Skewness-Aware Stream Partitioning: A Key Splitting Suppression Method for Distributed Stream Processing

Dawei Sun, Weilong Lv, Shang Gao, *Member, IEEE*, Keqin Li, *Fellow, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

***Abstract*—Partitioning algorithms are crucial in distributed stream computing, as they directly affect load balancing across downstream task instances, cluster performance utilization, and data stream processing efficiency. However, existing algorithms suffer from key limitations: they fail to effectively balance load imbalance and key-splitting overhead as excessive splitting increases aggregation costs and memory usage, while insufficient splitting fails to reduce skew. In addition, they cannot timely identify newly emerging high-frequency keys, because historical frequency records dominate detection, which worsens load imbalance over time. To address these limitations, we propose Sa-Stream, a skewness-aware partitioning framework that rapidly captures changes in stream skewness and dynamically adjusts partitioning decisions, thereby balancing load imbalance and key-splitting overhead under evolving stream distributions. (1) A key frequency decay mechanism progressively reduces the influence of historical frequency statistics to improve the prediction of evolving key frequencies. (2) An adaptive threshold modulation method dynamically adjusts the hot-key threshold based on the current key frequency distribution, balancing load imbalance and aggregation costs. (3) A multi-tiered key segregation mechanism classifies keys into cold, warm, and hot categories: cold keys use deterministic hash routing to minimize aggregation costs, hot keys use degree-adjusted key splitting to reduce load imbalance, and warm keys use one-time key splitting to bridge the two extremes. Experiments conducted on Apache Storm demonstrate that Sa-Stream outperforms existing stream partitioning approaches, increasing throughput by up to 16% and reducing latency by up to 50%. Additionally, key splitting is reduced by up to 18%, and load imbalance is decreased by up to 98%.**

***Index Terms*—Stream partitioning, stateful operator, load balance, skewed data stream, stream processing.**

Received 23 October 2025; revised 12 June 2026; accepted 23 June 2026. Date of publication 25 June 2026; date of current version 10 August 2026. This work was supported by the National Natural Science Foundation of China under Grant 62372419 and in part by the Fundamental Research Funds for the Central Universities under Grant 265QZ2021001. *(Corresponding author: Dawei Sun.)*

Dawei Sun and Weilong Lv are with the School of Artificial Intelligence, China University of Geosciences, Beijing 100083, China (e-mail: lvweilong@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 Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, University of Melbourne, Parkville, VIC 3010, Australia (e-mail: rbuyya@unimelb.edu.au).

Digital Object Identifier 10.1109/TSC.2026.3707515

I. Introduction

IN CONTEMPORARY service computing scenarios, such as real-time recommendation, intelligent transportation, online monitoring, and financial transaction systems, massive streaming data must be continuously processed in real time. Consequently, distributed stream computing has become a fundamental infrastructure for modern service-oriented computing applications. In practical distributed stream processing systems, applications are typically organized as directed acyclic graph (DAG)-based topologies, where streaming data continuously flows through multiple interconnected operators. Such processing pipelines can also be viewed as real-time data processing workflows in modern service computing environments. Modern distributed stream processing systems address this demand through parallelized data pipeline architectures [1], [2], [3]. However, real-world data streams face two co-occurring challenges: persistent data stream skewness [4] and temporal key popularity volatility, where dominant keys exhibit non-stationary distribution patterns across execution windows [5], [6]. Dealing with these skewed and dynamically changing key distributions is particularly challenging for maintaining processing efficiency in traditional stream processing paradigms [7], [8].

A critical objective in distributed stream processing is achieving load balancing across the cluster, as imbalanced allocation can lead to worker overload and resource waste [9], [10]. For stateful data streams, it is even more important to consider the memory overhead caused by maintaining replicated states on workers [11], [12].

Initially, two primary stream grouping algorithms were used to process data streams: field grouping and shuffle grouping. Field grouping usually employs hash-based routing to ensure tuple-to-task instance affinity, thereby maintaining key state consistency. However, this approach encounters limitations when handling skewed data distributions, resulting in persistent load imbalance across parallel task instances [13] and inefficient resource utilization due to overloaded nodes [14].

Shuffle grouping uses cyclic tuple distribution to achieve workload balance, but this approach introduces critical operational costs. Preserving per-key state continuity and aggregation semantics requires proportional memory and computational overhead as task instance parallelism increases [15].

To address the limitations of key partitioning and the overhead of shuffle grouping, Nasir et al. introduced the partial key grouping (PKG) algorithm, which allows tuples with the same key to be processed by two task instances [16]. However, PKG is also prone to load imbalance under high-frequency keys. To improve this, Nasir et al. proposed D-Choice, which assigns high-frequency keys to multiple instances while low-frequency keys remain under PKG [17]. This hybrid approach theoretically optimizes the trade-off between state consistency and load balance. However, its reliance on the Space Saving algorithm [18] for key frequency estimation makes it slow to adapt to rapidly changing stream distributions.

To overcome these limitations, we propose **Sa-Stream**, a skewness-aware stream partitioning framework featuring a **frequency decay prediction algorithm** for detecting emerging warm and hot keys in real time. Sa-Stream divides key frequency storage into two parts: warm and hot keys and their frequencies are stored in a list, and cold keys and their frequencies are maintained using a data structure called **Count-Min Sketch**, which is designed for approximate frequency estimation. Count-Min Sketch uses a compact probabilistic data structure that tracks item frequencies using sub-linear memory, at the cost of slight overestimation due to hash collisions.

In addition, Sa-Stream incorporates a **dynamic threshold adjustment mechanism** to balance aggregation cost (due to hot key splitting) and load imbalance (due to static partitioning). When key splitting is excessive but load imbalance is low, the hot key threshold is increased to reduce splitting overhead. Conversely, when splitting is low but load imbalance is high, the threshold is decreased to increase key splitting for improved balance.

Our main contributions are as follows:

- A skewness-aware stream partitioning framework, Sa-Stream, that integrates frequency prediction, adaptive hot-key threshold adjustment, and three-tier cold/warm/hot key routing.
- A frequency-aware adaptive partitioning mechanism that dynamically adjusts key-splitting decisions according to evolving stream skewness and key-frequency distributions.
- A comprehensive experimental evaluation on both synthetic and real-world datasets, demonstrating the effectiveness of Sa-Stream compared with representative existing methods.

In service-oriented real-time computing applications, the performance of distributed stream processing systems directly affects service quality and reliability. Workload imbalance caused by changing stream skewness can lead to latency fluctuations, throughput degradation, and unstable system behavior. By balancing processing load while controlling key-splitting overhead, Sa-Stream helps maintain stable latency and reliable throughput under changing workloads, thereby improving QoS and ensuring SLA (Service-Level Agreements) compliance.

The rest of the paper is structured as follows: Section II introduces related work. Sections III analyzes the problem and presents the system model. Section IV describes the architecture of Sa-Stream and its core algorithmic workflow. Section V reports the experimental evaluation and discusses the results. Finally, Section VI concludes the paper and outlines future research directions.

II. Related Work

In this section, we first review partitioning algorithms designed for load balancing and key splitting in distributed stream processing systems, and then examine existing approaches for key frequency prediction in dynamically changing streams.

A. Stream Partitioning

To balance load among downstream instances while minimizing aggregation costs and memory overhead caused by key splitting, current partitioning algorithms primarily rely on key frequency-based strategies [17], [19], [20], [21], [22], [23]. These methods assign high-frequency keys to multiple task instances, while routing low-frequency keys to a single instance using hash partitioning.

For example, PStream [20] and FlexSP [22] dynamically adjust the high-frequency threshold to accommodate the changing proportion of high-frequency keys, thereby balancing key splitting and load distribution. However, PStream [20] distributes high-frequency keys evenly among downstream instances, which incurs significant key splitting overhead. Similarly, skewness [22] divides keys into three frequency tiers: low, middle, and high, and uses adaptive key splitting for high frequency keys. Yet, its reliance on the Space-Saving algorithm limits its ability to accurately distinguish between low- and mid-frequency keys. By contrast, our proposed Sa-Stream accurately differentiates between low- and mid-frequency keys, and use adaptive key splitting and high-frequency threshold tuning to achieve a balanced trade-off between key splitting and load distribution.

To further reduce key splitting, some studies, such as DAGreedy [24] and Prompt [25], perform offline or micro-batch statistical analysis prior to partitioning. However, the offline approach in DAGreedy [24] cannot cope with real-time stream dynamics, while the micro-batch strategy in Prompt [25], although achieving near-optimal load balance and minimal key splitting, incurs high latency, making it unsuitable for real-time processing systems. By contrast, Sa-Stream performs lightweight cycle-based frequency analysis, significantly reducing computational overhead.

A summary of these research efforts is provided in Table I. Unlike previous approaches, our method introduces a multi-tiered hot key segregation mechanism that applies hierarchical key splitting strategies tailored to distinct frequency ranges. At the end of each processing cycle, real-time statistical analysis of the frequent-key list enables dynamic threshold adjustments, tuning the global key splitting ratio for optimal balance.

B. Key Frequency Prediction

The effectiveness of partitioning strategies fundamentally depends on accurate frequency prediction for keys in the data stream, particularly for detecting high-frequency keys [27]. Techniques in this domain often incorporate decay factors to

TABLE I
RELATED WORK

Method	System Performance		Data Distribution	
	Accuracy	Dynamic	Key Splitting	Complexity
FlexD [9]	×	×	Medium	Low
SQ-RR [19]	×	✓	Medium	High
D/W [17]	×	✓	Medium	Low
DAGreedy [24]	✓	✓	Low	High
PKG [16]	×	×	Low	Low
FISH [23]	×	✓	Medium	Low
PStream [20]	×	✓	High	Medium
Dalton [26]	×	✓	Medium	High
Prompt [25]	×	✓	Low	High
STA [21]	✓	×	Medium	Medium
FlexSP [22]	×	✓	Medium	Medium
Sa-Stream (Ours)	✓	✓	Low	Medium

TABLE II
NOTATION USED IN THIS STUDY

Symbol	Description
$S^{i,j}$	Data stream from bolt B_i to bolt B_j.
t_i	i^{th} tuple in data stream.
k	Key value of tuple.
$S^{i,j}_m$	Data substream assigned to task instance m of $S^{i,j}$.
Lb_{B_j}	Load imbalance of bolt B_j.
B^i_j	Task instance i of Bolt B_j.
$L^i_{B_j}$	Load of task instance i in bolt B_j.
SN_{B_j}	Number of stream distinct keys of bolt B_j.
$SN_{B^i_j}$	Number of stream distinct keys of task instance B^i_j.
Rf_{B_j}	Replication factor of bolt B_j.
Rf_G	Replication factor for topology G.
Lb_G	Load imbalance for the topology G.
f_k	Frequency of key k.
θ_{hot}	Boundary threshold between hot and warm keys.
θ_{cold}	Boundary threshold between warm and cold keys.
Ib	Impact of load imbalance.
Iks	Impact of key splitting.

adapt to dynamic changes or employ multi-stage prediction models to organize key storage based on observed frequencies [28].

Early approaches such as PStream [20] and FISH [23] utilize the coin-toss algorithm and Space-Saving algorithm, respectively, to track key frequencies. Both methods apply attenuation at the end of each processing cycle, diminishing the influence of historical distributions and enabling faster adaptation to evolving frequency patterns. However, these methods adopt coarse-grained classification, distinguishing only between low-frequency and high-frequency keys while neglecting mid-frequency patterns. This limitation often leads to excessive key splitting.

Recent methods, including WavingSketch, SfSketch and ASketch [21], [29], [30], [31], enhance Count-Min Sketch-based models by implementing a two-tiered structure for frequency estimation. The primary sketch monitors low-frequency keys and promotes candidates to the secondary tier when their estimated frequency exceeds a threshold. This separation allows the primary sketch to focus on low-frequency key detection with improved precision, while the secondary structure tracks high-frequency keys. Such a decoupled design reduces complexity by assigning lightweight sketch operations to low-frequency tracking and reserving costly predictions for high-frequency keys. However, these methods respond poorly to dynamic changes in hot-key distributions: although low-frequency variations are detected quickly, updates for high-frequency keys incur latency during cycle re-initialization.

To address these limitations, our proposed Sa-Stream introduces a frequency decay prediction that combines frequency stratification with periodic decay. This design enables real-time adaptation to dynamic key distributions, maintaining high prediction accuracy even under volatile workloads.

Compared with FISH, Sa-Stream further refines the frequency hierarchy by explicitly introducing a warm-key category between cold and hot keys. Traditional two-tier hot/cold partitioning may treat medium-frequency keys similarly to hot keys, which can lead to excessive randomized routing and unnecessary key replication. By introducing a dedicated warm-key layer with differentiated routing behavior, Sa-Stream reduces unnecessary key splitting while maintaining load balancing capability.

Compared with FlexSP and D-Choice, which mainly rely on static or window-based frequency estimation mechanisms. Although these approaches can effectively capture dominant hot keys under stable workloads, they are less sensitive to rapidly evolving stream distributions because historical statistics may continue to dominate frequency estimation. In contrast, Sa-Stream incorporates periodic frequency decay into the prediction process, which reduces the influence of outdated statistics and enables faster identification of newly emerging hot and warm keys under dynamically changing skewed streams.

III. PROBLEM STATEMENT

In this section, we first introduce the splitting process in stream processing systems and explain the operational principles of stateful operators. We then discuss two important parameters related to data stream partitioning: load imbalance and replication factor.

A. Stateful Operator

In distributed stream processing systems, a streaming applications can be represented as a topology graph in the form of a directed acyclic graph (DAG). In this graph: Spouts are data source components that ingest external data (e.g., message queues, logs) and emit streams of tuples into the topology. They continuously read incoming events and inject them into the processing pipeline. Bolts are processing units that subscribe to one or more input streams, perform computations on the tuples (such as filtering, aggregation, or transformation), and emit new streams to downstream bolts.

Together, spouts and bolts form the vertices of the DAG, while the data streams connecting them constitute the edges, defining the flow of data through the system [32], [33].

A data stream consists of a continuous series of tuples, denoted as $S = \{t_1, t_2, \ldots, t_i, \ldots\}$, where t_i represents a tuple in the stream, and each tuple is associated with a key value k [34], [35].

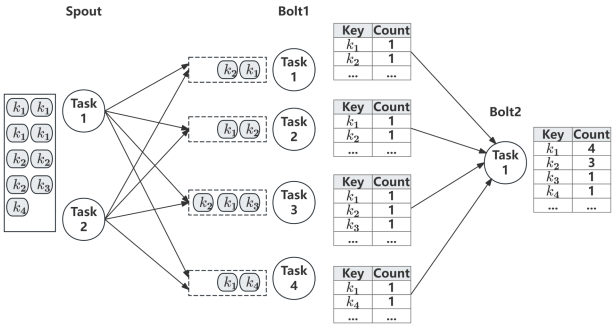


Fig. 1. Stateful operators with field grouping.

The processing logic of each bolt is executed by multiple parallel task instances. Each task instance functions as an autonomous execution unit that fully replicates and performs the operations defined by the bolt. These task instances are responsible for receiving data from upstream task instances, processing it, and forwarding the results to downstream bolts. Each task instance is the smallest unit of execution and is typically implemented as a single thread by default.

When a data stream $S^{i,j}$ is transmitted from bolt B_i to a downstream bolt B_j, it is partitioned into substreams and distributed across the bolt's m task instances $\{B_j^1, B_j^2, \ldots B_j^i, \ldots B_j^m\}$. The process of splitting the stream involves dividing the input stream $S^{i,j}$ into m substreams, each containing a subset of tuples from the original stream: $S^{i,j} \rightarrow \{S_1^{i,j}, S_2^{i,j}, S_3^{i,j}, \ldots, S_m^{i,j}\}$.

In stateful operators, it is essential to retain information about the keys associated with tuples. Since different tuples may have different key values, a stateful operator processes the stream as a sequence of key-associated data. For example, $S = \{\underbrace{t_1, t_2, t_3, t_4}_{k_1}, \underbrace{t_5, t_6, t_7}_{k_2}, \underbrace{t_8}_{k_3}, \ldots\}$.

When using the field grouping algorithm, as shown in Fig. 1, tuples with the same key value are routed to the same downstream task instance. The inter-bolt substreams from bolt B_i to different instances of bolt B_j can be represented as: $S_1^{i,j} = \{t_1, t_2, t_3, t_4\}$, $S_2^{i,j} = \{t_5, t_6, t_7\}$, and $S_3^{i,j} = \{t_8\}$ [36], [37].

B. Load Imbalance

Distributed stream processing architectures require partitioning algorithms that balance the load across task instances within each bolt. For B_j^i, the i^{th} task instance of bolt B_j, its load $L_{B_j}^i$ is defined as the number of tuples it processes per unit time [38]. The load imbalance degree among n task instances of bolt B_j, denoted as Lb_{B_j}, is defined as:

$$Lb_{B_j} = \frac{\max\limits_{1 \le i \le n} \{L_{B_j}^i\} - \frac{1}{n}\sum_{i=1}^{n} L_{B_j}^i}{\frac{1}{n}\sum_{i=1}^{n} L_{B_j}^i}. \tag{1}$$

When the load imbalance degree Lb_{B_j} indicates significant disparity in workload among task instances of bolt B_j. For example, as shown in Fig. 1, Task 1 and Task 4 of Bolt 1 receive data at a 4:1 ratio, highlighting severe load imbalance. Such imbalance increases the possibility of task instance overload and inefficient resource utilization.

To measure the overall load imbalance across the entire topology G, we define Lb_G as the average of all individual bolt load imbalance values. As shown in (2), where m is the number of bolts (i.e., the number of vertices in the DAG).

$$Lb_G = \frac{\sum_{j=1}^{m} Lb_{B_j}}{m}. \tag{2}$$

C. Replication Factor

To improve load balancing among task instances, several partitioning strategies, such as Partial Key Grouping (PKG) [16] and shuffle grouping [15], introduce key splitting, which allows a key to be processed by multiple task instances. As illustrated in the Fig. 1, tuples sharing the same key value are distributed among different task instances of bolt 1. This distribution strategy achieves a similar tuple count across the bolt's task instances, thereby enhancing load balancing.

Let us define the replication factor of bolt B_j. Suppose $SN_{B_j^i}$ denotes the number of unique keys stored in task instance B_j^i, and $SN_{B_j}(t)$ denotes the total number of distinct keys handled by the bolt B_j. The replication factor Rf_{B_j} is defined as:

$$Rf_{B_j} = \frac{\sum_{i=1}^{n} SN_{B_j^i}}{SN_{B_j}}. \tag{3}$$

This factor Rf_{B_j} captures the degree of key state duplication across instances. A higher replication factor implies that more keys are redundantly stored, increasing memory and aggregation costs. The minimum replication factor is achieved when each key is processed by exactly one task instance, as is the case under field grouping.

To evaluate the system-wide replication, we define the overall replication factor Rf_G as the average of all bolt replication factors in the topology:

$$Rf_G = \frac{\sum_{j=1}^{m} Rf_{B_j}}{m}, \tag{4}$$

where n is the number of bolts in G.

Partitioning algorithms aim to minimize Lb_G and Rf_G, thereby reducing the overload impact caused by load imbalance and high aggregation costs. Many algorithms, such as D-W choices [17], predict hot keys in the data stream and selectively split only those keys that dominate the stream. This approach helps lower both Lb_G and Rf_G.

IV. Sa-Stream Design and Implementation

In this section, we introduce the proposed Sa-Stream, detailing its components, frequency attenuation prediction algorithms, dynamic hot key threshold adjustment, and partitioning strategies for keys of different frequency levels.

A. Sa-Stream Components

As shown in Fig. 2, Sa-Stream comprises three coordinated components:

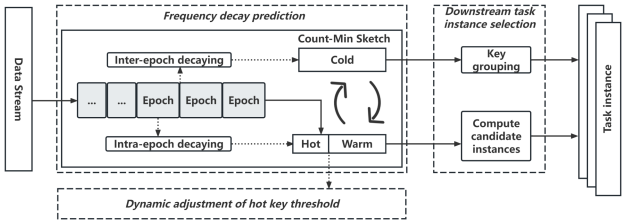


Fig. 2. Sa-stream components.

Frequency decay prediction: This module integrates the Space-Saving algorithm and Count-Min Sketch to track key frequency distributions. A sliding window mechanism is employed to dynamically capture evolving patterns within data streams, while an exponential decay function is utilized to mitigate the influence of stale historical data. This design enables timely updates to the frequent-key list in response to substantial fluctuations in key popularity.

Downstream task instance selection: To address skewed data distributions and sustain load balancing, keys are categorized into three tiers (hot, warm, and cold) based on their frequency decay prediction. Distinct field grouping strategies are applied to each category, accompanied by adaptive key splitting. Furthermore, a dynamic hot-key threshold adjustment algorithm is proposed, which periodically rebalances the trade-off between load balancing and key splitting according to the current key frequency distribution.

Dynamic hot-key threshold adjustment: To balance load imbalance and key-splitting overhead under dynamically changing stream distributions, Sa-Stream periodically adjusts the hot-key threshold according to the current frequency characteristics of hot and warm keys. This mechanism enables faster adaptation of partitioning strategies to evolving skewed streams.

The core design of Sa-Stream focuses on rapidly perceiving and responding to changes in stream key-frequency distributions. Sa-Stream first employs a periodic frequency decay mechanism to reduce the influence of outdated statistics, enabling faster identification of evolving hot- and warm-key characteristics in dynamically changing streams. Based on the detected frequency distribution, Sa-Stream further introduces an adaptive hot-key threshold adjustment mechanism, which directly estimates whether the current system overhead is dominated by load imbalance or excessive key-splitting overhead according to the number and cumulative frequency of hot and warm keys. Unlike traditional approaches that adjust thresholds according to the observed post-partition load balance or replication factor, Sa-Stream directly reacts to changes in the incoming stream distribution, allowing faster and more direct adaptation of the partitioning strategy. Finally, a three-level cold–warm–hot routing mechanism is adopted to efficiently map the adjusted partitioning decisions onto downstream task instances.

B. Frequency Decay Prediction

We use Count-Min Sketch to store and predict the frequencies of cold (low-frequency) keys, while mid- and high-frequency (warm and hot) keys are stored in a separate frequent-key list.

1) Count-Min Sketch: As a hash-based probabilistic data structure, Count-Min Sketch tracks frequency with sublinear memory.

It consists of a table with width $w = \lceil e/\epsilon \rceil$ and depth $d = \lceil \ln(1/\delta) \rceil$, along with d pairwise-independent hash functions $h_1, \ldots, h_d$. With a probability of $1 - \delta$, the total error in frequency estimation is bounded by ϵ. In our implementation, we set $\delta = 0.1$ and $\epsilon = 0.05$, resulting in a Count-Min Sketch of width $w = 28$ and depth $d = 3$.

When a new data item (i_t, c) arrives, where i_t is the key value of the new item at time t and c is the number of occurrences of this key value, the d hash functions compute d different values for i_t. Then, the corresponding rows in the table are incremented by c. In the streaming model, when each tuple is processed by Count-Min Sketch, the key value of the tuple is i_t, and each tuple appears once, with all c values being 1.

To query the estimated frequency of key i_t, the sketch calculates the values corresponding to d hash functions, and then find the minimum value of the corresponding row, which is $\min_j count[j, h_j(i_t)]$ This value provides an upper-bound estimate due to possible hash collisions. As the number of stored items increases, estimation error accumulates. Therefore, it is necessary to periodically clear or decay the counters to maintain accuracy.

2) Frequency Decay Prediction: This module consists of two key procedures:

i) *Frequent-key list maintenance:* Upon receiving a tuple, the module first checks whether its key is in the frequent-key list. If present, the corresponding key's counter is incremented by one. Otherwise, the module checks for available space in the list. If space is available, the key is inserted with an initial count of 1.

When encountering a tuple whose key is absent from a full frequent-key list, the algorithm: (1) inserts the key into the Count-Min Sketch, (2) updates the corresponding Count-Min Sketch counters via hashing operations, and (3) obtains the frequency estimate. This estimate is then compared with the cold key threshold θ_{cold} multiplied by the global tuple count. When the frequency estimate of the key value is less than the cold key threshold θ_{cold} multiplied by the global tuple count, it is judged as a cold key. Otherwise, further processing is carried out in procedure (2).

Cold keys are identified as those exceeding a predefined threshold within a specific time window in the data stream, rather than being determined over the entire runtime of the topology. The judgment is performed from the start of each cycle. Specifically, we classify a key as cold, warm, or hot based on the proportion of its occurrence count in the cycle relative to the total number of tuples processed in the current cycle. We set each cycle to process a fixed batch of 10,000 tuples.

ii) *Cold to warm promotion:* When a cold key's frequency exceeds a threshold θ_{cold}, it is treated as a promotion candidate. However, instead of immediately replacing the lowest-frequency key in the hot list, which can lead to frequent churn and overhead, Sa-Stream uses a gradual decay mechanism: Each time a cold

Algorithm 1: Frequency Decay Prediction.

Input: C: Count-Min Sketch, S: data stream, K: frequent-key list, N_{epoch}: epoch size, θ_{cold}: threshold, K_{max}: capacity of K, H: estimated frequency of key k from Count-Min Sketch

```
n_epoch ← 0, K ← ∅, C ← ∅
for t_i ∈ S do
    k_i ← t_i.k, n_epoch ← n_epoch + 1
    if n_epoch = N_epoch then
        Reset(C), n_epoch ← 0
    end
    if k_i ∈ K then
        UpdateCount(k_i, K)
    else
        if |K| < K_max then
            Insert(k_i)
        else
            H ← Estimate(C, k_i)
            if H > θ_cold then
                UpdateHot(k_i, K, H)
            end
        end
    end
end
```

Algorithm 2: Assign Task Instances Based on Key Frequency.

Input: W_{num}: total number of task instances; f_k: frequency of key k; θ_{hot}: hot key threshold; K: set of hot keys

Output: d: number of candidate task instances for key k

```
if k ∈ K then
    if f_k > θ_hot ∧ f_k < 0.5 then
        d ← 2 · f_k/θ_hot + 1; /* Assign the number of
          candidate task instances to the hot key. */
    end
    else if f_k > 0.5 then
        d ← W_num;
    end
    else
        d ← 2; /* Assign the warm key to two task
          instances. */
    end
    return d;
end
else
    return 1; /* Assign cold key using hash. */
end
```

key exceeds θ_{cold}, the count of the least frequent hot key is decremented. If the decremented count falls below θ_{cold}, the hot key is replaced with the cold key.

This mechanism prevents premature replacement and helps maintain stability in hot key tracking. Furthermore, the decay process removes stale hot keys that no longer appear in the stream, ensuring the list reflects current trends.

iii) *Frequency decay and time window management:* To ensure accurate and adaptive frequency estimation, Sa-Stream employs two time-based mechanisms: (1) Cold key decay. At the end of each time window, all counters in the Count-Min Sketch are reset to zero. The frequent-key list is preserved. This avoids false accumulation across windows, which could otherwise distort frequency estimates due to hash collisions from different keys over time. (2) Warm and hot key decay. To avoid persistent inflation of warm and hot key counts, which may block the identification of newly emerging hot keys, a decay mechanism inspired by the Misra-Gries algorithm [39] is applied: Whenever a cold key exceeds θ_{cold}, the counts of warm and hot keys are decremented by one. If a warm and hot key's count drops below θ_{cold}, it may be evicted and replaced by the cold key.

This dynamic strategy enables Sa-Stream to adapt to evolving key distributions and maintain accurate warm and hot key identification over time.

C. Downstream Task Instance Selection

Sa-Stream adopts a three-level classification structure to adaptively select the target instance for each tuple according to the frequency characteristics of its corresponding key. Keys stored in the frequent-key list are classified as hot keys when their frequencies exceed the hot-key threshold, and as warm keys when their frequencies are below the threshold, while keys maintained in the Count-Min Sketch are treated as cold keys.

Algorithm 2 outlines a stratified processing mechanism consisting of three coordinated strategies: (1) Cold keys are assigned using hash-based mapping to minimize aggregation cost. (2) Warm keys (with frequencies below the hot key threshold) are processed using the PKG algorithm. (3) Hot keys are assigned to a number of candidate task instances proportional to their frequency relative to the hot key threshold. By tuning the hot key threshold, the trade-off between load balancing and replication cost can be dynamically managed.

D. Dynamic Adjustment of Hot Key Threshold

As introduced in Section IV-C, we classify keys into three categories: Hot keys: frequency above the hot key threshold θ_{hot}, Warm keys: frequency between θ_{hot} and θ_{cold}, and Cold keys: frequency below θ_{cold}.

Hot keys are split and distributed across multiple task instances to avoid overload. Warm keys, while not frequent enough to require splitting, may still cause load imbalance if assigned via simple hash functions; thus, they are routed to two task instances using PKG. Cold keys, due to their minimal frequency, are hashed to a single instance to reduce replication overhead.

To optimize performance, the frequency threshold θ_{hot} must satisfy the following: (i) Minimize hot key count n_{hot} to reduce aggregation overhead. (ii) Limit cumulative cold key frequency f_{cold} to avoid skew. (iii) Maximize warm key frequency f_{warm}, ensuring only high-impact keys are split.

1) *Quantifying load imbalance and key splitting:* Let n denote the number of downstream task instances, and let d be the adaptive splitting degree applied to hot keys. We denote the normalized total frequency of hot, warm, and cold keys as f_{hot}, f_{warm}, f_{cold}, and their key counts as n_{hot}, n_{warm}, n_{cold}, respectively. We define the impact of each tier (hot, warm, cold) on load imbalance Ib and key splitting Iks as:

$$Ib = \left(\frac{d}{n}\right)^d f_{hot} + \left(\frac{d}{n}\right)^2 f_{warm} + \frac{d}{N} f_{cold}. \tag{5}$$

$$Iks = \frac{d \cdot n_{hot} + 2 \cdot n_{warm} + n_{cold}}{n_{hot} + n_{warm} + n_{cold}} \tag{6}$$

We model the overall performance cost C as a weighted sum of load imbalance and key splitting impacts:

$$C = W_{lb} \cdot Ib + W_{ks} \cdot Iks, \tag{7}$$

$$W_{lb} + W_{ks} = 1, \tag{8}$$

where W_{lb} is the weight of load imbalance, and W_{ks} is the weight of key splitting (replication).

2) *Frequency Distribution Constraints:* Assuming all keys collectively sum to 1 in frequency, we define:

$$\begin{cases} f_{hot} + f_{warm} + f_{cold} = 1 \\ n_{hot} + n_{warm} + n_{cold} = \text{count} \\ n_{hot} + n_{warm} = \text{count}_{\text{high}}. \end{cases} \tag{9}$$

Both count and $\text{count}_{\text{high}}$ are fixed in (9). Therefore, the optimizable part of (7) can be reduced to three dynamic terms: $W_{lb}[(\frac{d}{n})^d - \frac{d}{n}]f_{hot} + W_{lb}[(\frac{d}{n})^2 - \frac{d}{n}]f_{warm} + W_{ks}\frac{(d-2)n_{hot}}{\text{count}}$.

The cost function C is governed by three state variables. We determine the optimal threshold adjustment by evaluating the marginal impact of these variables. Given that the variation in the sum of hot key frequencies (f_{hot}) is equal in magnitude but opposite in sign to that of warm key frequencies (f_{warm}) (i.e., $\Delta f_{hot} = -\Delta f_{warm}$), the net cost variation ΔC for a threshold increment is formulated as:

$$\Delta C = W_{lb}\left[\left(\frac{d}{n}\right)^d - \left(\frac{d}{n}\right)^2\right]\Delta f_{hot} + W_{ks}\frac{(d-2)\Delta n_{hot}}{\text{count}}. \tag{10}$$

Consequently, we employ a gradient-based decision strategy. By computing the sign of ΔC resulting from a hypothetical threshold increase, we determine the adjustment direction: if $\Delta C > 0$, the threshold is decreased to reduce cost; otherwise, it is increased. Initially, we set the cold key threshold $\theta_{cold} = \frac{1}{n \ln n}$ and the hot key threshold $\theta_{hot} = \frac{1}{n}$, where n is the number of active keys. To ensure stable convergence a midst data fluctuations, the adjustment magnitude at each step is strictly constrained to 5% of the current threshold value (i.e., $\theta_{new} = \theta_{old} \times (1 \pm 0.05)$).

V. Performance Evaluation

We evaluate performance of our proposed Sa-Stream algorithm and compare its results with related algorithms in terms of latency, throughput, replication factor, and load imbalance.

TABLE III
Cluster Software Configurations

Software	Version	Function
Ubuntu	16.04.2	Operating System
JDK	1.8.0_271	JAVA runtime environment
Python	2.7	Python environment
Zookeeper	3.5.7	Distributed Service Framework
Apache Storm	2.4.0	Distributed stream processing platform

A. Experiment Setup

To evaluate Sa-Stream's performance, we deploy a Storm 2.4.0 cluster spanning 15 compute nodes, all virtualized on the VMware platform with underlying hardware powered by the Intel Xeon X5650 @ 2.67 GHz processor. Each node is equipped with 2 virtual CPUs, 2 GB of memory, a 20 GB hard disk, and a 100 Mbps network adapter. Table III shows the software configurations in each computing node.

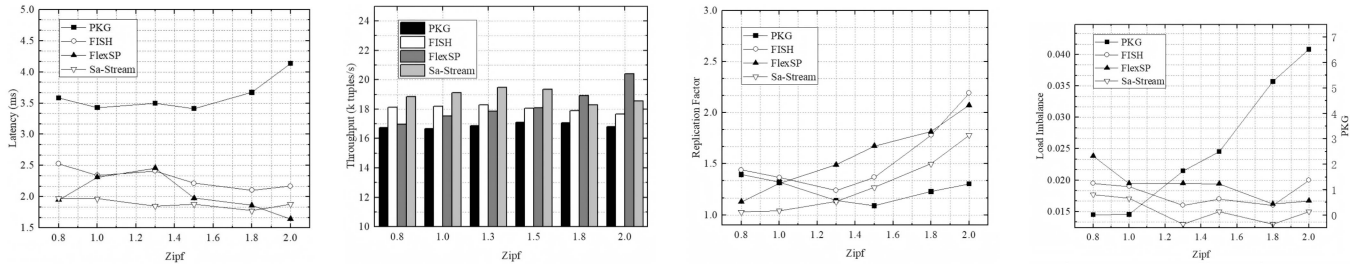
1) Datasets: We conduct comparative experiments using a real-world Didi ride-hailing dataset [40] and a synthetic fixed-skewness dataset: the Didi dataset contains 32.7 million vehicle trajectory records from Chengdu, China (November 2016), each comprising a UNIX timestamp (millisecond precision) and WGS84 geospatial coordinates (longitude: 103.7145°–104.3267°, latitude: 30.5228°–30.9114°); the synthetic dataset employs a controlled Zipfian generator to produce 10^6 unique keys, where the distribution of key value frequencies follows the skew set at the time of generation, with the skewness varying from 0.8 to 2.0.

The probability quality function of Zipf distribution is defined as (11):

$$P(X = k) = \frac{1/k^\alpha}{\sum_{i=1}^{n} \frac{1}{i^\alpha}}. \tag{11}$$

In (11), k denotes the rank position of an item (with higher frequencies corresponding to lower rank values), α represents the skewness controlling distribution asymmetry, and the denominator constitutes the normalization constant – a finite-term approximation of the Riemann zeta function. The skewness critically determines distribution characteristics: when $\alpha = 0$, the distribution degenerates to a uniform distribution with equal probability across all ranks. For $\alpha > 0$, increasing skewness induces exponential probability growth for top-ranked items ($k \to 1$), while simultaneously causing rapid probability decay in the tail ($k \to n$), thereby generating distinctive heavy-tailed distribution properties. The magnitude of α directly quantifies distribution concentration, with higher values producing extreme frequency disparities between head and tail items.

2) Baselines: We select PKG (the fundamental key splitting algorithm), FISH (an enhanced variant of PKG with D/W-Choices optimization), and FlexSP as baselines for evaluating Sa-Stream. FlexSP jointly adapts the hot key threshold, warm key ratio, and hot key splitting degree in a dynamic manner, which makes its design highly similar to ours. PKG achieves low aggregation costs aggregation base with low computational complexity. FISH incorporates the frequency decay prediction



(a) Latency. (b) Average throughput. (c) Replication factor. (d) Load imbalance.

Fig. 3. Performance of different grouping algorithms in processing datasets with different skewness.

algorithm and frequency proportional task instance allocation, theoretically ensuring lower aggregation costs and load imbalance when facing various data streams.

3) Experimental Comparison Design: To validate the effectiveness of the proposed three-tier key classification strategy (cold/warm/hot) in Sa-Stream over a modified two-tier variant (cold/hot) of the same algorithm, we design two sets of targeted comparative experiments:

i) *Two-tier vs three-tier comparison in Sa-Stream:* This experiment modifies the key classification and routing logic while keeping the overall framework unchanged. Specifically, cold keys ($p_k < 1/n \ln n$) remain with deterministic field-based routing ($D = 1$); whereas both warm and hot keys (merged as warm and hot keys and identified by Space Saving-based frequency prediction) are uniformly assigned adaptive D values using the same formulation in Algorithm 2, which is the same one employed for hot keys in the three-tier design.

This controlled setting isolates the impact of the warm key intermediate layer and verifies that fixed PKG routing for warm keys reduces unnecessary key splitting and computational overhead while maintaining load balance.

(ii) *Cycle period sensitivity comparison:* The cycle period used for Space-Saving-based frequency estimation and hot/warm key detection directly affects Sa-Stream's adaptability to time-varying data streams. To evaluate robustness, we conduct experiments across a wide range of cycle sizes (10^2, 10^3, 10^4, 10^5, 10^6 tuples) and measure performance in terms of latency, throughput, replication factor, and load imbalance. This analysis aims to identify an optimal cycle size that balances detection timeliness and computational overhead.

iii) *Decay step size sensitivity comparison:* To verify the impact of the decay step size on system latency, the experiment is conducted based using the test topology and the real-world dataset. The end-to-end latency per minute during the first 10 minutes is measured under five step sizes: 3%, 5%, 8%, 10%, and 33%.

4) Metrics: To compare the performance of the algorithms, we chose latency, throughput, replication factor, CPU usage, and load imbalance as comparison metrics.

Latency is the time it takes for each tuple to pass through the entire topology. Throughput is the total number of tuples processed by the current topology. These two metrics represent the processing speed of the system and can be obtained in the Storm UI interface. Replication factor Rf_G is calculated by (3), representing the key splitting for topology G. Degree of load imbalance Lb_{B_j} is calculated by (1), which represents the degree of load gap between task instances within the same bolt B_j. And Lb_G represents the average load imbalance for all the bolts in topology G. CPU utilization refers to the average CPU utilization of the entire topology within 3 minutes when using the test topology (1 Spout, 4 bolts).

B. Overall Performance Comparison With State-of-the-Art Algorithms

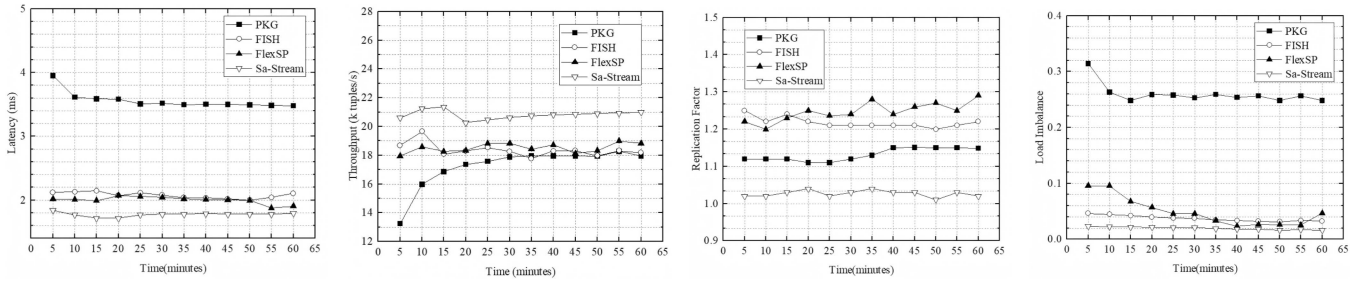
Fig. 3 and Fig. 4 present the performance of different grouping algorithms under synthetic datasets with varying skewness and the real-world dataset, respectively.

1) CPU Utilization: As shown in Fig. 5, Field Grouping achieves the lowest CPU utilization, since it only performs a single hash computation for tuple routing. All other algorithms consume more CPU resources due to more sophisticated mechanisms.

Sa-Stream introduces additional components, including Count-Min Sketch-based frequency estimation, adaptive hot-key threshold adjustment, dynamic key splitting updates, and frequency decay for warm and hot keys. Despite these mechanisms, its CPU utilization is only 0.75% higher than Field Grouping.

Compared with FISH, which employs Space-Saving and periodic frequency decay, Sa-Stream incurs only a marginal increase of 0.16%. Although FlexSP does not employ complex adaptive mechanisms and frequency prediction as Sa-Stream, it still uses formula-based adaptive key splitting for hot keys. The CPU utilization of Sa-Stream is only 0.033% higher than that of FlexSP.

2) Latency: Fig. 3(a) shows the variation of latency using synthesized datasets with different skewness. Sa-Stream achieves consistently lower latency across different skewness levels. Specifically, latency is reduced by 17.51%, 7.16%, and 47.61% compared to FISH, FlexSP, and PKG, respectively. In absolute terms, PKG exhibits latency no lower than 3.4 ms, while FISH remains above 1.9 ms. FlexSP ranges from 1.87 ms to 2.1 ms, whereas Sa-Stream maintains a stable latency around 1.8 ms, with the maximum not exceeding 1.9 ms.



(a) Latency. (b) Average throughput. (c) Replication factor. (d) Load imbalance.

Fig. 4. Performance of different grouping algorithms in processing a real-world dataset.

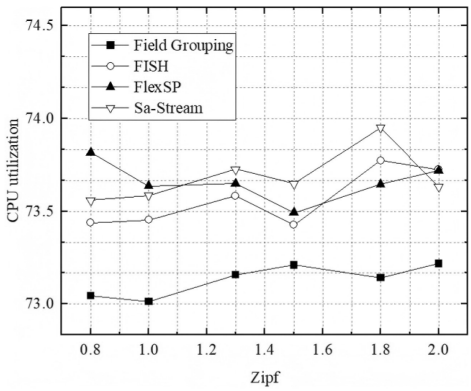


Fig. 5. CPU utilization of different grouping algorithms under varying Zipf coefficients.

Fig. 4(a) shows the latency comparison on the real Didi dataset. Over the entire topology execution, Sa-Stream achieves latency reductions of 14.38%, 11.2%, and 50.25% compared to FISH, FlexSP, and PKG, respectively.

3) Throughput: Since a fixed tuple arrival rate is used, throughput variations are relatively limited. FlexSP shows a gradual increase in throughput with increasing skewness, reaching a maximum of 20.394 k tuples/s.

Compared to Sa-Stream, throughput improvements are modest, with gains of 6.09% over FISH, 11.24% over PKG, and 3.52% over FlexSP under varying skewness. Over the entire execution, Sa-Stream achieves improvements of 15.42%, 16.90%, and 11.44%, respectively.

4) Key Splitting: Fig. 3(c) shows the comparison of replication factors generated by different algorithms when using datasets with different skewness. It can be seen that the replication factor decreases with increasing slope when using PKG, and then gradually increases again. The highest replication factor is 1.395 and the lowest is 1.09, with little fluctuation. When using FISH, the trend of replication factor changes is similar to PKG, both decreasing first and then increasing, but the increase in FISH is greater, with a minimum replication factor of 1.24 and a maximum of 2.19.

When facing datasets with low skewness, FISH provides two candidate task instances for most key values because the frequency of most key values cannot exceed the hot key threshold, while a small number of key values exceed the hot key threshold. Due to the small skewness of the dataset, the frequency of hot keys cannot be separated from the frequency of the remaining keys, making it difficult to identify. Therefore, there will always be different keys identified as hot keys, leading to an increase in the replication factor.

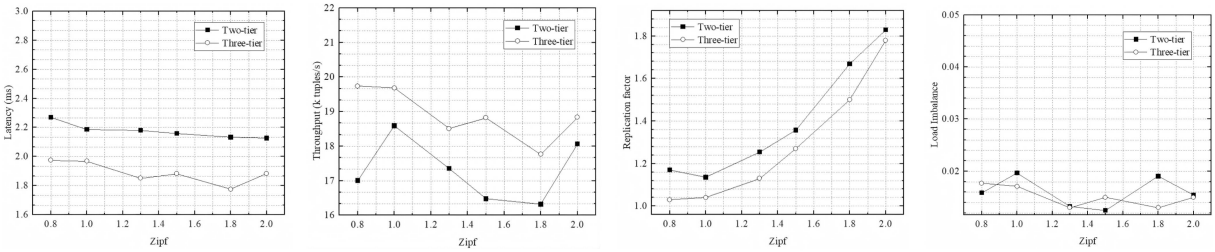
Sa-Stream applies deterministic routing for cold keys without any key splitting, resulting in a much lower replication factor under low skewness. As skewness increases, Sa-Stream adaptively increases the splitting degree for hot keys to maintain load balance. Thus, the replication factor of Sa-Stream rises steadily with skewness and does not decrease.

As the skewness gradually increases, the frequency of hot keys becomes more prominent, making it easier for prediction algorithms to identify hot keys, resulting in a decrease in the replication factor. But later on, as the skewness further increased, the number of keys greater than the hot key threshold increased, and therefore the replication factor also increased.

In FlexSP, the replication factor increases with data skewness. At low skewness, secondary hot keys are assigned the same splitting degree as the hottest keys, resulting in unnecessary replication. As skewness increases, FlexSP further expands splitting to both hot and lower-frequency keys to maintain load balance, leading to a higher overall replication factor.

When using Sa-Stream, because the algorithm sets cold keys, warm keys, and hot keys, only keys greater than the cold key threshold will be provided with two candidate task instances. The number of candidate task instances for hot keys is determined by the frequency of the hot key itself and the hot key threshold. In datasets with lower skewness, the frequency of hot keys is lower, so initially, most of the cold keys are assigned to one task instance using hash algorithms, while a small number of warm keys and hot keys are assigned to two or fewer task instances.

As the skewness increases, Sa-Stream will provide a candidate task instance number for high-frequency hot keys that matches its frequency, so the replication factor will also increase. However, because dynamically adjusting the hot key threshold will refer to the frequency distribution of the current hot key and warm key to adjust the replication factor, the hot key threshold will also increase as the hot key frequency increases. This is because the number of candidate task instances for hot keys is determined by the ratio to the current hot key threshold, so the



(a) Latency. (b) Average throughput. (c) Replication factor. (d) Load imbalance.

Fig. 6. Performance of three-tier and two-tier strategies in processing datasets with different skewness.

increase in replication factor is not as high as FISH. On datasets with different skewness, the replication factor of Sa-Stream decreases by an average of 17.11% compared to FISH, 18.35% compared to FlexSP, while it increases by an average of 4.48% compared to PKG.

Fig. 4(c) shows the comparison of replication factors when using the real-world dataset. It can be seen that the factors of these three algorithms fluctuate within a relatively small range. During the entire topology operation, the replication factor of Sa-Stream decreases by 15.73% compared to FISH, 17.68% compared to FlexSP, and 8.93% compared to PKG.

5) Load Balance: Fig. 3(d) shows the comparison of load imbalance when using datasets with different skewness. As the load imbalance of PKG increases rapidly with the increase of dataset skewness, the difference in load imbalance between Sa-Stream and FISH is not significant in the figure. Therefore, we present the results of PKG on the right y-axis and compare the load imbalance of these two algorithms (Sa-Stream and FISH). It can be seen that the load imbalance of both decreases with the increase of skewness and then increases. Because as the skewness of the dataset increases, the prediction algorithm of FISH and our Sa-Stream can more accurately identify hot keys.

However, when the skewness reaches 2.0, the number of candidate task instances allocated to the highest frequency hot key by FISH remains the same as the downstream task instances, while the number of candidate task instances allocated to other hot keys does not increase due to the increased frequency difference with the highest frequency hot key, resulting in an increase in load imbalance. Sa-Stream increases the hot key threshold to reduce aggregation costs, resulting in a decrease in the number of candidate task instances for some hot keys and an increase in load imbalance. However, the fluctuation range of load imbalance caused by these partitioning algorithms of FISH and our Sa-Stream is relatively small and has not significantly affected system performance. On datasets with different skewness, Sa-Stream shows a 15.58% decrease in load imbalance compared to FISH, 21.11% compared to FlexSP, while PKG shows a 73.77% decrease.

Fig. 4(d) shows the changes in load imbalance when using the real-world Didi dataset. Except for PKG, which fluctuates slightly due to the different skewness of data streams in different periods, both FISH and Sa-Stream can adapt to data streams that change over time due to the use of warm and hot key decay algorithms, resulting in smaller fluctuations in load imbalance. During the entire topology operation, Sa-Stream reduces load imbalance by 46.25% compared to FISH, 59.9% compared to FlexSP, and 92.23% compared to PKG.

C. Effectiveness of Three-Tier Vs. Two-Tier Strategy

1) Latency: Fig. 6(a) shows the latency on synthesized datasets with varying skewness. For both strategies, latency decreases as skewness increases. This is because as the proportion of hot keys in the data stream becomes increasingly dominant, fewer new hot keys need to be identified within each new epoch, which greatly reduces the occurrence of cold keys being converted into warm and hot keys.

However, the two-tier strategy consistently exhibits higher latency than the three-tier counterpart. This is because collapsing the three-tier structure into two-tier forces the adaptive splitting mechanism (originally designed for hot keys) to be applied to warm keys, which introduces unnecessary computational overhead. As a result, the latency of the two-tier strategy is 15.19% higher than that of the proposed three-tier approach.

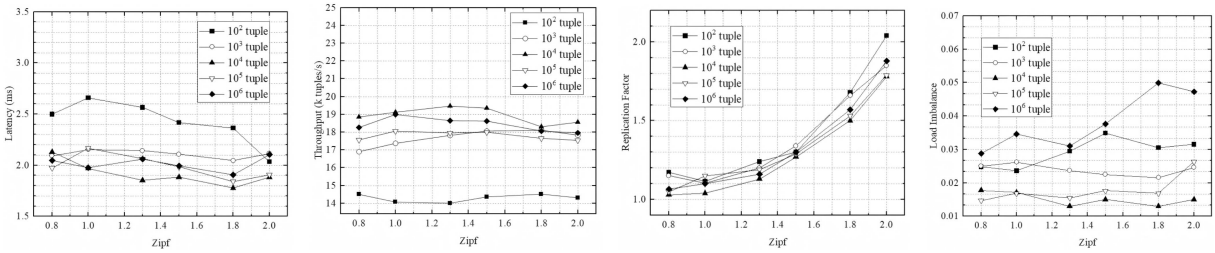
2) Throughput: Fig. 6(b) shows the average throughput under different skewness levels. The two-tier strategy achieves a maximum throughput of 18.563 k tuples/s and a minimum of 16.3186 k tuples/s, consistently lower than that of the three-tier design. Overall, the throughput decreases by 8.41% compared to the proposed approach.

3) Key Splitting: Fig. 6(c) compares the replication factors of the two strategies. Although both exhibit similar trends across different skewness levels, the two-tier strategy consistently results in a higher replication factor (+8.58%).

In the three-tier design, warm keys are assigned a fixed splitting degree of two, ensuring stable load balancing. In contrast, the two-tier strategy applies adaptive splitting to all warm and hot keys, causing some warm keys to be assigned a splitting degree of $D = 1$, which weakens load balancing.

To compensate, the system lowers the hot-key threshold, promoting more keys into the hot category and increasing their splitting degree. While this adjustment improves load balance, it also increases the number of split keys, thereby raising the overall replication factor.

As skewness increases, the gap between warm-key frequencies and the hot-key threshold widens. Consequently, fewer warm keys are affected by threshold adjustments, and only a limited number of hot keys increase their splitting degree, resulting in a relatively stable replication trend.



(a) Latency. (b) Average throughput. (c) Replication factor. (d) Load imbalance.

Fig. 7. Performance of different cycle periods in processing datasets with different skewness.

4) Load Balance: Fig. 6(b) shows the load imbalance under different skewness levels. Although the two-tier strategy exhibits slightly higher imbalance (+5.37%), neither approach consistently dominates across all settings. This is because the adaptive hot-key threshold mechanism dynamically adjusts the threshold based on the observed load imbalance, increasing or decreasing the splitting degree of warm and hot keys to stabilize the system.

Overall, while the two-tier strategy partially compensates for imbalance through threshold adaptation, the absence of a dedicated warm tier leads to less efficient and less stable load distribution.

D. Parameter Sensitivity Analysis

1) Sensitivity to Cycle Period: Fig. 7 presents the performance of Sa-Stream under different cycle periods. Overall, shorter cycles require more frequent resets of the Count-Min Sketch, which introduce substantial maintenance overhead and increase latency. In contrast, excessively long cycles cause frequency estimates to become stale, delaying the detection of emerging hot keys and resulting in a sharp increase in load imbalance.

Although the adaptive hot-key threshold mechanism is designed to mitigate load imbalance by lowering the threshold and increasing key splitting, it becomes ineffective when detection is delayed. Without timely identification of newly emerging hot keys, adjusting the threshold alone cannot correct the underlying distribution skew, and latency continues to increase.

The experimental results show that a cycle length of 10^4tuples achieves the best trade-off between sketch maintenance overhead and detection accuracy.

2) Sensitivity to Decay Step Size: Fig. 8 presents the latency of Sa-Stream when processing the real-world dataset under different decay step sizes. Overall, a larger decay step size enables the algorithm to approach the optimal solution more rapidly, leading to a fast initial drop in latency. However, an excessively large step cannot converge precisely to the optimal state, resulting in a slightly higher steady-state latency. In contrast, a smaller decay step size slows down the convergence toward the optimal threshold, requiring multiple successive decay operations to meet the adjustment criterion. This causes relatively high latency during the early stage, although gradual fine-grained adjustment eventually yields a more accurate optimal solution.

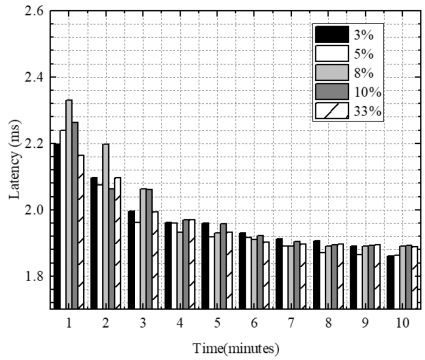


Fig. 8. Latency of different decay step sizes on the real-world dataset.

The results show that the 5% decay step achieves the lowest latency and fastest convergence, providing the best balance between balancing convergence speed and adjustment precision.

E. Experimental Discussion

1) Scalability Discussion: To further analyze the scalability characteristics of Sa-Stream, we discuss its behavior under larger-scale deployments. The experiments in this work were conducted on a 15-node virtualized cluster. The scalability limitation of Sa-Stream is relatively moderate due to its adaptive partitioning mechanism. In Sa-Stream, the splitting degree of hot keys is not statically predefined, but dynamically adjusted according to the current key frequency distribution and the adaptive hot-key threshold. Since the threshold adjustment mechanism jointly considers load imbalance and key-splitting overhead, the algorithm attempts to maintain acceptable load balance while avoiding unnecessary key replication overhead.

The adaptive hot key threshold adjustment mechanism helps Sa-Stream maintain scalability by dynamically balancing load imbalance and key-splitting overhead as cluster parallelism increases. Nevertheless, larger cluster deployments require the framework to maintain load information for a greater number of downstream instances and perform more extensive threshold adjustment operations. As a result, the computational overhead associated with load monitoring and threshold updates may also increase.

2) Discussion on Applicability and Limitations: From the CPU utilization results, Sa-Stream introduces slightly higher

computational overhead than the compared methods due to its frequency prediction mechanism based on Count-Min Sketch, Space-Saving, and periodic frequency decay. Therefore, the effectiveness of Sa-Stream depends on whether the reduction in skew-induced imbalance can outweigh the additional algorithm overhead.

This trade-off can also be observed under different skewness levels and workload dynamics. Under extremely high or low skew workloads (e.g., skewness values of 0.8 and 2.0), Sa-Stream may exhibit slightly higher latency and replication factor than FlexSP. This is because lightweight window-based estimation methods can already identify dominant hot keys effectively in highly skewed streams, while prediction errors under low-skew streams have relatively limited impact on load balancing.

The performance difference becomes more evident under dynamically evolving workloads. In the synthetic datasets, skewness fluctuations are mainly generated through randomized distributions, resulting in relatively stable long-term key popularity patterns. In contrast, the DiDi dataset exhibits more frequent skewness variations and dynamic hot-key replacements, representing a more volatile real-world streaming scenario. Under such conditions, the adaptive prediction and decay mechanisms of Sa-Stream become more beneficial, enabling more accurate hot-key identification and faster adaptation to changing stream distributions than window-based estimation approaches such as FlexSP.

Experimental results show that the latency improvement of Sa-Stream over FlexSP is significantly larger on the DiDi dataset than on the synthetic datasets. The reason is that the window-based key frequency estimation mechanism adopted by FlexSP is less effective in tracking rapidly changing key distributions under dynamically evolving workloads. By contrast, Sa-Stream is specifically designed for streams with dynamically changing skewness and hot-key distributions. Its adaptive mechanism continuously senses changes in stream skewness and rapidly adjusts hot-key thresholds and partitioning strategies accordingly, rather than relying solely on downstream load imbalance observations after partitioning decisions have already been applied.

In addition, the current study focuses on homogeneous cluster environments. Similar to many existing skew-aware partitioning approaches, the load of downstream instances is estimated primarily according to the number of tuples processed by each instance. Although effective in homogeneous environments, this metric does not explicitly consider differences in node processing capabilities. Consequently, Sa-Stream does not currently distinguish between load imbalance caused by skewed key distributions and performance differences resulting from heterogeneous computing resources.

VI. Conclusion and Future Work

We proposed Sa-Stream, a skewness-aware stream partitioning framework that rapidly captures changes in stream skewness and dynamically adapts partitioning decisions to balance load imbalance and key-splitting overhead. Sa-Stream integrates frequency decay prediction, adaptive hot-key threshold adjustment, and three-tier cold–warm–hot key routing into a unified framework. Experimental results on Apache Storm demonstrate the effectiveness of Sa-Stream compared with representative existing methods, including FISH, PKG, and FlexSP.

In future work, we plan to incorporate adaptive load-capacity estimation mechanisms to better characterize downstream processing capabilities and further improve the applicability of Sa-Stream in heterogeneous and dynamically changing cluster environments.

References

[1] A. Toshniwal et al., "Storm, Twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.

[2] H. Hadian, M. Farrokh, M. Sharifi, and A. Jafari, "An elastic and traffic-aware scheduler for distributed data stream processing in heterogeneous clusters," *J. Supercomputing*, vol. 79, no. 1, pp. 461–498, 2023.

[3] S. De Capitani di Vimercati et al., "Scalable distributed data anonymization for large datasets," *IEEE Trans. Big Data*, vol. 9, no. 3, pp. 818–831, Jun. 2023.

[4] H. Chen, F. Zhang, and H. Jin, "Popularity-aware differentiated distributed stream processing on skewed streams," in *Proc. IEEE 25th Int. Conf. Netw. Protoc.*, 2017, pp. 1–10.

[5] D. Wladdimiro, L. Arantes, P. Sens, and N. Hidalgo, "PA-SPS: A predictive adaptive approach for an elastic stream processing system," *J. Parallel Distrib. Comput.*, vol. 192, 2024, Art. no. 104940.

[6] Y. Xiao, X. Li, T. Li, R. Wang, Y. Pang, and G. Wang, "A distributed generative adversarial network for data augmentation under vertical federated learning," *IEEE Trans. Big Data*, vol. 11, no. 1, pp. 74–85, Feb. 2025.

[7] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, "Rhino: Efficient management of very large distributed state for stream processing engines," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2471–2486.

[8] A. Slo, S. Bhowmik, and K. Rothermel, "State-aware load shedding from input event streams in complex event processing," *IEEE Trans. Big Data*, vol. 8, no. 5, pp. 1340–1357, May 2022.

[9] G. Liu, Z. Wang, A. C. Zhou, and R. Mao, "Adaptive key partitioning in distributed stream processing," *CCF Trans. High Perform. Comput.*, vol. 6, no. 2, pp. 164–178, 2024.

[10] P. Omoregbee and M. Forshaw, "Performability requirements in making a rescaling decision for streaming applications," in *Proc. Eur. Workshop Perform. Eng.*, 2022, pp. 133–147.

[11] S. Ding, L. Yang, J. Cao, W. Cai, M. Tan, and Z. Wang, "Partitioning stateful data stream applications in dynamic edge cloud environments," *IEEE Trans. Services Comput.*, vol. 15, no. 4, pp. 2368–2381, Jul./Aug. 2021.

[12] H. Li, J. Xia, W. Luo, and H. Fang, "Cost-efficient scheduling of streaming applications in apache flink on cloud," *IEEE Trans. Big Data*, vol. 9, no. 4, pp. 1086–1101, Aug. 2022.

[13] A. Daghistani, M. Khayat, M. Felemban, W. G. Aref, and A. Ghafoor, "Guard: Attack-resilient adaptive load balancing in distributed streaming systems," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 6, pp. 4172–4186, Nov./Dec. 2022.

[14] A. Daghistani, W. G. Aref, A. Ghafoor, and A. R. Mahmood, "Swarm: Adaptive load balancing in distributed streaming systems for big spatial data," *ACM Trans. Spatial Algorithms Syst.*, vol. 7, no. 3, pp. 1–43, 2021.

[15] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola, "Online scheduling for shuffle grouping in distributed stream processing systems," in *Proc. 17th Int. Middleware Conf.*, 2016, pp. 1–12.

[16] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 137–148.

[17] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini, "When two choices are not enough: Balancing at scale in distributed stream processing," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 589–600.

[18] J. Shi, B. Wang, and Y. Xu, "Spruce: A fast yet space-saving structure for dynamic graph storage," in *Proc. ACM Manage. Data*, vol. 2, no. 1, pp. 1–26, 2024.

[19] F. Garcia-Carballeira, A. Calderon, and J. Carretero, "Enhancing the power of two choices load balancing algorithm using round robin policy," *Cluster Comput.*, vol. 24, no. 2, pp. 611–624, 2021.

[20] 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.
[21] A. Aslam, G. Simonini, L. Gagliardelli, L. Zecchini, and S. Bergamaschi, "Stream-aware indexing for distributed inequality join processing," *Inf. Syst.*, vol. 125, 2024, Art. no. 102425.
[22] S. Chen, D. Zuo, and Z. Zhang, "FlexSP:(1 β)-choice based flexible stream partitioning for stateful operators," in *Proc. 53 rd Int. Conf. Parallel Process.*, 2024, pp. 732–741.
[23] X. Liao, Y. Huang, L. Zheng, and H. Jin, "Efficient time-evolving stream processing at scale," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2165–2178, Oct. 2019.
[24] A. Pacaci and M. T. Özsu, "Distribution-aware stream partitioning for distributed stream processing systems," in *Proc. 5th ACM SIGMOD Workshop Algorithms Syst. MapReduce Beyond*, 2018, pp. 1–10.
[25] A. S. Abdelhamid, A. R. Mahmood, A. Daghistani, and W. G. Aref, "Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2455–2469.
[26] E. Zapridou, I. Mytilinis, and A. Ailamaki, "Dalton: Learned partitioning for distributed data streams," in *Proc. VLDB Endowment*, vol. 16, no. 3, pp. 491–504, 2022.
[27] R. Shahout and M. Mitzenmacher, "Learning-based heavy hitters and flow frequency estimation in streams," in *Proc. IEEE 32nd Int. Conf. Netw. Protoc.*, 2024, pp. 1–13.
[28] G. Zhuo, Z. Shu, and Z. Yu, "Deep residual coupled prompt learning for zero-shot sketch-based image retrieval," *IEEE Trans. Big Data*, vol. 11, no. 3, pp. 1493–1507, Jun. 2025.
[29] J. Li et al., "WavingSketch: An unbiased and generic sketch for finding top-k items in data streams," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 1574–1584.
[30] L. Liu et al., "SF-sketch: A two-stage sketch for data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2263–2276, Oct. 2020.
[31] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. 2016 Int. Conf. Manage. Data*, 2016, pp. 1449–1463.
[32] D. Sun, H. He, H. Yan, S. Gao, X. Liu, and X. Zheng, "Lr-stream: Using latency and resource aware scheduling to improve latency and throughput for streaming applications," *Future Gener. Comput. Syst.*, vol. 114, pp. 243–258, 2021.
[33] G. Russo Russo, V. Cardellini, and F. Lo Presti, "Hierarchical auto-scaling policies for data stream processing on heterogeneous resources," *ACM Trans. Auton. Adaptive Syst.*, vol. 18, no. 4, pp. 1–44, 2023.
[34] Y. Zhang et al., "Data-aware adaptive compression for stream processing," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 9, pp. 4531–4549, Sep. 2024.
[35] Y. Zhang, F. Zhang, H. Li, S. Zhang, and X. Du, "CompressStreamDB: Fine-grained adaptive stream processing without decompression," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 408–422.
[36] A. Aghdai, C.-Y. Chu, Y. Xu, D. H. Dai, J. Xu, and H. J. Chao, "Spotlight: Scalable transport layer load balancing for data center networks," *IEEE Trans. Cloud Comput.*, vol. 10, no. 3, pp. 2131–2145, 3rd Quart. 2022.
[37] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Auto-scaling for real-time stream analytics," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3338–3352, Dec. 2017.
[38] D. Los and T. Sauerwald, "An improved drift theorem for balanced allocations," *ACM Trans. Algorithms*, vol. 20, no. 4, pp. 1–39, 2024.
[39] C. J. Lebeda and J. Tetek, "Better differentially private approximate histograms and heavy hitters using the misra-gries sketch," in *Proc. 42nd ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, 2023, pp. 79–88.
[40] D. C. G. Initiative, "Didi chuxing dataset," 2021. [Online]. Available: https://outreach.didichuxing.com/

Dawei Sun received the PhD degree in computer science from Northeastern University, China, in 2012, and conducted the postdoctoral research in the department of computer science and technology with Tsinghua University, China, in 2015. He is a professor in the School of Information Engineering, China University of Geosciences, Beijing, China. His current research interests include Big Data computing, cloud computing and distributed systems. In these areas, he has authored more than 100 journal and conference papers.

Weilong Lv received the bachelor degree in computer science and technology from the China University of Geosciences Beijing, in 2023. He is currently working toward the master's degree in computer technology with the China University of Geosciences, Beijing. His research interests include Big Data stream processing and distributed systems.

Shang Gao (Member, IEEE) received the PhD 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.

Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, Big Data computing and distributed system etc. He is among the world's top 5 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 has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, and the *IEEE Transactions on Services Computing*. 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 director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory with 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 more than 850 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 181 with 171,700+ citations). He is among the world' s top 2 most influential scientists in distributed computing in terms of both singleyear impact and career-long impact based on a composite indicator of Scopus citation database.