A Popularity-Aware Discriminative Grouping Strategy in Distributed Stream Computing Systems

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

***Abstract*—Stream grouping strategy plays an important role in stateful stream computing environments. Many existing grouping strategies overlook various cost factors associated with grouping while balancing stream load. To overcome this limitation, we propose Pd-Stream, a popularity-aware discriminative grouping strategy that identifies the hot keys in dynamic real-time streams and assigns them to instances with high balance and low cost. Our solution includes: (1) A stream application model is constructed, along with a skewed data stream model and a data stream grouping model. Data stream grouping optimization problems are formalized. (2) A hot key probability estimation algorithm is designed, which estimates real-time probabilities of hot keys based on their popularity within the sampling window. (3) An instance assignment algorithm is designed using dynamic routing. This algorithm determines the minimal number of candidate instances based on the probabilities of hot keys, and selects the target instance with the lowest load through a dynamic routing table. Experimental results show that Pd-Stream provides near-optimal load balancing with low memory, achieving load imbalance as low as 10^{-5} and replication factor as low as 1.74. It outperforms state-of-the-art works, reducing latency by 27%–46% and improving throughput by 23%–52%.**

***Index Terms*—Stream computing system, Grouping strategy, Load balancing, Popularity-aware, Hot key identification.**

I. Introduction

In the big data era, there has been a growing demand for real-time monitoring and analysis of massive volumes of data [1]. Distributed stream computing systems, such as Storm [2], Heron [3], Spark [4] and Flink [5], provide effective solutions for processing dynamic and volatile data streams with high-throughput and low-latency [6], [7]. These distributed stream computing systems have been widely deployed in different domains, such as social network analysis [8], real-time risk detection [9], and Internet of Things (IoT) [10].

To achieve high system throughput, operator instances in a stream application are deployed across multiple computing nodes. Stream grouping strategies distribute data tuples to the target operator instances on different nodes for parallel execution. Specifically, Shuffle Grouping (SG) is typically used for stateless operators, where tuples are randomly and evenly assigned to downstream operator instances. Key Grouping (KG) is often used for stateful operators, where a specified field value in tuples serves as a key, and tuples with the same key are assigned to the same downstream operator instance.

However, in real-world scenarios, the distribution of keys in data streams is often skewed, typically exhibiting a pronounced long-tail property, where about 20% of frequent keys are associated with more than 80% of the tuples [3]. For instance, in social media, a small number of trending topics usually attract massive user discussions, while in network security, specific target IPs can trigger a large concentration of alert events [11]. If the system cannot effectively discriminate these hotspot keys, a large number of tuples will be mapped to a few operator instances [12]. This results in resource saturation on these instances, while other instances remain underutilized, thereby causing severe load imbalance.

To deal with skewed data stream for stateful operators, traditional solution is operator migration [13]–[16]. Once a situation of load imbalance is detected, the system migrates part of the keys and their associated states away from the overloaded instances. To ensure consistency in mapping keys to downstream instances, the source operator needs to maintain routing tables after state migration. However, in typical network-mining applications, each routing table can easily contain billions of keys [15], which leads to enormous memory costs and greatly limits overall system performance.

To avoid the overhead incurred by operator migration, PKG [17] is a widely adopted solution to addressing the load imbalance caused by skewed data streams for stateful operators. It employs a key splitting technique to assign each key to two instances. While this solution is effective for mildly skewed data streams, it struggles with highly skewed streams and large-scale instances. Based on PKG, Anis et al. [18] uses a heavy hitter algorithm to identify keys that are significantly more frequent than others, referred to as "hot keys", and allocates these hot keys across more than two instances to achieve load balancing. Given that hot keys often evolve over time, some grouping strategies [12], [19] are dedicated to the accurate identification of recent hot keys in dynamic real-time streams. These approaches enhance load balancing through reliable identification of hot keys.

In solutions that utilize key splitting for stream grouping, tuples associated with each key are processed by multiple instances, each instance generating a partial processing result. The final result is then obtained through a downstream

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

Dawei Sun, Minghui Wu and Jie Wen are with the School of Information Engineering, China University of Geosciences, Beijing, 100083, China (e-mail: sundaweicn@cugb.edu.cn; wuminghui@email.cugb.edu.cn; wenjie2573@email.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 Quantum Cloud Computing and Distributed Systems (qCLOUDS) Lab, School of Computing and Information Systems, The University of Melbourne, Grattan Street, Parkville, Victoria, 3010, Australia (e-mail: rbuyya@unimelb.edu.au)

aggregation step. While ensuring load balancing, these solutions often overlook additional costs related to memory and aggregation [20]. To reduce the memory and aggregation costs caused by key splitting, several holistic grouping algorithms [21]–[23] have been proposed. These algorithms consider both the load on instances and the routing information for hot keys. By calculating a comprehensive score that factors in both load balancing and key splitting for each candidate instance through an objective function, the instance with the highest score is selected as the target instance for the incoming data tuple. While these algorithms achieve promising load balancing, they ignore these overheads: memory cost from maintaining key state across multiple candidate instances and time cost from the grouping computation itself.

As such, we propose Pd-Stream, a popularity-aware discriminative grouping strategy with high balance and low cost. Unlike existing methods that either split keys aggressively or not at all, Pd-Stream dynamically calculates the real-time popularity of each hot key and determines the appropriate number of downstream instances to which it should be split. Our novelty lies in this fine-grained control: instead of just migrating a hot key to another instance, we intelligently distribute its load across an optimally calculated number of instances based on its popularity probability. This approach offers two key advantages over existing work. First, by tailoring the degree of parallelism to the actual popularity of each key, we achieve a higher degree of load balance than methods that treat all hot keys uniformly. Second, by only splitting the most popular keys and controlling the extent of the split, we significantly reduce the size of the required routing information, thereby minimizing the memory and aggregation costs that burden conventional state-migration strategies. Our contributions are as follows:

(1) We investigate stateful grouping for skewed data streams and formalize data stream grouping optimization problems by modeling stream application, skewed data stream, and data stream grouping.

(2) We propose a hot key probability estimation algorithm that leverages sliding window sampling. By optimally configuring the length of the sampling window based on a predefined hot key probability threshold, we achieve precise real-time estimation of hot key probability with minimal time and memory costs.

(3) We propose an instance assignment algorithm that first determines the minimal number of candidate instances for load balancing, informed by the probabilities of hot keys. It then selects the target instance with the lowest load through a dynamic routing table, minimizing both the memory cost of the table and the time cost of target instance selection.

(4) We implement Pd-Stream on the Apache Storm platform and evaluate metrics, including load imbalance, replication factor, latency and throughput, to verify the efficiency of the proposed stateful grouping strategy.

The rest of this paper is organized as follows: Section II discusses related work. Section III introduces relevant models; Section IV formalizes the grouping optimization problems; Section V presents the Pd-Stream system and its main algorithms; Section VI evaluates the performance of Pd-Stream; Section VII concludes our work and outlines future directions.

II. Related work

In this section, we review related studies across two main categories: data stream grouping and hot key identification.

A. Data stream grouping

Key grouping strategies often lead to load imbalance in tuple transfer between upstream and downstream instances when processing skewed data streams. Consequently, optimizing load balancing for data stream grouping has been widely studied in recent years.

Key-splitting-based state aggregation. To mitigate load imbalance in skewed data streams, Chen et al. [24] extended the "power of two choices" [17] with a $(1 + \beta)$-choice partitioning scheme, where a fraction $\beta \in (0, 1)$ of keys are selectively split among multiple candidate instances. Zhang et al. [25] further proposed Back Propagation Grouping (BPG), which periodically exchanges global load information to enhance balance. However, limiting hot keys to two instances remains inadequate under extreme skew. To address this, Nasir et al. [18] developed D-Choices (D-C) and W-Choices (W-C), which detect hot keys and distribute their load across multiple instances, while non-hot keys continue to follow PKG. These solutions achieve load balancing, but often at the expense of increased memory and aggregation overhead.

Cost-aware and learning-based grouping. To minimize load imbalance while considering memory and aggregation costs, Katsipoulakis et al. [21] proposed a cost-based cardinality-aware grouping strategy. It maps each key to two downstream instances and selects the target based on instance cardinality and load. More recently, Zapridou et al. [26] proposed Dalton, an adaptive grouping strategy that employs reinforcement learning to balance load and cost under dynamic stream conditions, updating its global policy through distributed state aggregation. However, these solutions ignore the cost incurred by maintaining key state across multiple candidate instances.

Our proposed Pd-Stream builds on the key-splitting technique in [17]. A dynamic routing table adaptively adjusts key mapping according to the estimated probabilities to balance load and control the degree of key splitting. The instance assignment algorithm further leverages the probability and routing tables to select target instances efficiently, thereby reducing grouping latency.

B. Hot key identification

Current stream grouping strategies generally use approximate stream computing methods for hot key identification [23]. By estimating the frequency of items in the data stream, high-frequency items can be determined to identify hot keys [22]. Approximate stream grouping can be broadly divided into two categories: counter-based and sketch-based methods.

Counter-based grouping. A stream summary structure [27], [28] is used to record approximate frequencies of data items. It consists of m counter slots storing items and their

frequency counts. When a new item arrives, the algorithm replaces the item with the lowest frequency, setting the new item's count to $n_{min}+1$, where n_{min} is the minimum counter value. These algorithms have a fixed space overhead of $O(m)$. However, they may introduce significant errors when the data stream distribution changes rapidly, as they treat all new items as potential high-frequency items.

Sketch-based grouping. The frequency of data items is estimated by maintaining a two-dimensional array [29], [30]. Independent hash functions map each data item to positions in one-dimensional arrays, and the corresponding counters at these positions are incremented. The estimated frequency of a data item is taken as the minimum value among all its mapped counters. However, the multi-hashing strategy improves frequency estimation accuracy at the cost of higher memory usage [31], as each cell in the two-dimensional array serves to aggregate hash collisions [12].

The aforementioned methods mainly focus on identifying hot items from the beginning of the measurement period and often overlook dynamic changes in the data stream, making it difficult to identify hot items in real time. Therefore, Pd-Stream detects hot keys and estimates their occurrence probabilities using a sliding-window sampling mechanism to update hot items dynamically.

III. System model

In this section, we formalize the foundational models in a stateful stream computing environment, including the stream application model, skewed data stream model, and data stream grouping model. For the sake of clarity, in Table I, we summarize the main notations used throughout the paper.

TABLE I
Description of main symbols used in the Pd-Stream.

Symbol	Description
v_i	The i-th vertex of the streaming application
e_{v_i,v_j}	A data stream path from vertex v_i to v_j
$D(e_{v_i,v_j})$	Instance set of downstream vertex v_j of vertex v_i
$n_{D(e_{v_i,v_j})}$	Number of instances in $D(e_{v_i,v_j})$
$L^w_{v_{j,n}}$	Load of downstream instance $v_{j,n}$ within a window
$LI^w_{D(e_{v_i,v_j})}$	Load imbalance degree between downstream instances
$RF^w_{D(e_{v_i,v_j})}$	Replication factor of key splitting for $D(e_{v_i,v_j})$
X_k	Appearance probability of hot key k
$P_k(X_k = n_k)$	Probability of hot key k appearing n_k times
α	Limit of the upper bound of $P_k(X_k = 0)$
$\hat{n}_{C(D(e_{v_i,v_j}))_k}$	Estimated number of candidate instances for key k
$C(D(e_{v_i,v_j}))_k$	Set of candidate instances for the incoming tuple

A. Stream application model

The logical topology of a stream application can be modeled as a Directed Acyclic Graph (DAG) $G = (V(G), E(G))$ [32], where $V(G) = \{v_i | i \in 1, 2, \ldots, \mathcal{N}\}$ is a finite set with $\mathcal{N}$ vertices and each vertex $v_i \in V(G)$ represents an operator with a specific function defined by the user. $E(G) = \{e_{v_i,v_j} | v_i, v_j \in V(G), i \neq j\}$ is a finite set of directed edges. Each edge e_{v_i,v_j} indicates a data stream path from vertex v_i to vertex v_j, where v_i and v_j denote the upstream and downstream vertices of e_{v_i,v_j}, respectively.

Vertex v_i comprises multiple instances that work in parallel, with each instance executing the same function. For a vertex pair v_i and v_j connected via e_{v_i,v_j}, we use $U(e_{v_i,v_j}) = \{v_{i,m} | m \in 1, 2, \ldots, n_{U(e_{v_i,v_j})}\}$ to represent the instance set of upstream vertex v_i, and $D(e_{v_i,v_j}) = \{v_{j,n} | n \in 1, 2, \ldots, n_{D(e_{v_i,v_j})}\}$ to represent the instance set of downstream vertex v_j. $n_{U(e_{v_i,v_j})} = |U(e_{v_i,v_j})|$ denotes the number of instances in $U(e_{v_i,v_j})$, and $n_{D(e_{v_i,v_j})} = |D(e_{v_i,v_j})|$ denotes the number of instances in $D(e_{v_i,v_j})$.

B. Skewed data stream model

Each data stream $ds = \{dt_1, dt_2, \ldots\}$ consists of a sequence of data tuples [29]. Each data tuple dt can be represented as a triplet $\langle \tau, k, v \rangle$, where τ, k, and v represent the timestamp, key, and value of data tuple dt, respectively. If the keys in data stream ds follow a skewed distribution D in a finite key space K, it means some of the keys appear more frequently than others. Let p_k represent the probability of key k appearing in the data stream ds. We can rank the keys based on their p_k values, with a higher ranking indicating a higher probability of the key appearing in the data stream. Therefore, the descending probability ranking of the keys is as follows: $p_{1^{st}} \geq p_{2^{nd}} \geq \ldots \geq p_{|K|^{th}}$, with $\sum_{k \in K} p_k = 1$. We define the key with a probability greater than or equal to a set threshold θ as a hot key. The set of hot keys $H(k)$ can be described by (1).

$$H(k) = \{k \in K \mid p_k \geq \theta\}. \tag{1}$$

To achieve lightweight hot key identification, we set a static threshold θ based on PKG [17]. However, using a fixed θ has certain limitations. It cannot flexibly control the number of hot keys under different skew levels. In extremely skewed data streams, most keys may either exceed θ or fall below it, which can make the grouping strategy suboptimal. A dynamic threshold that adapts to the observed key distribution could better handle such cases, although it would increase computational overhead and system complexity.

C. Data stream grouping model

Each data tuple can be emitted to a specific operator instance through the grouping strategy [16]. Pd-Stream incorporates a built-in window to monitor data stream runtime information for identifying the popularity of keys. Each data tuple passes through this built-in window and is distributed to downstream instances. Then, each data stream ds can be divided into a series of logical windows $W(ds) : ds \rightarrow \{ds^1, ds^2, \ldots, ds^w, \ldots\}$ based on time or data tuple count. Data tuples within window ds^w are assigned from upstream instances $U(e_{v_i,v_j})$ to downstream instances $D(e_{v_i,v_j})$ via e_{v_i,v_j} using a stream grouping function $G(ds^w_{U(e_{v_i,v_j})}) : ds^w_{U(e_{v_i,v_j})} \rightarrow D(e_{v_i,v_j})$, where $ds^w_{U(e_{v_i,v_j})}$ denotes the data tuples in the upstream instances $U(e_{v_i,v_j})$ within window ds^w.

$n^w_{v_{i,m},v_{j,n}}(dt) = |ds^w_{v_{i,m},v_{j,n}}|$ denotes the number of data tuples within window ds^w assigned from the upstream instances $v_{i,m}$ to the downstream instance $v_{j,n}$. The total number $n^w_{v_{j,n}}(dt)$ of data tuples within window ds^w assigned from

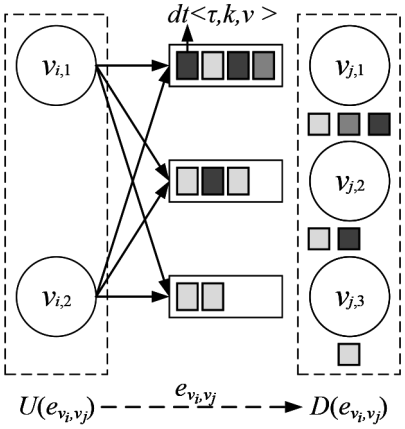


Fig. 1. Stream grouping from upstream to downstream instances.

upstream instances $U(e_{v_i,v_j})$ to the downstream instance $v_{j,n}$ of $D(e_{v_i,v_j})$ can be calculated by (2).

$$n^w_{v_{j,n}}(dt) = \sum_{m=1}^{n_{U(e_{v_i,v_j})}} n^w_{v_{i,m},v_{j,n}}(dt). \tag{2}$$

$n^w_{v_{i,m},v_{j,n}}(k) = \|ds^w_{v_{i,m},v_{j,n}}\|$ denotes the number of distinct keys within window ds^w assigned from the upstream instances $v_{i,m}$ to the downstream instance $v_{j,n}$. The total number $n^w_{v_{j,n}}(k)$ of distinct keys within window ds^w assigned from upstream instances $U(e_{v_i,v_j})$ to the downstream instance $v_{j,n}$ can be calculated by (3).

$$n^w_{v_{j,n}}(k) = \sum_{m=1}^{n_{U(e_{v_i,v_j})}} n^w_{v_{i,m},v_{j,n}}(k). \tag{3}$$

As depicted in Figure 1, nine data tuples within a window ds^w, containing three keys, are distributed from the upstream instances $U(e_{v_i,v_j}) = \{v_{i,1}, v_{i,2}\}$ to the downstream instances $D(e_{v_i,v_j}) = \{v_{j,1}, v_{j,2}, v_{j,3}\}$ via the stream grouping function $G(ds^w_{U(e_{v_i,v_j})})$, where the number of tuples for $v_{j,1}$ is $n^w_{v_{j,1}}(dt) = 4$, the number of keys for $v_{j,1}$ is $n^w_{v_{j,1}}(k) = 3$; similarly, $n^w_{v_{j,2}}(dt) = 3$, $n^w_{v_{j,2}}(k) = 2$; and $n^w_{v_{j,3}}(dt) = 2$, $n^w_{v_{j,3}}(k) = 1$.

IV. Problem Statement

Previous key splitting-based grouping strategies have struggled to achieve load balancing and control key splitting under low-latency conditions. To address the challenges, we formalize the following grouping-related problems: load balancing and key splitting, before proposing our optimization solutions.

A. Load balancing optimization

We aim to optimize the computational load distribution across instances for better load balancing. The computational load of an instance is proportional to the number of data tuples it should process. Therefore, we quantify the load of a downstream instance $v_{j,n}$ within window ds^w, denoted as $L^w_{v_{j,n}}$, as the total number of data tuples $n^w_{v_{j,n}}(dt)$ assigned from upstream instances $U(e_{v_i,v_j})$ to the downstream instance $v_{j,n}$. System performance often depends on the number of parallel instances with the heaviest load [20], thus we define load imbalance degree $LI^w_{D(e_{v_i,v_j})}$ as a metric to quantify the load balancing across the set of downstream instances $D(e_{v_i,v_j})$ within window ds^w after being grouped by the stream grouping function $G(ds^w_{U(e_{v_i,v_j})})$. It can be described by (4).

$$LI^w_{D(e_{v_i,v_j})} = \frac{\max\limits_{v_{j,n}\in D(e_{v_i,v_j})}\left(L^w_{v_{j,n}}\right) - \operatorname{avg}\limits_{v_{j,n}\in D(e_{v_i,v_j})}\left(L^w_{v_{j,n}}\right)}{\operatorname{avg}\limits_{v_{j,n}\in D(e_{v_i,v_j})}\left(L^w_{v_{j,n}}\right)}, \tag{4}$$

where $\max\left(L^w_{v_{j,n}}\right)$ is the maximum load of downstream instances $D(e_{v_i,v_j})$ within window ds^w, and $\operatorname{avg}\left(L^w_{v_{j,n}}\right)$ is the average load of downstream instances $D(e_{v_i,v_j})$ within window ds^w. The lower the imbalance degree $LI^w_{D(e_{v_i,v_j})}$, the better the load balance of $D(e_{v_i,v_j})$.

The load balancing optimization problem for stream grouping can be formalized as (5):

$$\min\left(LI^w_{D(e_{v_i,v_j})}\right), \tag{5}$$

subject to

$$0 \le L^w_{v_{j,n}} \le \max\left(L^w_{v_{j,n}}\right), v_{j,n} \in D(e_{v_i,v_j}). \tag{6}$$

As shown in Figure 1, the maximum load $\max\left(L^w_{v_{j,n}}\right)$ among downstream instances $D(e_{v_i,v_j})$ is 4 (as $n^w_{v_{j,1}}(dt) = 4$), while the average load $avg(L^w_{v_{j,n}})$ among downstream instances $D(e_{v_i,v_j})$ is 3. This results in a load imbalance degree $LI^w_{D(e_{v_i,v_j})}$ of 0.33. A load imbalance degree $LI^w_{D(e_{v_i,v_j})}$ of 0 signifies perfect balance across downstream instances $D(e_{v_i,v_j})$ within window ds^w.

B. Key splitting optimization

Key splitting technique can effectively achieve load balancing without state migration, but it also incurs additional costs. These costs consist of two parts: (1) Memory cost. The stream grouping function assigns tuples with identical keys to multiple instances of an operator, and each instance may maintain partial results as states for various keys, requiring memory to store these partial states. (2) Aggregation cost. The partial states must be aggregated in the downstream instances, leading to resource requirements during the aggregation process. These costs are proportional to the degree of key splitting. Our objective is to minimize the extent of key splitting as much as possible, while still benefiting from the absence of state migration.

We define replication factor $RF^w_{D(e_{v_i,v_j})}$ as a metric to quantify the extent of key splitting across the set of downstream instances $D(e_{v_i,v_j})$ within window ds^w after being grouped by the stream grouping function $G(ds^w_{U(e_{v_i,v_j})})$. It can be described by (7).

$$RF^w_{D(e_{v_i,v_j})} = \frac{\sum_{n=1}^{n_{D(e_{v_i,v_j})}} n^w_{v_{j,n}}(k)}{n^w_{D(e_{v_i,v_j})}(k)}, \tag{7}$$

where $n^w_{v_{j,n}}(k)$ is the number of distinct keys within window ds^w assigned from upstream instances $U(e_{v_i,v_j})$ to downstream instance $v_{j,n}$, and $n^w_{D(e_{v_i,v_j})}(k)$ is the total number of distinct keys within window ds^w assigned from upstream instances $U(e_{v_i,v_j})$ to downstream instances $D(e_{v_i,v_j})$. The lower the replication factor $RF^w_{D(e_{v_i,v_j})}$, the better the degree of key splitting for $D(e_{v_i,v_j})$.

The key splitting optimization problem for stream grouping can be formalized as (8):

$$\min\left(RF^w_{D(e_{v_i,v_j})}\right), \tag{8}$$

subject to

$$0 \le n^w_{v_{j,n}}(k) \le n^w_{D(e_{v_i,v_j})}(k), v_{j,n} \in D(e_{v_i,v_j}). \tag{9}$$

As shown in Figure 1, the sum of number of distinct keys $n^w_{v_{j,n}}(k)$ that each downstream instance $v_{j,n}$ receives is 6, while the total number of distinct keys $n^w_{D(e_{v_i,v_j})}(k)$ sent to downstream instances $D(e_{v_i,v_j})$ is 3. This leads to a replication factor $RF^w_{D(e_{v_i,v_j})}$ of 2 for downstream instances $D(e_{v_i,v_j})$, indicating that each key is split an average of two times. A replication factor $RF^w_{D(e_{v_i,v_j})}$ of 1 indicates that no key splitting has occurred across downstream instances $D(e_{v_i,v_j})$ within window ds^w. As we need to split the hot keys in skewed data streams for load balancing, our optimization objective is to keep the replication factor as close to 1 as possible.

V. Pd-Stream: Architecture and Algorithms

Based on the above analysis, we propose Pd-Stream, a high-balance, low-cost stream grouping strategy for skewed data streams. In this section, we first provide an overview of Pd-Stream's architecture, followed by detailed descriptions of its key components and algorithms.

A. System architecture

Built on Apache Storm platform, Pd-Stream is mainly composed of Nimbus, Zookeeper and Supervisor. Nimbus serves as the primary scheduler, assigning the tasks of a user-submitted topology to available compute nodes. For cluster coordination, Zookeeper maintains distributed state information, including topology configurations and task assignments. It also monitors the liveness of worker nodes through a heartbeat mechanism, supplying Nimbus with the necessary information to dynamically supervise the cluster and manage task placements. Supervisor starts or stops the workers to Nimbus's instructions. Worker nodes are managed by Supervisors, with each Worker containing multiple Executors. Each instance of a vertex in the stream application's topology corresponds to a stream application task, and each task is executed by an Executor.

The Grouper proposed by Pd-Stream can be customized by implementing the `CustomStreamGrouping` interface [33]. Grouper operates within each Executor, distributing data streams to the Executors responsible for downstream tasks according to the stream application topology. If the Executor fails, the Grouper's state will be reinitialized along with the deployment of a new Executor. As shown in Figure 2, the Grouper consists of three components: a sliding sampling window, hot key probability estimation, and instance assignment.

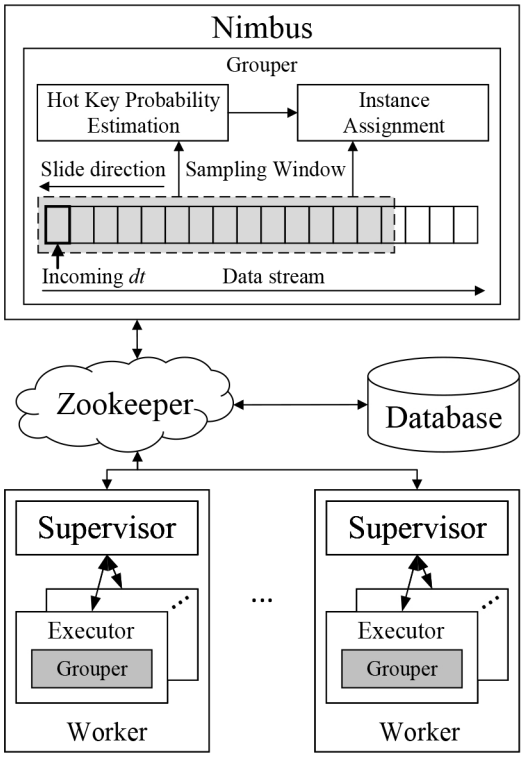


Fig. 2. Pd-Stream architecture.

The **sliding window** samples the key in each incoming tuple. Its window length is determined based on a predefined hot key probability threshold. Using this threshold, we can calculate the minimum window length required to sample hot keys in real time, ensuring that all current hot keys are sampled while minimizing the memory overhead caused by the sampling window. The sliding window moves along with the input data stream, allowing for real-time updates on hot keys and their associated probabilities.

The **hot key probability estimation** component estimates the probabilities of hot keys based on their occurrence popularity within the sliding window. In this component, we employ a binary search algorithm to estimate the probability of each hot key. To reduce the computational cost of the search process and minimize grouping latency, we maintain a probability table that records the mapping between hot key popularity and their corresponding probability. This table is generated during the Grouper initialization phase.

The **instance assignment** component selects an appropriate target downstream instance for each incoming data tuple through routing table generated by the hot key probability estimations. In this component, we first calculate the required number of candidate instances based on the real-time hot key probability and determine a candidate set. This approach allows for the selection of target instances from a subset of downstream instances rather than from all downstream instances, thereby reducing time costs. If the incoming tuple does not contain a hot key, the candidate set is determined

using two hash functions [17] based on the non-hot key value. Otherwise, the candidate set is retrieved from a routing table, which records the mapping between instances and hot keys. If the required number of candidate instances exceeds the number of candidate instances recorded in the table for the hot key, we add the downstream instance with the lowest load to the candidate set. The instance with the lowest load from the candidate set is then selected as the target instance for the incoming tuple. Subsequently, the candidate set is updated in the table. As the sliding window advances, hot keys that fall outside the window's scope are removed from the routing table to minimize memory cost. Details of this instance assignment process are explained in Section V-D.

B. Sliding sampling window

To cope with the dynamic nature of data streams, we use the sliding window as a forgetting mechanism, keeping the frequency estimation aligned with current data distribution. As we continuously sample the key of each incoming tuple within a sampling window, hot keys appear multiple times. This allows us to approximate their probabilities based on popularity. As the data stream progresses, the sampling window slides forward, with outdated data discarded and new data incorporated, enabling the frequency estimates to remain sensitive to recent trends. In this way, we can timely identify emerging hot items and react promptly to short-term bursts by maintaining a focus on recent activity. Selecting an optimal window length is crucial for accurately estimating these probabilities. This window must be long enough to capture sufficient occurrences of hot keys, yet not so long as it incurs substantial memory overhead. Below, we outline the process for determining the appropriate window length.

The window length N can be determined based on the probability threshold θ of hot keys. To ensure that a hot key k is sampled with a probability of at least θ, the likelihood of it appearing at least once in N consecutive samples should be maximized. This implies that the probability of it not appearing in N samples must be minimized. Let X_k represent the appearance popularity of hot key k within a window of length N. The probability $P_k(X_k = 0)$ of it not appearing within the window of length N is calculated by (10).

$$P_k(X_k = 0) = (1 - p_k)^N \leq (1 - \theta)^N, \tag{10}$$

where p_k is the probability of key k appearing. We set $\theta = \frac{2}{n_{D(e_{v_i,v_j})}}$ based on PKG [17], and define $N = \alpha \times \frac{n_{D(e_{v_i,v_j})}}{2}$, where $n_{D(e_{v_i,v_j})}$ is the number of downstream instances $D(e_{v_i,v_j})$ and α is the length coefficient to be determined. The upper bound of probability $P_k(X_k = 0)$ can be then calculated by (11).

$$P_k(X_k = 0) \leq \left[(1 - \theta)^{\frac{1}{\theta}}\right]^{\alpha}, \tag{11}$$

The limit of $(1 - \theta)^{\frac{1}{\theta}}$ is calculated by (12).

$$\lim_{n_{D(e_{v_i,v_j})} \to \infty} (1 - \theta)^{\frac{1}{\theta}} = \frac{1}{e}. \tag{12}$$

The limit of the upper bound of $P_k(X_k = 0)$ is calculated by (13).

$$P_k(X_k = 0) < \left(\frac{1}{e}\right)^{\alpha}. \tag{13}$$

Regardless of the number of downstream instances, the upper bound of $P_k(X_k = 0)$ will not exceed $(\frac{1}{e})^{\alpha}$.

Next, we determine the value of α based on the limit of the upper bound of $P_k(X_k = 0)$. α determines the sampling window length, which ensures that the window is sufficiently large to capture enough occurrences of hot keys. For simplicity in calculations, we default α to be an integer.

As illustrated in Table II, the upper bound of $P_k(X_k = 0)$ decreases markedly with an increase in α. This reduction becomes more gradual upon reaching $\alpha = 3$, and it approaches a value close to zero when $\alpha = 6$.

TABLE II
LIMIT OF UPPER BOUND OF $P_k(X_k = 0)$ UNDER DIFFERENT α.

α	1	2	3	**4**	5	6	...
$\left(\frac{1}{e}\right)^{\alpha}$	36.78%	13.53%	4.97%	**1.83%**	0.67%	0.24%	...

A lower upper bound correlates with a higher likelihood of sampling all hot keys. However, this advantage is counterbalanced by an increase in memory overhead caused by the longer sampling window. To optimize both for a minimal $P_k(X_k = 0)$ and reduced memory overhead, we set $\alpha = 4$. When $\alpha = 4$, $P_k(X_k = 0) < (\frac{1}{e})^4$, meaning there is more than 98.16% probability that any hot key will appear at least once in the sampling window. Consequently, we set the sampling window length to $N = 2n_{D(e_{v_i,v_j})}$.

We also need to determine the sliding step size of the sampling window. By default, we set the step size to 1 to ensure that hot keys and their probabilities are updated in real time. This step size enhances the accuracy of subsequent calculations for the number of hot key candidate instances, thereby maintaining the real-time performance of the grouping strategy. If the step size is set to 2 or another value, it could result in delayed updates to the hot key probabilities. As a result, when the probability of a hot key suddenly increases, the number of hot key candidate instances might not increase accordingly, leading to a brief load imbalance in downstream instances. However, this step size of 1 also allows keys that appear only once to enter the sampling window. To address this, we consider such keys that appear only once in the sampling window as non-hot keys. The method for handling these non-hot keys is discussed in detail in Section V-D.

C. Hot key probability estimation

Upon determining the length and step size of the sampling window, the next step involves estimating the probabilities of hot keys. Given that each key behaves independently, their occurrences or absence adhere to a binomial distribution [34]. For a hot key k, the probability $P_k(X_k = n_k)$ of it appearing n_k times within a sampling window of length N can be calculated by (14).

$$P_k(X_k = n_k) = \binom{N}{n_k} \times {p_k}^{n_k} \times (1 - p_k)^{N - n_k}. \tag{14}$$

This equation represents the probability mass function (PMF) of a binomial distribution. $\binom{N}{n_k}$ is the binomial coefficient, which represents the number of ways to choose n_k occurrences of hot key k out of N samples. ${p_k}^{n_k}$ is the probability of hot key k appearing n_k times, with p_k being the probability of a single appearance of hot key k. $(1-p_k)^{N-n_k}$ is the probability of hot key k not appearing in the remaining $N-n_k$ samples, with $(1-p_k)$ being the probability of hot key k not appearing in a single sample.

The probability $P_k(X_k \geq n_k)$ of the hot key k appearing at least n_k times in the sampling window of length N can be calculated by (15).

$$P_k(X_k \geq n_k) = \sum_{i=n_k}^{N} P(X_k = i). \tag{15}$$

This equation represents the cumulative distribution function (CDF) of a binomial distribution. Due to the lack of a closed-form solution for $P_k^{-1}(X_k \geq n_k)$, we adopt a binary search method to estimate an appropriate hot key probability $\hat{p_k}$ for the hot key k. Algorithm 1 shows the details of this process.

Algorithm 1: Hot key probability search

Input: Number of occurrences n_k of hot key k in the sampling window, expected probability $\hat{P}_k(X_k \geq n_k)$ of hot key k appearing at least n_k times in the sampling window, acceptable margin of search error ϵ.

Output: Estimated probability $\hat{p_k}$ of hot key k.

```
Initialize p_lower ← 0, p_upper ← 1;
while p_upper − p_lower > ϵ do
    /* Calculate the midpoint */
    p_mid ← (p_upper + p_lower)/2;
    /* Calculate the probability of hot key k appearing at least n_k times */
    Calculate P_k(X_k ≥ n_k) by p_mid according to (14) and (15).;
    /* Update the midpoint based on the calculated probability */
    if P_k(X_k ≥ n_k) < P̂_k(X_k ≥ n_k) then
        p_lower ← p_mid.;
    else
        p_upper ← p_mid.;
    end
end
p̂_k ← p_mid;
return p̂_k
```

The input of this algorithm includes the occurrence number n_k of hot key k appearing in the sampling window, the expected probability $\hat{P}_k(X_k \geq n_k)$ of hot key k appearing at least n_{k_i} times in the sampling window, and the acceptable margin of search error ϵ. The output is the estimated probability $\hat{p_k}$ of hot key k.

Step 1 sets the initial search range, with the lower bound p_{lower} set to 0, and the upper bound p_{upper} set to 1. Steps 2 to 10 iteratively conduct a binary search, substituting the midpoint p_{mid} (p_k) in (14) and (15) to compute the probability $P_k(X_k \geq n_k)$ of key k appearing at least n_k times in the sampling window. One of the lower bound p_{lower} and upper bound p_{upper} is updated to p_{mid} by comparing $P_k(X_k \geq n_k)$ with the expected probability $\hat{P}_k(X_k \geq n_k)$. The search process concludes when the interval between p_{lower} and p_{upper} narrows to less than the acceptable margin of search error ϵ. Step 11 sets the final search result p_{mid} as the estimated probability $\hat{p_k}$ of the hot key k. The time complexity of Algorithm 1 is $O(log_2(1/\epsilon))$.

Using Algorithm 1, a probability table mapping n_k to $\hat{p_k}$ can be constructed based on the number of downstream instances. Table III illustrates an example of such a probability table, where the window length $N = 16$, the maximum expected probability $\hat{P}_k(X_k \geq n) = 99.9\%$. The expected probability primarily determines the number of candidate instances for each hot key. The higher the expected probability, the larger the number of candidate instances. We set the acceptable margin of search error ϵ to 10^{-4}, which is the termination condition of Algorithm 1, ensuring that the upper and lower bounds (p_{lower} and p_{upper}) converge. The occurrence number n_k of key k appearing in the sampling window is a variable that can range from 1 to 16. For each value of n_k, we can obtain an estimated probability $\hat{p_k}$ by applying Algorithm 1.

TABLE III
AN EXAMPLE OF PROBABILITY TABLE WITH $N = 16$, $\hat{P}_k(X_k \geq n) = 99\%$, AND $\epsilon = 10^{-4}$.

n_k	1	2	3	4	5	6	7	8
$\hat{p_k}$	25.0%	34.9%	43.1%	50.3%	56.9%	63.0%	68.7%	73.9%
n_k	9	10	11	12	13	14	15	16
$\hat{p_k}$	78.8%	83.4%	87.5%	91.2%	94.5%	97.1%	99.0%	99.9%

The probability table is integrated into the hot key probability estimation component and constructed during the Grouper initialization phase. Each Grouper has its own hot key probability table. During the execution of the application topology, we can directly obtain the key's probability for each incoming tuple based on their popularity within the sampling window, thus avoiding the computational cost caused by searching the key probability by Algorithm 1.

D. Instance assignment

To choose an appropriate target instance $v_{j,target}$ from the downstream instances $D(e_{v_i,v_j})$ for an incoming data tuple dt, we first need to determine the number of candidate instances. According to Algorithm 1, the estimated number of candidate instances $\hat{n}_{C(D(e_{v_i,v_j}))_k}$ for the incoming data tuple dt with key k can be calculated by (16).

$$\hat{n}_{C(D(e_{v_i,v_j}))_k} = \lfloor \hat{p_k} \times n_{D(e_{v_i,v_j})} \rfloor, \tag{16}$$

where $C(D(e_{v_i,v_j}))_k$ is the set of candidate instances for the incoming data tuple dt with key k, $\hat{p_k}$ is the estimated probability of key k derived from the probability table, and

$n_{D(e_{v_i,v_j})}$ is the number of downstream instances $D(e_{v_i,v_j})$. Since the number of candidate instances must be an integer, we use flooring to get an estimated number of candidate instances $\hat{n}_{C(D(e_{v_i,v_j}))_k}$.

Next, we utilize a heuristic approach to choose an appropriate target instance $v_{j,target}$ for the incoming data tuple dt. Algorithm 2 outlines the details of this process.

Algorithm 2: Instance assignment

Input: Key k of the incoming data tuple dt.
Output: Target instance $v_{j,target}$ for dt.

1 Get number of occurrences n_k of key k in the sampling window;
/* Get the candidate instances for the incoming data tuple */
2 **if** $n_k = 1$ **then**
/* Non-hot key, get candidate instances using two hash functions */
3 $C(D(e_{v_i,v_j}))_k \leftarrow \{v_{j,H_1(k)}, v_{j,H_2(k)}\}$;
4 **end**
5 **else**
6 Retrieve routing table entry e_k for key k;
7 **if** e_k *is null* **then**
8 Create a new entry e_k for key k;
9 Add instances $v_{j,H_1(k)}$ and $v_{j,H_2(k)}$ to the new entry e_k using hash functions H_1 and H_2;
10 **end**
11 Retrieve the candidate instances $C(D(e_{v_i,v_j}))_k$ from entry e_k, and get its size $n_{C(D(e_{v_i,v_j}))_k}$;
12 Calculate the estimated number of candidate instances $\hat{n}_{C(D(e_{v_i,v_j}))_k}$ by (16);
13 **if** $\hat{n}_{C(D(e_{v_i,v_j}))_k} > n_{C(D(e_{v_i,v_j}))_k}$ **then**
14 Add downstream instance with lowest load from downstream instances $D(e_{v_i,v_j})$ to candidate instances $C(D(e_{v_i,v_j}))_k$;
15 **end**
16 **end**
/* Choose the target instance */
17 $v_{j,target} \leftarrow$ instance with lowest load in $C(D(e_{v_i,v_j}))_k$;
/* Update the sampling window and routing table */
18 Add k to the head of the sampling window;
19 Remove the last key from tail of the sampling window;
20 **if** *the sampling window does not contain the tail key* ***and*** *the entry for tail key is not null* **then**
21 Remove entry for tail key from routing table;
22 **end**
23 **return** $v_{j,target}$

The input of this algorithm is the key k of the incoming tuple dt. The output is the target instance $v_{j,target}$ for tuple dt. Steps 1 to 15 select candidate downstream instances $C(D(e_{v_i,v_j}))_k$ from the downstream instances $D(e_{v_i,v_j})$ for the incoming data tuple dt with key k. We determine whether a key k is a hot key based on the number of its occurrences n_k in the sampling window.

If n_k equals to 1, we consider key k as a non-hot key (Steps 2 to 3). Similar to PKG, we directly select candidate instances $C(D(e_{v_i,v_j}))_k$ using two hash functions, H_1 and H_2. If key k is potentially a hot key, the number of occurrences n_k increases as corresponding data tuples arrive, allowing us to pre-map key k to two downstream instances. Since we directly select candidate instances for non-hot keys through hash functions, there is no need to record them in the routing table, thereby reducing the memory cost of the table.

If the number of occurrences n_k is greater than or equal to 2, we consider key k to be a hot key (Steps 4 to 15). We then retrieve the candidate instances $C(D(e_{v_i,v_j}))_k$ from entry k in the routing table. To balance load, each time the routing table is updated, we only select the instance with the lowest load from the downstream instances $D(e_{v_i,v_j})$. If the instance with the lowest load is already recorded in the candidate instances $C(D(e_{v_i,v_j}))_k$, there is no need to add additional instance from the downstream instances $D(e_{v_i,v_j})$, thereby minimizing the number of key splitting. The number of candidate instances $n_{C(D(e_{v_i,v_j}))_k}$ is the minimum value that satisfies load balancing, effectively minimizing key splitting while ensuring balanced load distribution.

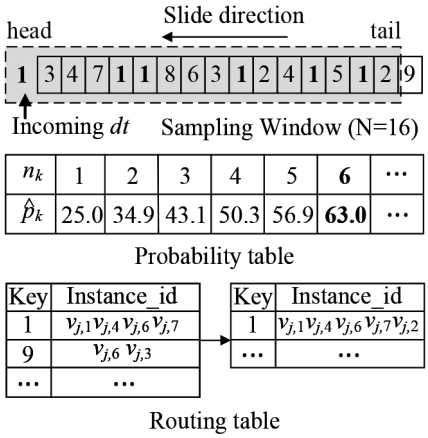
Step 16 selects the instance with the lowest load from the candidate instances $C(D(e_{v_i,v_j}))_k$ as the target downstream instance $v_{j,target}$. The target instance is chosen from the candidate instances $C(D(e_{v_i,v_j}))_k$ rather than from all downstream instances $D(e_{v_i,v_j})$, which reduces the algorithm's latency.

Steps 17 to 21 update the sampling window and routing table. As data streams vary, new hot keys emerge, and the previous hot keys may become non-hot keys. If outdated hot keys entries in the routing table are not deleted promptly, they can consume significant memory resources. Therefore, we only record the hot keys present in the sampling window in the routing table. As the sampling window slides, entries for outdated hot keys that are no longer present in the sampling window are promptly deleted, ensuring a low memory cost for the routing table.

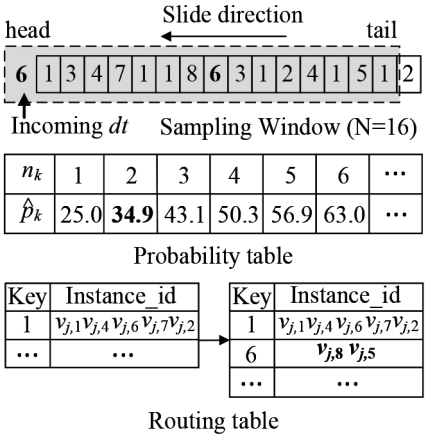
Algorithm 2 requires traversing the entire sampling window to obtain the number of occurrences n_k for key k in the window, as well as traversing the candidate instances $C(D(e_{v_i,v_j}))_k$ to select the instance with the lowest load [18]. The time complexity of this algorithm is $O(N + n_{C(D(e_{v_i,v_j}))_k})$, where N is the length of the sampling window, and $n_{C(D(e_{v_i,v_j}))_k}$ is the number of candidate instances for the incoming data tuple dt with key k.

Figure 3 depicts an example of routing table being updated as the sampling window slides. The dashed boxes in the figures represent sliding windows, and each number represents a distinct key. This example demonstrates three scenarios of routing table updates resulting from sliding the sampling window twice. The length of the sampling window N is 16, and the probability table used for routing table updates is presented in Table III. For convenience, this table is also replicated in Figure 3.

In Fig. 3(a), as the sampling window slides, the first key to enter is key 1, which now appears 6 times, i.e., $n_k = 6$. Based



(a) First update to routing table: add a new instance $v_{j,2}$ to entry e_1, and delete the old entry e_9.



(b) Second update to routing table: create a new entry e_6.

Fig. 3. An example of routing table being updated as the sampling window slides.

on the probability table, we derive its estimated probability $\hat{p_1}$ to be 62.99%. We calculate the estimated number of candidate instances $\hat{n}_{C(D(e_{v_i,v_j}))_1}$ to be 5 by (16), surpassing the number of candidate instances $n_{C(D(e_{v_i,v_j}))_1}$ (4) in the routing table. Consequently, we select the instance with the lowest load (e.g., $v_{j,2}$) from the downstream instances $D(e_{v_i,v_j})$ and integrate it into entry e_1. As the sampling window advances, key 9 is no longer within the sampling window, so we remove entry e_9 from the routing table.

In Fig. 3(b), the second incoming key is key 6, which is not present in the routing table (keys that appear only once are considered non-hot keys). Since key 6 now appears twice in the sampling window, we categorize it as a hot key. Subsequently, we establish a new entry e_6 for key 6 and include two instances (e.g., $v_{j,8}$ and $v_{j,5}$), determined by the two distinct hash functions, H_1 and H_2. The new entry e_6 is then appended to the routing table.

VI. Performance Evaluation

We evaluate the proposed Pd-Stream through comparative experiments. Before analyzing the results, we first discuss the experimental environment and parameter settings.

A. Experimental environment

The cluster comprises 50 compute nodes, with one serving as the master node to host the Nimbus, while the remaining nodes run Supervisors. The experimental setup utilizes Ubuntu 20.04.4 as the operating system and Storm 2.1.0 as the stream computing system.

1) Datasets. The experimental datasets comprise both synthetic and real-world datasets. The synthetic datasets follow a Zipf distribution [20] with coefficients z ranging from 1.0 to 2.0. A higher coefficient indicates a greater degree of skewness within the dataset. There are two real-world datasets. The first contains a set of DDoS flows extracted from public IDS datasets [35]. These flows include various features such as timestamps, source and destination IPs, and more. We use the source IP as the key. The second consists of hashtags extracted from tweets related to the novel coronavirus (COVID-19) from January to March 2022 [36]. We use the hashtag as the key. Tables IV and V summarize the statistics of the synthetic datasets with different Zipf coefficients and the real datasets, respectively. The statistics include the number of messages, the number of keys, and the probability of the most frequent key ($p_{1^{st}}$).

TABLE IV
SUMMARY OF THE SYNTHETIC DATASETS.

Zipf Coeff.	No. of Messages	No. of Keys	$p_{1^{st}}(\%)$
1.0	10M	1.9M	5.9%
1.2	10M	563K	18.5%
1.4	10M	127K	32.2%
1.6	10M	33.1K	43.7%
1.8	10M	10.7K	53.2%
2.0	10M	4.3K	60.7%

TABLE V
SUMMARY OF THE REAL-WORLD DATASETS.

Dataset	Symbol	No. of Messages	No. of Keys	$p_{1^{st}}(\%)$
DDoS	DS	12M	2.4M	20.2%
Hashtags	HT	67M	2.2M	8.5%

2) Streaming applications. We use the streaming application provided by the Benchmark [7] to evaluate the system performance. The parallelism and function of each vertex in the streaming application are presented in Table VI.

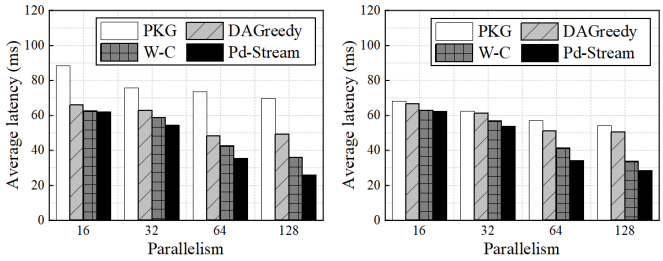
3) Baseline schemes. Pd-Stream is compared against three grouping strategies: Partial Key Grouping (PKG), W-Choices grouping [18] (W-C), and Distribution-Aware Greedy Grouping [22] (DAGreedy). These three grouping strategies represent the evolution of key splitting-based grouping strategies, progressing from classical load balancing to comprehensive state-of-the-art grouping optimization. Among them, PKG is the first strategy to adopt key splitting technique. W-C achieves the lowest load imbalance, while DAGreedy is the first holistic strategy to consider both load imbalance and replication factor.

B. System latency

Latency is an important metric for evaluating system performance. The average latency reflects the average time required for the system to process a tuple.

TABLE VI
VERTEX FUNCTION OF THE EXPERIMENTAL APPLICATION.

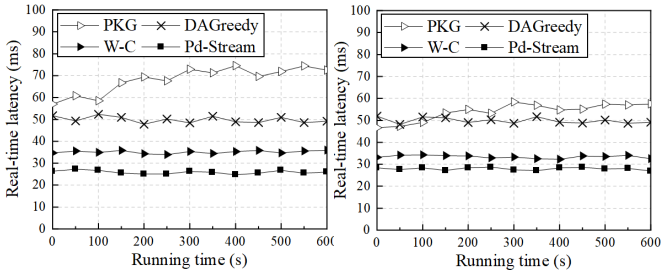
Vertex	Parallelism	Function
v_{read}	8	Read tuples from data stream
$v_{deserialize}$	8	Convert the string to structured data
v_{filter}	8	Filter tuples
$v_{project}$	8	Extract the identifier and timestamp
v_{join}	16,32,64,128	Join tuple identifiers
$v_{aggregate}$	4	Aggregate results from upstream



(a) Average latency on DS. (b) Average latency on HT.

Fig. 4. Average latency comparison on real-world datasets under varying levels of parallelism.

As shown in Fig. 4, the average latency for different grouping strategies under various parallelism are compared on the real-world datasets DS and HT, respectively. As the level of parallelism increases, Pd-Stream's system latency significantly decreases. For dataset DS, Pd-Stream's average system latency is 61.92 ms at a parallelism of 16 and 26.11 ms at a parallelism of 128, representing a reduction of 57.82%. At a parallelism of 128, Pd-Stream reduces latency by 27.16% and 46.92% compared to W-C and DAGreedy, respectively. For dataset HT, Pd-Stream's average system latency is 62.33 ms at a parallelism of 16 and 28.64 ms at a parallelism of 128, representing a reduction of 54.05%. At a parallelism of 128, Pd-Stream reduces latency by 14.67% and 42.97% compared to W-C and DAGreedy, respectively. Pd-Stream's low system latency is primarily due to its effective load balancing, which allows stream applications to fully utilize the increased computational resources provided by higher parallelism, significantly reducing data waiting time. Moreover, Pd-Stream strictly controls key splitting, which reduces data aggregation latency. Its inherent efficiency also contributes to the reduction of grouping latency.



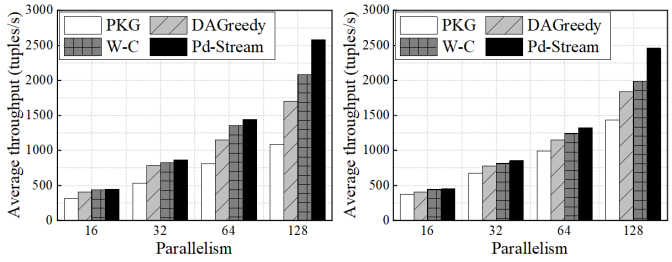
(a) Real-time latency on DS. (b) Real-time latency on HT.

Fig. 5. Real-time latency comparison on real-world datasets.

As shown in Fig. 5, the real-time system latency under different grouping strategies is presented for the real-world datasets DS and HT, with the parallelism level set to 128. It can be observed that the system latency of the Pd-Stream is significantly lower than that of PKG, DAGreedy, and W-C. On dataset DS, the system latency of Pd-Stream remains consistently around 26 ms, while on dataset HT, it fluctuates slightly around 28 ms. The low and stable latency of the Pd-Stream indicates that its hot-key probability estimation algorithm can effectively identify real-time hot keys. Combined with the dynamic routing allocation mechanism, it distributes hot-key tuples appropriately to downstream instances, thereby achieving efficient load balancing.

C. System throughput

The throughput evaluation metric is the average system throughput, which reflects the average number of output tuples per unit time of the system.

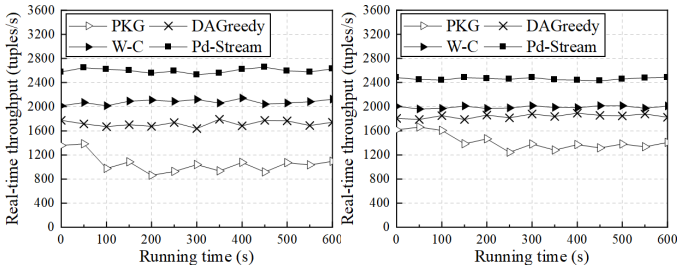


(a) Average throughput on DS. (b) Average throughput on HT.

Fig. 6. Average throughput comparison on real-world datasets under varying levels of parallelism.

As shown in Fig. 6, the average throughput for different grouping strategies under various parallelism are compared on the real-world datasets DS and HT, respectively. As the level of parallelism increases, Pd-Stream's throughput increases significantly. For dataset DS, Pd-Stream's average throughput is 445 tuples/s at a parallelism of 16 and 2582 tuples/s at a parallelism of 128. At a parallelism of 128, Pd-Stream's throughput is 23.89% and 52.06% higher than W-C and DAGreedy, respectively. For dataset HT, Pd-Stream's average throughput is 448 tuples/s at a parallelism of 16 and 2459 tuples/s at a parallelism of 128. At a parallelism of 128, Pd-Stream's throughput is 23.69% and 33.78% higher than W-C and DAGreedy, respectively. The high throughput of Pd-Stream is partly due to its effective load balancing, and partly because Pd-Stream's low key splitting and low grouping latency do not become bottlenecks for system throughput. This allows stream applications to fully utilize the increased computational resources provided by higher parallelism, resulting in a significant increase in throughput.

As shown in Fig. 7, the real-time throughput under different grouping strategies is presented for the real-world datasets DS and HT, with the parallelism level set to 128. It can be observed that regardless of the degree of data skew, Pd-Stream consistently achieves the highest throughput (approximately 2500 tuples/s for DS and around 2400 tuples/s for HT). This advantage primarily stems from two factors: (1) Pd-Stream exhibits strong load balancing capabilities, allowing it to fully utilize the computational resources. (2) Pd-Stream



(a) Real-time throughput on DS. (b) Real-time throughput on HT.

Fig. 7. Real-time throughput comparison on real-world datasets.

effectively controls the degree of key splitting, thereby reducing aggregation overhead. In contrast, although W-C achieves load balancing, it does not control key splitting, which leads to significantly increased aggregation overhead under high parallelism and thus lower throughput compared to Pd-Stream. DAGreedy and PKG, due to their weaker load balancing capabilities, show both lower throughput and less stability than Pd-Stream.

D. Load imbalance

An effective grouping strategy must first ensure load balancing. We assess the load balancing capabilities of grouping strategies using the load imbalance degree described in Section IV. This metric is calculated by tracking the load on each instance of vertex v_{join}.

As illustrated in Figure 8, we compare the load imbalance of different grouping strategies using synthetic datasets with different Zipf coefficients and varying levels of parallelism.

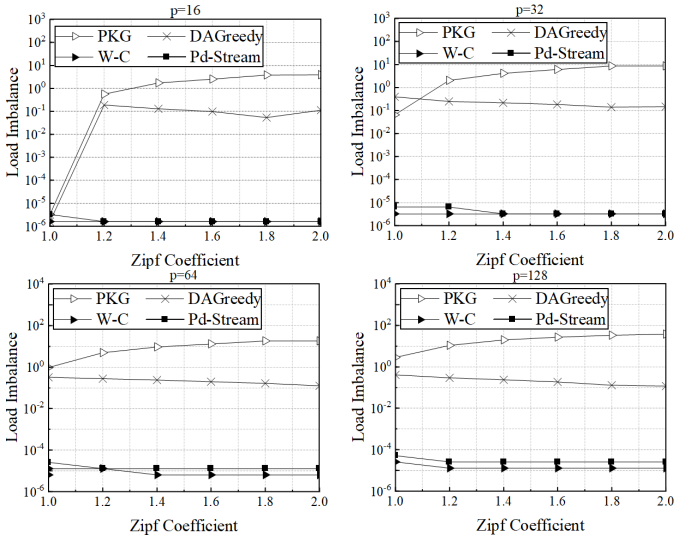
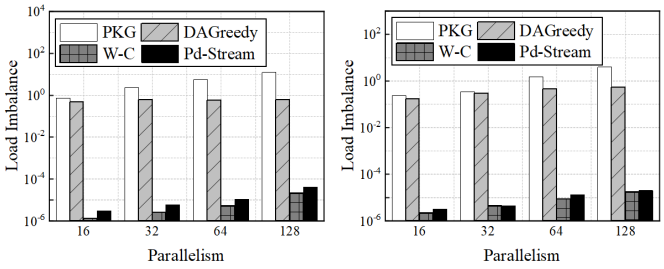


Fig. 8. Load imbalance comparison on synthetic datasets with different Zipf coefficients under varying levels of parallelism p.

Since PKG only selects two downstream instances as candidate target instances, it can maintain low load imbalance only when both the Zipf coefficient and downstream instance parallelism are relatively low (e.g., $z = 1.0$, $p = 16$). However, the load imbalance of PKG significantly increases as the Zipf coefficient and parallelism rise. For example, when the parallelism is 64, the load imbalance of PKG jumps from 0.97 at a Zipf coefficient of 1.0 to 18.45 at a Zipf coefficient of 2.0. Similarly, at a Zipf coefficient of 1.6, the load imbalance of PKG increases from 3.85 at a parallelism of 16 to 27.01 at a parallelism of 128.

In contrast, DAGreedy exhibits significantly lower load imbalance, which is less affected by changes in the Zipf coefficient and parallelism, maintaining within a range of 0.1 to 1.0. At lower Zipf coefficients, DAGreedy tends to limit the extent of key splitting, while at higher Zipf coefficients, DAGreedy tends to balance the load. This results in a downward trend as the Zipf coefficient increases, yet the load imbalance still falls significantly short of the ideal state (0).

W-C and Pd-Stream achieve nearly ideal load imbalance. The load imbalance of W-C and Pd-Stream remains low and is essentially unaffected by increases in the Zipf coefficient and parallelism. At parallelism levels of 16 and 32, the load imbalance of W-C and Pd-Stream stabilizes at 10^{-6}. Even at higher parallelism levels of 64 and 128, the load imbalance of W-C and Pd-Stream only slightly increases, remaining at 10^{-5}. This result demonstrates that Pd-Stream, which selects a subset of downstream instances as candidate target instances, can fully meet the load balancing requirements, matching the performance of W-C, which selects all downstream instances as candidate target instances for hot keys.



(a) Load Imbalance on DS. (b) Load Imbalance on HT.

Fig. 9. Load Imbalance comparison on real-world datasets under varying levels of parallelism.

Figure 9 compares the load imbalance of these grouping algorithms under various levels of parallelism on real datasets (DS and HT). As shown in the figure, the degree of data skew has minimal impact on the load imbalance of Pd-Stream, with only a slow increase as parallelism increases. For dataset DS, the load imbalance of Pd-Stream is 3.09×10^{-6} at a parallelism of 16 and 4.17×10^{-5} at a parallelism of 128. For dataset HT, the load imbalance of Pd-Stream is 3.23×10^{-6} at a parallelism of 16 and 1.98×10^{-5} at a parallelism of 128. The load imbalance of Pd-Stream at different levels of parallelism is significantly lower than that of PKG and DAGreedy, although slightly higher than that of W-C. Nevertheless, it is sufficient to meet the load balancing requirements of real-world application scenarios.

E. Replication factor

While ensuring load balancing, the grouping strategy needs to minimize the extent of key splitting. We measure the extent of key splitting by evaluating the replication factor, as

described in Section IV. The replication factor is determined by tracking the number of distinct keys on each instance of vertex v_{join}.

Figure 10 shows the replication factor of these grouping strategies on synthetic datasets with different Zipf coefficients under varying levels of parallelism.

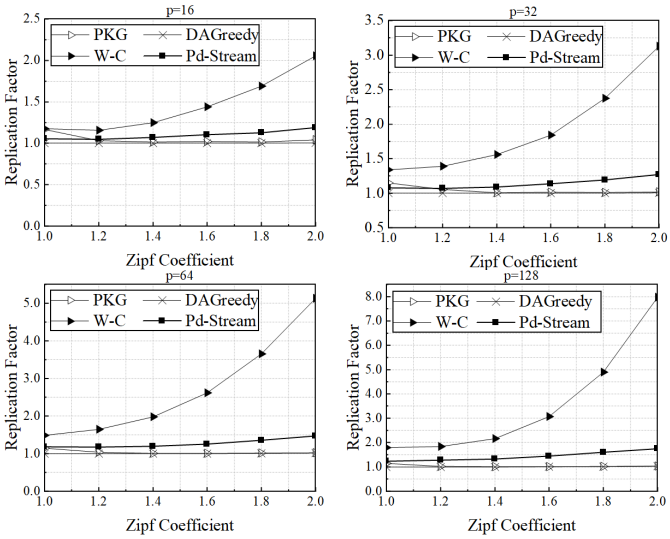
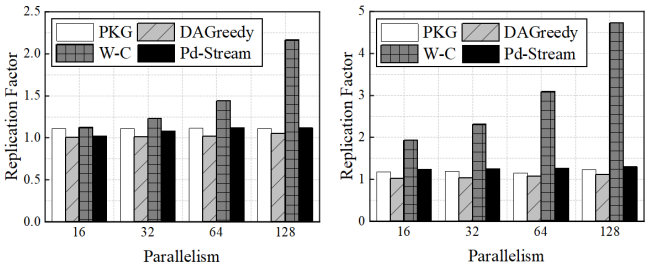


Fig. 10. Replication factor comparison on synthetic datasets with different Zipf coefficients under varying levels of parallelism.

Since PKG maps each key to only two downstream instances, its replication factor is not affected by increases in parallelism and only slightly decreases as the Zipf coefficient increases. As the Zipf coefficient rises from 1.0 to 2.0, PKG's replication factor decreases from approximately 1.15 to around 1.03. Similar to PKG, the replication factor of DAGreedy is essentially unaffected by increases in the Zipf coefficient and parallelism. Due to its strict control over key splitting, DAGreedy's replication factor is even slightly lower than that of PKG, with a maximum of just 1.029 (when $z = 2, p = 128$).

W-C, which selects all downstream instances as candidate instances for hot keys, experiences a significant increase in its replication factor as the Zipf coefficient and parallelism grow. From a Zipf coefficient of 1.0 and parallelism of 16 to a Zipf coefficient of 2.0 and parallelism of 128, its replication factor rises from 1.18 to 7.97, an increase of 5.75 times.

In contrast, Pd-Stream's replication factor increases only gradually with higher Zipf coefficients and parallelism, remaining below 2.0. When the Zipf coefficient is 1.2 and the parallelism is 16, the replication factor of Pd-Stream is 1.05. As the Zipf coefficient increases to 1.4 and the parallelism to 64, the replication factor rises to 1.19. With a Zipf coefficient of 1.8 and parallelism of 64, Pd-Stream's replication factor further increases to 1.35. Even in the extreme case of a Zipf coefficient of 2.0 and parallelism of 128, its replication factor is only 1.74. This is because Pd-Stream controls the number of downstream instances to which hot keys are mapped, with the number of candidate instances determined by the probability of hot keys. Mapping hot keys to this number of downstream instances ensures load balancing without unnecessary key splitting.



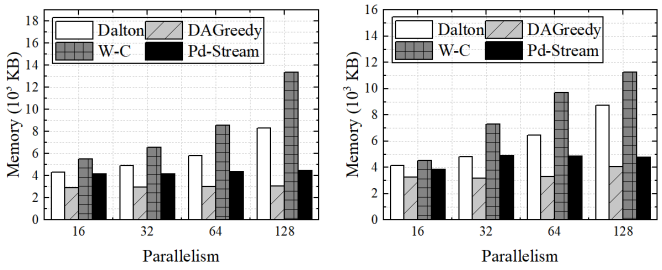
(a) Replication factor on DS. (b) Replication factor on HT.

Fig. 11. Replication factor comparison on real-world datasets under varying levels of parallelism.

Figure 11 compares the replication factor of different grouping algorithms under various levels of parallelism on real datasets. As shown in the figure, the replication factor of Pd-Stream remains almost constant and is not affected by increasing parallelism. For dataset DS, the replication factor of Pd-Stream is 1.24 at a parallelism of 16 and 1.30 at a parallelism of 128, representing an increase of only 4.8%. For dataset HT, the replication factor of Pd-Stream is 1.02 at a parallelism of 16 and 1.12 at a parallelism of 128, representing an increase of only 9.8%. Pd-Stream maintains a low replication factor across different levels of parallelism due to its ability to accurately identify hot keys and map only the hot keys to multiple downstream instances. Additionally, the degree of skew in real data streams is relatively low, with hot keys accounting for only a small portion. For most non-hot keys, Pd-Stream directly employs PKG for processing.

F. Grouping costs

To achieve load balancing, different partitioning strategies commonly introduce varying degrees of key splitting to mitigate skewed loads. However, key splitting incurs additional partitioning costs, which primarily include memory consumption and grouping latency.



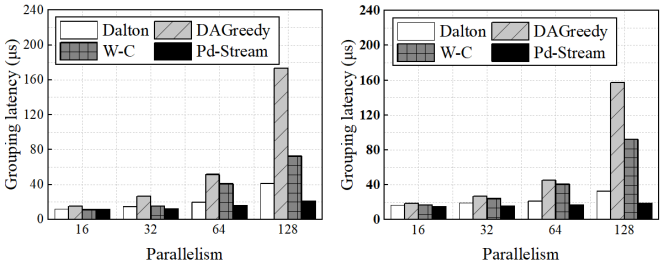
(a) Memory consumption on DS. (b) Memory consumption on HT.

Fig. 12. Memory consumption on real-world datasets under varying levels of parallelism.

As shown in Fig. 12, the memory consumption for different grouping strategies under various levels of parallelism is compared on the real-world datasets DS and HT. The results show that W-C exhibits the highest memory consumption, growing non-linearly to 13,342 KB (DS) and 11,242 KB (HT) at 128 parallelism. Dalton increases from about 4,330 KB to 8,291 KB on DS and from 4,130 KB to 8,718 KB on HT as parallelism increases from 16 to 128. DAGreedy remains the most

memory-efficient, with usage staying around 2,932–3,054 KB (DS) and 3,232–4,054 KB (HT) across parallelism levels. Pd-Stream shows moderate memory consumption, reaching approximately 4,457 KB (DS) and 4,757 KB (HT) at 128 parallelism.

The observed differences in memory overhead stem from how each strategy handles hot keys and state information. W-C incurs the highest overhead by broadcasting keys, which causes severe data redundancy across worker states. Dalton randomly assigns keys to multiple instances during early learning, causing state replication and higher memory usage. DAGreedy minimizes memory use by computing near-optimal, stable assignments for keys and thus keeps per-key state compact, albeit with higher grouping latency. Pd-Stream offers a balanced trade-off: it avoids the extremes of W-C's redundancy and Dalton's exploration overhead, accepting moderate memory usage to achieve lower latency and better overall system performance.



(a) Grouping latency on DS. (b) Grouping latency on HT.

Fig. 13. Grouping latency comparison on real-world datasets under varying levels of parallelism.

As shown in Fig. 13, the grouping latency for different grouping strategies under various levels of parallelism is compared on the real-world datasets DS and HT. As the level of parallelism increases, Pd-Stream consistently maintains low grouping latency. For dataset DS, Pd-Stream's grouping latency is 11.72 µs at a parallelism of 16 and 21.26 µs at a parallelism of 128, representing reductions of 48.78%, 70.66% and 87.73% compared to Dalton, W-C and DAGreedy, respectively. For dataset HT, Pd-Stream's grouping latency is 14.73 µs at a parallelism of 16 and 19.13 µs at a parallelism of 128, representing reductions of 40.62%, 79.23% and 87.81% compared to Dalton, W-C and DAGreedy, respectively. The low grouping latency of Pd-Stream is due to its ability to efficiently select an appropriate number of candidate downstream instances based on key probability, allowing for quick selection of the target instance.

VII. Conclusion and Future Work

When handling skewed data streams, key splitting-based stream grouping strategies achieve load balancing but incur additional key aggregation costs. To address this issue, we propose Pd-Stream, a lightweight, high-balance, and low-cost data stream grouping strategy for skewed data streams in stateful stream computing environments. This strategy identifies hot keys using a sliding sampling window and calculates real-time hot key probabilities based on the popularity of hot keys appearing within the window. Tuples with hot keys are assigned to the minimum number of downstream instances necessary to satisfy load balancing, based on the hot keys' probabilities, thereby reducing the extent of key splitting.

Experiments have shown that Pd-Stream can ensure load balancing while reducing the degree of key splitting when dealing with highly skewed data streams and highly parallel downstream instances.

Our future work will focus on the following two aspects:

(1) Extending our implementation to other distributed stream computing platforms to further demonstrate the generality of Pd-Stream.

(2) Further developing an adaptive mechanism that incorporates both an α-adjusted window size and a θ-adjusted hot key identification to handle the dynamic skewness of data streams.

References

[1] K. Xiao, S. Yang, F. Li, L. Zhu, X. Chen, and X. Fu, "Making serverless not so cold in edge clouds: A cost-effective online approach," *IEEE Transactions on Mobile Computing*, vol. 23, no. 9, pp. 8789–8802, 2024.

[2] O. Runsewe and N. Samaan, "Cloud resource scaling for time-bounded and unbounded big data streaming applications," *IEEE Transactions on Cloud Computing*, vol. 9, no. 2, pp. 504–517, 2021.

[3] 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.

[4] S. Bora, B. Walker, and M. Fidler, "The tiny-tasks granularity trade-off: Balancing overhead versus performance in parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1128–1144, 2023.

[5] Y. Guo, H. Shan, S. Huang, K. Hwang, J. Fan, and Z. Yu, "Gml: Efficiently auto-tuning flink's configurations via guided machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2921–2935, 2021.

[6] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, p. 507–541, 2024.

[7] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1789–1792.

[8] G. Li, J. Zeng, Z. Peng, Y. Liang, X. Zheng, and T. Wang, "E2ec: Edge-to-edge collaboration for efficient real-time video surveillance inference," *IEEE Transactions on Mobile Computing*, pp. 1–15, 2025.

[9] 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.

[10] S. Chaturvedi, S. Tyagi, and Y. Simmhan, "Cost-effective sharing of streaming dataflows for iot applications," *IEEE Transactions on Cloud Computing*, vol. 9, no. 4, pp. 1391–1407, 2021.

[11] Y. Qing and W. Zheng, " Towards Fine-Grained Scalability for Stateful Stream Processing Systems ," in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, 2025, pp. 3835–3848.

[12] 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.

[13] J. Fang, R. Zhang, T. Z. 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, p. 2223–2240, 2018.

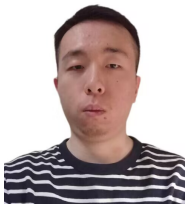
[14] S. Yu, H. Chen, and H. Jin, "Nereus: A distributed stream band join system with adaptive range partitioning," *IEEE Transactions on Consumer Electronics*, vol. 69, no. 4, pp. 949–961, 2023.

[15] 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.

[16] 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.
[17] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *Proceedings of the 31st International Conference on Data Engineering*, 2015, pp. 137–148.
[18] M. Anis Uddin Nasir, G. De Francisci Morales, N. Kourtellis, and M. Serafini, "When two choices are not enough: Balancing at scale in distributed stream processing," in *Proceedings of the 32nd International Conference on Data Engineering*, 2016, pp. 589–600.
[19] X. Liao, Y. Huang, L. Zheng, and H. Jin, "Efficient time-evolving stream processing at scale," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2165–2178, 2019.
[20] M. Wu, D. Sun, S. Gao, K. Li, and R. Buyya, "Ls-stream: Lightening stragglers in join operators for skewed data stream processing," *IEEE Transactions on Computers*, vol. 74, no. 8, pp. 2841–2855, 2025.
[21] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "A holistic view of stream partitioning costs," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, p. 1286–1297, 2017.
[22] A. Pacaci and M. T. Özsu, "Distribution-aware stream partitioning for distributed stream processing systems," in *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, 2018, p. 1–10.
[23] K. Li, G. Liu, and M. Lu, "A holistic stream partitioning algorithm for distributed stream processing systems," in *Proceedings of the 20th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2019, pp. 202–207.
[24] S. Chen, D. Zuo, and Z. Zhang, "Flexsp: $(1+\beta)$-choice based flexible stream partitioning for stateful operators," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, p. 732–741.
[25] X. Zhang, H. Chen, and F. Hu, "Back propagation grouping: Load balancing at global scale when sources are skewed," in *Proceedings of the 2017 IEEE International Conference on Services Computing*, 2017, pp. 426–433.
[26] 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, nov 2022.
[27] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory*, 2005, p. 398–412.
[28] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.
[29] X. Gou, Y. Zhang, Z. Hu, L. He, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, "A sketch framework for approximate data stream processing in sliding windows," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 5, pp. 4411–4424, 2023.
[30] 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 2018 International Conference on Management of Data*, 2018, pp. 741–756.
[31] P. Jia, P. Wang, J. Zhao, Y. Yuan, J. Tao, and X. Guan, "Loglog filter: Filtering cold items within a large range over high speed data streams," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 804–815.
[32] C.-W. Ching, X. Chen, C. Kim, T. Wang, D. Chen, D. Da Silva, and L. Hu, "Agiledart: An agile and scalable edge stream processing engine," *IEEE Transactions on Mobile Computing*, vol. 24, no. 5, pp. 4510–4528, 2025.
[33] "Customstreamgrouping," https://storm.apache.org/releases/2.4.0/javadocs/org/apache/storm/grouping/CustomStreamGrouping.html, 2022.
[34] E. Cai, R. Shiraki, and E. Oki, "Cloud server backup resource allocation models based on probabilistic protection," in *2025 21st International Conference on the Design of Reliable Communication Networks (DRCN)*, 2025, pp. 1–5.
[35] C. A. M Devendra Prasad, Prasanta Babu V, "Machine learning ddos detection using stochastic gradient boosting," *International Journal of Computer Sciences and Engineering*, vol. 7, pp. 157–166, 2019.
[36] C. E. Lopez and C. Gallemore, "An augmented multilingual twitter dataset for studying the covid-19 infodemic," *Social Network Analysis and Mining*, vol. 11, no. 1, pp. 1–14, 2021.

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.

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.



Jie Wen is a postgraduate student at the School of Information Engineering, China University of Geosciences, Beijing, China. He received his Bachelor Degree in Electrical Engineering and Automation from Anhui Polytechnic University in 2021. His research interests include big data stream computing, data analytics and distributed systems.

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 Quantum Cloud Computing and Distributed Systems (qCLOUDS) 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 176 with 165,400+ citations). He is among the world' s top 2 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. 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.