system throughput and latency.

In our future work, we aim to integrate auto-scaling capabilities within the Ls-Stream system to dynamically adjust the number of join instances, further enhancing overall system performance. Additionally, we plan to extend the application of the proposed data tuple scheduler to single data streams and to support tumbling windows for processing data streams, broadening its versatility and applicability.

REFERENCES

[1] Y. Qiu, S. Papadias, and K. Yi, "Streaming hypercube: A massively parallel stream join algorithm," in *Proc. Int. Conf. Extending Database Technol.*, 2019, pp. 1–4.

[2] W. Chen, I. Paik, and Z. Li, "Cost-aware streaming workflow allocation on geo-distributed data centers," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 256–271, Feb. 2017.

[3] M. Wu, D. Sun, Y. Cui, S. Gao, X. Liu, and R. Buyya, "A state lossless scheduling strategy in distributed stream computing systems," *J. Netw. Comput. Appl.*, vol. 206, pp. 1–16, Oct. 2022.

[4] Z. Deng et al., "Spatial-keyword skyline publish/subscribe query processing over distributed sliding window streaming data," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2659–2674, Oct. 2022.

[5] M. Najafi, M. Sadoghi, and H. Jacobsen, "Scalable multiway stream joins in hardware," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 12, pp. 2438–2452, Dec. 2020.

[6] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join," *IEEE Trans. Big Data*, vol. 7, pp. 299–312, 2021.

[7] J. Fang, R. Zhang, Y. Zhao, K. Zheng, X. Zhou, and A. Zhou, "A-DSP: An adaptive join algorithm for dynamic data stream on cloud system," *IEEE Trans. Knowl. Data Eng.*, vol. 33, pp. 1861–1876, 2021.

[8] H. Zhang, M. Qiao, J. Yu, and H. Cheng, "Fast distributed complex join processing," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, 2021, pp. 2087–2092.

[9] J. Fang, R. Zhang, X. Wang, and A. Zhou, "Distributed stream join under workload variance," *World Wide Web*, vol. 20, pp. 1089–111, Jan. 2017.

[10] D. Sun, M. Wu, Z. Yang, A. Sajjanhar, and R. Buyya, "A Twotier coordinated load balancing strategy over skewed data streams," *J. SuperComput.*, vol. 79, pp. 1–29, Jun. 2023.

[11] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and X. Zhou, "Distributed stream rebalance for stateful operator under workload variance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, pp. 2223–2240, 2018.

[12] D. C. G. Initiative, "Didi Chuxing dataset," 2021. [Online]. Available: https://outreach.didichuxing.com/

[13] W. Li, D. Liu, K. Chen, K. Li, and H. Qi, "Hone: Mitigating stragglers in distributed stream processing with tuple scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 99, pp. 2021–2034, Aug. 2021.

[14] H. Chen, F. Zhang, and H. Jin, "PStream: A popularity-aware differentiated distributed stream processing system," *IEEE Trans. Comput.*, vol. 70, no. 10, pp. 1582–1597, Oct. 2021.

[15] J. Fang, P. Chao, R. Zhang, and X. Zhou, "Integrating workload balancing and fault tolerance in distributed stream processing system," *World Wide Web*, vol. 22, pp. 2471–2496, Jan. 2019.

[16] J. Fang, P. Zhao, A. Liu, Z. Li, and L. Zhao, "Scalable and adaptive joins for trajectory data in distributed stream system," *J. Comput. Sci. Technol.*, vol. 34, pp. 747–761, Jul. 2019.

[17] F. Zhang, H. Chen, and H. Jin, "Simois: A scalable distributed stream join system with skewed workloads," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2019, pp. 176–185.

[18] Q. Wang, D. Zuo, Z. Zhang, S. Chen, and T. Liu, "An adaptive non-migrating load-balanced distributed stream window join system," *J. SuperComput.*, vol. 79, pp. 8236–8264, Dec. 2023.

[19] 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 Gener. Comput. Syst.*, vol. 141, pp. 67–80, Apr. 2023.

[20] I. M. A. Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "SpatialSSJP: QoS -aware adaptive approximate stream-static spatial join processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 1, pp. 73–88, Jan. 2024.

[21] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "Locality-preserving spatial partitioning for geo big data analytics in main memory frameworks," *in Proc. IEEE Global Commun. Conf.*, 2020, pp. 1–6.

[22] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu, "Scalable distributed stream join processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 811–825.

[23] S. Zhou, F. Zhang, H. Chen, H. Jin, and B. B. Zhou, "Fastjoin: A skewness-aware distributed stream join system," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS),* 2019, pp. 1042–1052.

[24] S. Yu, H. Chen, and H. Jin, "Nereus: A distributed stream band join system with adaptive range partitioning," *IEEE Trans. Consum. Electron.*, vol. 69, no. 4, pp. 949–961, Nov. 2023.

[25] X. Huang, Z. Shao, and Y. Yang, "POTUS: Predictive online tuple scheduling for data stream processing systems," *IEEE Trans. Cloud Comput.*, vol. 10, no. 4, pp. 2863–2875, Oct./Dec. 2022.

[26] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 6, pp. 1860–1876, Jun. 2023.

[27] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards concurrent stateful stream processing on multicore processors," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE),* 2020, pp. 1537–1548.

[28] Y. Li and B. C. Lee, "Phronesis: Efficient performance modeling for high-dimensional configuration tuning," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, pp. 1–26, 2022.

[29] G. van Dongen and D. V. den Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Aug. 2020.

[30] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *J. Syst. Softw.*, vol. 127, pp. 205–216, May 2017.

[31] S. Wang, X. Li, and R. Ruiz, "Performance analysis for heterogeneous cloud servers using queueing theory," *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 563–576, Apr. 2020.

[32] J. Prados-Garzon, P. Ameigeiras, J. J. Ramos-Munoz, J. Navarro-Ortiz, P. Andres-Maldonado, and J. M. Lopez-Soler, "Performance modeling of softwarized network services based on queuing theory with experimental validation," *IEEE Trans. Mobile Comput.*, vol. 20, no. 4, pp. 1558–1573, Apr. 2021.

[33] T. Shirai and N. Togawa, "Multi-spin-flip engineering in an Ising machine," *IEEE Trans. Comput.*, vol. 72, no. 3, pp. 759–771, Mar. 2023.

Minghui Wu received the bachelor's degree in network engineering from Zhengzhou University of Aeronautics, Zhengzhou, China, in 2020. He is currently working toward the Ph.D. degree with the School of Information Engineering, China University of Geosciences, Beijing, China. His research interests include big data stream computing, distributed systems, and blockchain.

Dawei Sun received the Ph.D. degree in computer science from the Northeastern University, China, in 2012. He conducted the Postdoctoral Research with the Department of Computer Science and Technology, Tsinghua University, China, in 2015. He is a Professor with the School of Information Engineering, China University of Geosciences, Beijing, P.R. China. His 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 (Member, IEEE) received the Ph.D. degree in computer science from the Northeastern University, China, in 2000. Currently, she is a Senior Lecturer with the School of Information Technology, Deakin University, Geelong, Australia. Her research interests include distributed system, cloud computing, and cyber security.

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor of computer science with the State University of New York. He is among the world's top five most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He received the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He is an AAAS Fellow and an AAIA Fellow. He is also a member of Academia Europaea (Academician of the Academy of Europe).

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry Distinguished Professor and the Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Victoria, 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 172 with 158,900+ citations). He is among the world' s top two most influential scientists in distributed computing in terms of both single year impact and career-long impact based on a composite indicator of Scopus citation database.