Low-Disturbance Virtual-Key Grouping for Recovery Lag in Distributed Stream Computing Systems

Yongjie Yang[1], Dawei Sun[1*], Shang Gao[2], Rajkumar Buyya[3]

[1]*School of Artificial Intelligence*, *China University of Geosciences*, Beijing 100083, China
[2]*School of Information Technology*, *Deakin University*, Waurn Ponds, Victoria 3216, Australia
[3]*Quantum Cloud Computing and Distributed Systems Lab*, *University of Melbourne*, Parkville, Victoria 3010, Australia

[1]yangyongjie@email.cugb.edu.cn, sundaweicn@cugb.edu.cn
[2]shang.gao@deakin.edu.au
[3]rbuyya@unimelb.edu.au

***Abstract*— Bursty and skewed streams can cause severe load imbalance in distributed stream computing systems, making performance degradation persist even after the input rate stabilizes. This post-surge recovery lag prolongs latency spikes and throughput deficits. Existing stream grouping and scaling strategies influence the recovery process but primarily optimize load balance, grouping efficiency, or resource allocation, rather than explicitly minimizing recovery time. To address this limitation, we propose Rt-Stream, a recovery-aware and low-disturbance virtual-key grouping framework for distributed stream computing systems. Rt-Stream includes: (1) a recovery-oriented metric model that quantifies recovery completion for individual topologies and surge events using latency-violation and throughput-deficit durations; (2) a two-gate diagnosis mechanism that identifies recovery-risk topologies and actionable task imbalance on key-partitioned edges; and (3) a low-disturbance virtual-key grouping mechanism that selectively splits high-frequency keys, preserves the state semantics required by stateful operators, and manages split rules through refresh and expiration. We implement Rt-Stream on Apache Storm and evaluate it using multi-topology workloads derived from a real-world dataset. Experimental results show that Rt-Stream reduces recovery time by 64.8%–77.8%, lowers P95 latency during recovery by 51.6%–70.6%, and improves maximum aggregate throughput by 18.7%–65.0%.**

***Index Terms*— Distributed stream computing, recovery lag, virtual-key grouping, skewed data streams, bursty workloads.**

I. Introduction

Distributed stream computing systems, such as Storm [1], Heron [2], and Flink [3], have been widely used to process continuous data streams with low latency and high throughput [4]. In these systems, a streaming application is usually organized as a directed topology, where operators are connected by data streams and executed in parallel across distributed task instances [5], [6]. Stateful operators often partition tuples by a key, i.e., a field used to group related tuples, and maintain per-key state, such as a user ID or an item ID. When the key distribution is skewed, a small number of high-frequency keys can concentrate many tuples on a few downstream task instances, leading to load imbalance [7], [8].

In many real-world scenarios, stream workloads are not only skewed but also bursty [9], [10]. For example, an e-commerce platform may receive a sudden flood of purchase events when a promotion starts, and an online registration system may experience a sharp increase in requests when registration opens [11], [12]. Such bursts can temporarily overload operators and cause latency growth and throughput degradation.

Our controlled experiments on Apache Storm show that the impact of an input-rate surge may last beyond the surge itself. Even after the input rate stabilizes, some topologies remain in a degraded state for an extended period before returning to normal performance. We refer to this phenomenon as post-surge *recovery lag*. Our experiments reveal that: (1) input-rate surges can trigger prolonged recovery lags; (2) skewed key distributions can amplify recovery lags by concentrating tuples on a few downstream tasks; and (3) co-running topologies exhibit clearly different recovery times even under the same grouping strategy. These observations indicate that post-surge recovery should be treated as an explicit runtime objective.

Existing studies have improved stream computing performance from different perspectives. Skew-aware grouping methods, such as PKG [7], PStream [13], Fa-Stream [14], Disco [15] and FlexSP [16], mitigate load imbalance through key splitting, popularity-aware routing, frequency-aware grouping, and flexible choice control. Elastic scaling methods, such as DRS [17] and AuTraScale+ [18], adjust operator parallelism or resource allocation to accommodate changing input rates. Although these methods influence the recovery process, they primarily optimize load balancing, routing efficiency, throughput, or resource utilization rather than post-surge recovery itself. When burstiness and skewness jointly create localized backlogs, optimizing runtime load distribution or adapting physical resources may not shorten the recovery period. This motivates us to study post-surge recovery lag as an explicit runtime optimization problem.

TABLE I
COMPARISON OF REPRESENTATIVE STUDIES RELATED TO RT-STREAM

Aspects	Related Works							
	PKG [7]	PStream [13]	Fa-Stream [14]	Disco [15]	FlexSP [16]	DRS [17]	AuTraScale+ [18]	Rt-Stream (Our work)
Skew awareness	✓	✓	✓	✓	✓	×	×	✓
Low-disturbance control	△	△	△	–	△	×	×	✓
Bursty-workload support	×	×	△	×	△	✓	✓	✓
Routing adaptation	✓	✓	✓	–	✓	×	×	✓
Runtime reconfiguration-free	✓	✓	✓	–	✓	×	×	✓
Recovery-lag-oriented objective	×	×	×	×	×	×	×	✓

Note: ✓, △, ×, and – denote fully addressed, partially addressed, not explicitly addressed, and not applicable, respectively.

To address this limitation, we propose Rt-Stream, a recovery-aware and low-disturbance virtual-key grouping framework for distributed stream computing systems. Rt-Stream first identifies topologies experiencing post-surge recovery pressure based on latency and throughput attainment, and then determines whether the recovery pressure is associated with high-frequency keys and downstream task imbalance on key-partitioned edges. For eligible high-frequency keys, Rt-Stream selectively generates split rules that derive virtual keys to redistribute tuples across existing downstream task instances while preserving the original key for state access and result aggregation. It further manages split rules through refresh and expiration, allowing useful rules to be reused during recurring surges while removing inactive rules after recovery pressure disappears. By combining recovery-aware diagnosis with low-disturbance virtual-key grouping, Rt-Stream shortens post-surge recovery without incurring costly task migration, operator reconfiguration, or persistent routing disturbance. Table I compares Rt-Stream with representative related studies.

We implement Rt-Stream on Apache Storm and evaluate it using multi-topology workloads derived from a real-world dataset. Experimental results show that Rt-Stream significantly reduces recovery time and P95 latency during recovery, while improving sustainable aggregate throughput and resource utilization.

This paper makes the following contributions:

1) We characterize post-surge recovery lags in distributed stream computing systems and formulate recovery-lag mitigation using recovery-oriented metrics.
2) We design a two-gate diagnosis mechanism to identify recovery-risk topologies and actionable key-partitioned edges.
3) We propose a low-disturbance virtual-key grouping mechanism with selective key splitting and split-rule lifecycle management.

The rest of this paper is organized as follows. Section II presents the motivation and observations. Section III introduces the system model and problem formulation. Section IV describes the design of Rt-Stream. Section V evaluates the performance of Rt-Stream. Section VI concludes this paper and discusses future work.

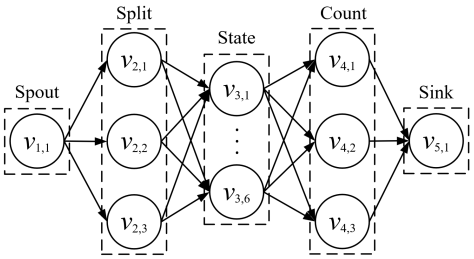


Fig. 1. Instance topology of WordCount

II. OBSERVATION AND MOTIVATION

We conduct a series of controlled experiments on an Apache Storm cluster to investigate the post-surge recovery dynamics in distributed stream computing systems.

A. Observational Experiments

We build an experimental platform on a cluster of homogeneous physical nodes. Each node is equipped with a 2-core Intel(R) Xeon(R) X5650 @ 2.67GHz CPU, 4GB RAM, and a 1000Mbps network interface. The system uses Apache Storm 2.8.0 with identical runtime configurations across all experiments. WordCount (WC) is a fundamental and widely-used benchmark for stream computing system performance evaluation. We employ its common rhombus topology as the test case, as shown in Fig. 1.

To emulate skewed streams, we generate keys following Zipf distributions with different skewness degrees [19]. A larger Zipf parameter indicates that more tuples are concentrated on a small number of keys. We use two commonly used grouping strategies: key grouping (KG) and shuffle grouping (SG) [16]. In our experiments, a topology is considered to be in the healthy region only when its end-to-end latency does not exceed 40 ms and its throughput attainment (Eq. (1)) is at least 0.95.

Observation 1: Input-rate surges can create post-surge recovery lags. We first demonstrate the existence of the post-surge recovery-lag phenomenon. Fig. 2 shows a latency trajectory under KG when the input rate increases from 3,000 to 9,000 tuples/s and then remains stable. The experiment uses a Zipf distribution with parameter $\theta = 1.0$. Although throughput attainment remains above 0.95 throughout the experiment, the latency does not immediately return to its previous stable

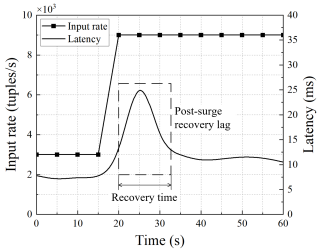


Fig. 2. Example of post-surge recovery-lag dynamics

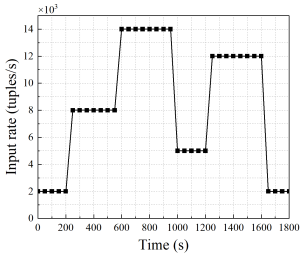


Fig. 3. Input-rate pattern used in the experiments

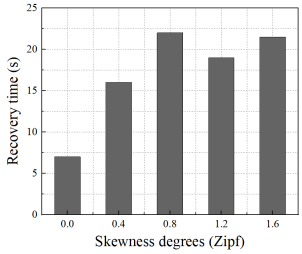


Fig. 4. Recovery time of different skewness degrees

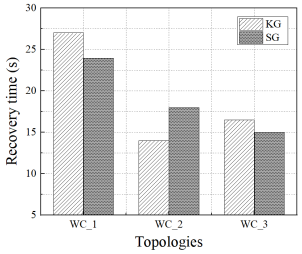


Fig. 5. Recovery time of WordCount (WC) topologies

level after the input rate stabilizes. Instead, it continues to rise for several seconds before gradually decreasing, forming a clear post-surge recovery lag. This observation indicates that recovery lag is a distinct runtime phenomenon and should be explicitly considered when adapting stream execution.

Observation 2: Data skew amplifies recovery lags. We next examine the impact of key skew on post-surge recovery. All runs in this experiment use KG and the input-rate pattern shown in Fig. 3. The Zipf parameter is varied as $\theta \in \{0.0, 0.4, 0.8, 1.2, 1.6\}$, and Fig. 4 reports the corresponding recovery times.

Compared with the no-skew case, skewed streams require substantially longer to recover after an input-rate surge. This result shows that data skew is not only a steady-state load-balancing problem, but can be a major contributor to prolonged recovery. Under key grouping, tuples with the same key are routed to the same downstream task. When a few high-frequency keys dominate the stream, these tasks become local bottlenecks and slow down backlog drainage during recovery. Therefore, mitigating post-surge recovery lags requires special attention to key distribution and the resulting downstream-task imbalance.

Observation 3: Recovery lags are uneven across co-running topologies. We further submit multiple WordCount (WC) topologies to the same cluster under the input-rate pattern shown in Fig. 3 and a Zipf distribution with $\theta = 1.0$. As shown in Fig. 5, co-running topologies exhibit clearly different recovery times even under the same grouping strategy. Under KG, WC_1 experiences a much longer recovery lag than WC_2 and WC_3. Under SG, the longest recovery lag still occurs in a specific topology rather than being evenly distributed across all co-running topologies. This result shows that recovery pressure is inherently uneven in a multi-topology stream computing cluster, and the slowest-recovering topology may dominate the overall recovery performance.

B. Motivations

The above observations show that post-surge recovery lag cannot be treated as ordinary runtime fluctuation or steady-state load imbalance. A straightforward solution is to redistribute tuples associated with high-frequency keys. However, indiscriminate key splitting is unsuitable for three reasons. First, key skew does not always cause prolonged recovery, so unnecessary grouping changes may introduce overhead without shortening recovery. Second, splitting an original key, i.e., the key used by the stream application to maintain per-key state, may violate state consistency unless the resulting partial states can be correctly merged. Third, since recovery pressure is transient, grouping adaptation should be limited in duration rather than retained after the topology returns to the healthy region.

These observations motivate the design of a recovery-aware and low-disturbance key grouping mechanism. This paper addresses the following questions:

1) Given runtime latency and throughput signals, how can prolonged post-surge recovery be detected promptly?
2) Given key distributions and downstream task workloads, how can recovery-relevant task imbalance be identified for grouping adaptation?
3) Given high-frequency keys during recovery, how can their tuples be redistributed with minimal disturbance while preserving the correctness of the original-key state?

III. System Model

A. Runtime Model and Signals

We consider a distributed stream computing cluster that continuously executes a set of co-running topologies $\mathcal{T} = \{1, 2, \ldots, N\}$. Each topology $i \in \mathcal{T}$ is represented as a directed acyclic graph $G_i = (V_i, E_i)$, where each vertex $v \in V_i$ denotes a stream operator, and each edge $e_{i,u,v} \in E_i$ denotes a data stream from operator u to operator v in topology i. An operator is a logical processing component, while its task instances are parallel runtime executors that process tuples assigned to this operator [20]. Let $p_{i,v}(w)$ denote the number of task instances (i.e., the parallelism degree) of operator v in topology i during monitoring window w. The duration of each monitoring window is denoted by Δt.

A topology receives external input tuples at rate $\Lambda_i(w)$ in window w. The service quality of topology i in window w is characterized by its end-to-end latency $L_i(w)$ and throughput attainment $A_i(w)$, where the latter can be defined as:

$$A_i(w) = \frac{\lambda_i(w)}{\Lambda_i(w)}, \tag{1}$$

where $\lambda_i(w)$ is its acknowledged throughput in window w. A lower $A_i(w)$ indicates that the topology i cannot process and acknowledge tuples at the expected rate.

The physical execution of a topology is constrained by the resources of the underlying cluster. Let $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ denote the set of compute nodes, where M is the number of nodes in the cluster. We consider a set of resource dimensions $\mathcal{D} = \{\text{cpu}, \text{mem}\}$, corresponding to CPU and memory resources [21]. For node c_m, let R_m^d denote its available amount of resource type $d \in \mathcal{D}$, and let $U_m^d(w)$ denote the resource usage of all workers running on node c_m in window w. The resource constraint is therefore

$$U_m^d(w) \leq R_m^d, \quad \forall c_m \in \mathcal{C}, d \in \mathcal{D}. \tag{2}$$

For an edge $e_{i,u,v} \in E_i$ in topology i, we call it a *key-partitioned edge* if tuples on this edge are assigned to task instances of downstream operator v according to the original key. Unless otherwise specified, a key k in this paper refers to the original key, i.e., the tuple field used by the stream application to group related tuples, maintain per-key state, and aggregate per-key results. Let $E_i^K \subseteq E_i$ denote the set of key-partitioned edges in topology i that are eligible for Rt-Stream intervention, where the superscript K indicates key partitioning. An edge is included in E_i^K only when the state associated with the same key k is decomposable and mergeable, meaning that partial results for the same key can be combined without changing the application result. Other key-partitioned edges keep their original grouping policy.

For a key-partitioned edge $e_{i,u,v}^K$, let $f_{i,u,v}(k, w)$ denote the fraction of tuples on this edge whose key is k in window w. A large $f_{i,u,v}(k, w)$ means that many tuples on this edge share the same key. Therefore, in a skewed stream, a few high-frequency keys account for a large fraction of tuples. Rt-Stream selects such high-frequency keys for split-rule planning (discussed in Section IV); they are still original keys, not a different key type.

For a key-partitioned edge $e_{i,u,v}^K$, let $x_{i,u,v,\ell}(w)$ denote the input rate delivered through this edge to the ℓ-th task instance of downstream operator v in topology i. We define the downstream-task input imbalance as

$$\Gamma_{i,u,v}^{task}(w) = \frac{\max_{1 \leq \ell \leq p_{i,v}(w)} x_{i,u,v,\ell}(w)}{\frac{1}{p_{i,v}(w)} \sum_{\ell=1}^{p_{i,v}(w)} x_{i,u,v,\ell}(w)}. \tag{3}$$

This metric compares the maximum input rate received by one task instance of downstream operator v with the average input rate across all task instances of v. A value close to 1 indicates balanced input distribution, while a larger value indicates that one or a few downstream task instances of v receive substantially more tuples than the average.

B. Recovery-lag Metrics

Within this paper, a surge event refers to an abrupt increase in the external input rate of one or more topologies. We denote a surge event s as

$$s = (w_s, \mathcal{T}_s) \in \mathcal{S}, \tag{4}$$

where $\mathcal{S}$ is the set of observed surge events, w_s is the window where the surge starts, and $\mathcal{T}_s \subseteq \mathcal{T}$ is the set of topologies whose input rates surge in event s and whose recovery behavior is evaluated. If multiple co-running topologies experience surges in the same workload episode, then $\mathcal{T}_s$ contains these topologies.

Let τ_L and $A_{\min}$ denote the system health thresholds for latency and throughput attainment, respectively. Topology i is considered healthy in window w if

$$L_i(w) \leq \tau_L, \tag{5}$$

and

$$A_i(w) \geq A_{\min}. \tag{6}$$

Let $\mathcal{W}_s = \{w_s, w_s + 1, \ldots, w_s + W_{\max} - 1\}$ denote the observation horizon after surge event s, where $W_{\max}$ is the maximum number of windows used to evaluate recovery. In our experiments, we use $W_{\max} = 300$ and $\Delta t = 1$s as a moderate setting. For each topology $i \in \mathcal{T}_s$, we define the latency-violation window set $V_i^L(s)$ and throughput-deficit window set $V_i^A(s)$ of topology i after surge event s as

$$V_i^L(s) = \{w \in W_s \mid L_i(w) > \tau_L\}, \tag{7}$$

and

$$V_i^A(s) = \{w \in W_s \mid A_i(w) < A_{\min}\}. \tag{8}$$

Accordingly, the latency-violation duration $LVD_i(s)$ and throughput-deficit duration $TDD_i(s)$ of topology i after surge event s are defined as

$$LVD_i(s) = |V_i^L(s)|, \tag{9}$$

and

$$TDD_i(s) = |V_i^A(s)|. \tag{10}$$

$LVD_i(s)$ measures how long topology i suffers from latency violation, while $TDD_i(s)$ measures how long it fails to keep up with the expected input rate. The unhealthy-window set of topology i after surge event s is defined as

$$U_i(s) = V_i^L(s) \cup V_i^A(s). \tag{11}$$

The recovery completion time of topology i after surge event s is denoted by $RCT_i(s)$, which is defined according to the last unhealthy window. If $U_i(s) = \emptyset$, topology i remains healthy during $\mathcal{W}_s$, and we set $RCT_i(s) = 0$. Otherwise, let $w_i^{last}(s) = \max U_i(s)$ denote the last unhealthy window. The recovery is confirmed only when topology i remains healthy for H consecutive windows after $w_i^{last}(s)$. In this case,

$$RCT_i(s) = w_i^{last}(s) - w_s + 1. \tag{12}$$

The consecutive-window requirement is used only to validate stable recovery and is not counted as additional recovery time. If the consecutive-health requirement cannot be satisfied within $\mathcal{W}_s$, we set $RCT_i(s) = W_{\max}$.

To characterize the recovery behavior after a surge event s, we define the event recovery completion time as

$$ERCT(s) = \max_{i \in \mathcal{T}_s} RCT_i(s). \tag{13}$$

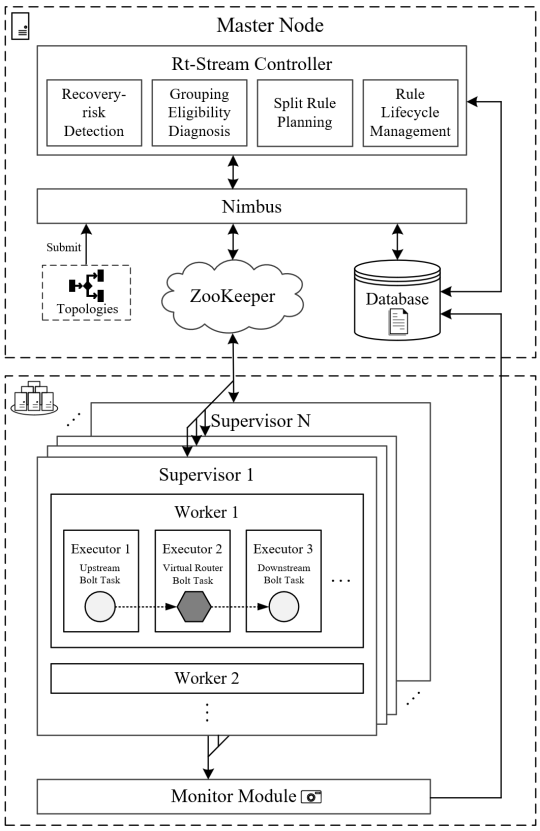


Fig. 6. System architecture of Rt-Stream

The desired recovery behavior is to minimize the recovery completion time:

$$\min_{\pi} \sum_{s \in \mathcal{S}} ERCT(s), \tag{14}$$

where π denotes the recovery-aware grouping policy.

Algorithm 1: Recovery-risk Detection and Grouping Eligibility Diagnosis

Input: Topologies $\mathcal{T}$, eligible key-partitioned edges E_i^K, metrics $L_i(w), A_i(w)$, key fraction $f_{i,u,v}(k,w)$, task input rate $x_{i,u,v,\ell}(w)$, thresholds τ_L, $A_{\min}$, imbalance threshold τ_Γ, maximum number of high-frequency keys K_h

Output: Split-rule planning requests $\mathcal{P}(w)$

foreach *control window* w **do**
  $\mathcal{P}(w) \leftarrow \emptyset$;
  `// Gate 1: detect recovery-risk topologies`
  $\mathcal{T}^r(w) \leftarrow \{i \in \mathcal{T} \mid L_i(w) > \tau_L \vee A_i(w) < A_{\min}\}$;
  if $\mathcal{T}^r(w) = \emptyset$ **then**
    continue;
  end
  `// Gate 2: check grouping eligibility on key-partitioned edges`
  foreach *topology* $i \in \mathcal{T}^r(w)$ **do**
    foreach *key-partitioned edge* $e_{i,u,v}^K \in E_i^K$ **do**
      $\mathcal{K}_{i,u,v}^h(w) \leftarrow$ top-K_h keys ranked by $f_{i,u,v}(k,w)$;
      Compute $\Gamma_{i,u,v}^{task}(w)$ by Eq. (3);
      if $\mathcal{K}_{i,u,v}^h(w) \neq \emptyset$ ***and*** $\Gamma_{i,u,v}^{task}(w) > \tau_\Gamma$ **then**
        $\mathcal{P}(w) \leftarrow \mathcal{P}(w) \cup \{(i, e_{i,u,v}^K, \mathcal{K}_{i,u,v}^h(w))\}$;
        Invoke Algorithm 2 with $(i, e_{i,u,v}^K, \mathcal{K}_{i,u,v}^h(w))$;
      end
    end
  end
end
return $\mathcal{P}(w)$;

IV. Rt-Stream: Architecture and Algorithms

A. System Architecture

Fig. 6 shows the system architecture of Rt-Stream. Users first submit streaming topologies to Nimbus. Nimbus cooperates with ZooKeeper to manage topology deployment and coordinates the execution on Supervisors. Each Supervisor hosts multiple Workers, and each Worker runs the corresponding Executors of the submitted topologies.

Rt-Stream extends this execution framework with three controlling components: the **Monitor Module**, the **Database**, and the **Rt-Stream Controller**. The **Monitor Module** interacts with the running Workers on Supervisors and continuously collects runtime information from co-running topologies, including external input rate $\Lambda_i(w)$, latency $L_i(w)$, throughput $\lambda_i(w)$, key fraction $f_{i,u,v}(k,w)$, and task input rates $x_{i,u,v,\ell}(w)$ on key-partitioned edges E_i^K. The **Database** stores these monitoring and controlling records, including split rules, rule updates, route statistics, and action logs. Split rules and action logs are written by the **Rt-Stream Controller**, while route statistics are collected from runtime execution and used for later grouping decisions.

The **Rt-Stream Controller** periodically reads monitoring data from the **Database** and performs four stages: (1) The Recovery-risk Detection stage checks whether a topology violates the latency or throughput-attainment health condition (Eqs. (5)–(6)) to identify recovery-risk topologies. (2) The Grouping Eligibility Diagnosis stage checks whether the degraded topology contains a key-partitioned edge with both high-frequency keys and downstream-task input imbalance. These keys are selected for split-rule planning. (3) The Split Rule Planning stage decides whether a high-frequency key should be split and how many virtual keys should be created for it; the chosen number is recorded as a *split factor*. The planning result is a *split rule*, which specifies how an original key should be mapped to routing keys. (4) The Rule Lifecycle Management stage installs, refreshes, expires, and records split rules. These operations do not perform physical task migration; they only control whether a split rule is active, reused, or removed.

For each eligible key-partitioned edge, Rt-Stream deploys a lightweight *Virtual Router Bolt* between the upstream and downstream operators. The *Virtual Router Bolt* generates an auxiliary *routing key* according to active split rules and uses this routing key only for downstream task assignment. Therefore, Rt-Stream adapts tuple grouping over existing task instances without task migration or operator reconfiguration.

B. Recovery-risk Detection and Grouping Eligibility Diagnosis

Rt-Stream first determines whether a topology needs grouping intervention and where such intervention can be applied. This diagnosis is organized as two gates.

The first gate detects recovery-risk topologies. In each control window w, Rt-Stream checks the health conditions of each topology (Eqs. (5)–(6)). A topology is added to the recovery-risk set $T^r(w)$ if it violates either the latency threshold τ_L or the throughput-attainment threshold $A_{\min}$. If $T^r(w) = \emptyset$, all monitored topologies are healthy in the current window, and Rt-Stream does not generate split-planning requests.

The second gate checks grouping eligibility on key-partitioned edges. For each topology $i \in T^r(w)$, Rt-Stream examines each eligible key-partitioned edge $e^K_{i,u,v} \in E^K_i$. On each edge, Rt-Stream selects at most K_h keys with the largest key fractions $f_{i,u,v}(k,w)$ as the high-frequency key set $K^h_{i,u,v}(w)$. It then computes the downstream-task input imbalance $\Gamma^{task}_{i,u,v}(w)$ (Eq. (3)). An edge is considered *grouping-eligible* only when $K^h_{i,u,v}(w) \neq \emptyset$ and $\Gamma^{task}_{i,u,v}(w) > \tau_\Gamma$, where τ_Γ is the imbalance threshold. Only grouping-eligible edges are passed to the split-rule planning procedure (Algorithm 2).

Algorithm 1 summarizes the two-gate diagnosis process. The controller uses statistics collected in each control window rather than per-tuple online optimization. In each control window, it scans the monitored topologies, checks eligible key-partitioned edges, maintains at most K_h high-frequency keys for each edge, and computes the downstream-task input imbalance. The per-window diagnosis cost is

$$O\left(|\mathcal{T}| + \sum_{i \in \mathcal{T}} \sum_{e^K_{i,u,v} \in E^K_i} (K_h + p_{i,v})\right).$$

This cost is incurred once per control window and can be regarded as part of the normal monitoring and control overhead of the stream computing system.

C. Low-disturbance Virtual-key Grouping

For a grouping-eligible key-partitioned edge $e^K_{i,u,v}$, Rt-Stream applies virtual-key grouping to redistribute tuples of selected high-frequency keys. As defined in Section III-A, k denotes the original key used by the application for state access and result aggregation. When a split rule is active, Rt-Stream derives virtual keys from k; each virtual key is an auxiliary routing key r used only for assigning tuples to downstream task instances. The original key k is not modified and is still used by downstream stateful or aggregation operators.

Algorithm 2: Low-disturbance Split-rule Planning

Input: Topology i, key-partitioned edge $e^K_{i,u,v}$, high-frequency keys $\mathcal{K}^h_{i,u,v}(w)$ from Algorithm 1, key fraction $f_{i,u,v}(k,w)$, split-factor set $\mathcal{Q}$, route-share upper bound $\rho_{\max}$

Output: Planned split rules $\mathcal{R}_{i,u,v}(w)$

$\mathcal{R}_{i,u,v}(w) \leftarrow \emptyset$;
Sort $\mathcal{K}^h_{i,u,v}(w)$ by descending $f_{i,u,v}(k,w)$;
foreach *key* $k \in \mathcal{K}^h_{i,u,v}(w)$ **do**
  $q_k \leftarrow 1$;
  `// Select the smallest effective split factor`
  foreach *split factor* $q \in \mathcal{Q}$ *in ascending order* **do**
    if $q = 1$ **then**
      continue;
    end
    Compute $\hat{\rho}_{i,u,v}(k,q,w) \leftarrow f_{i,u,v}(k,w)/q$;
    if $\hat{\rho}_{i,u,v}(k,q,w) \leq \rho_{\max}$ **then**
      $q_k \leftarrow q$;
      break;
    end
  end
  if $q_k > 1$ **then**
    $\mathcal{R}_{i,u,v}(w) \leftarrow \mathcal{R}_{i,u,v}(w) \cup \{(k, q_k)\}$;
  end
end
return $\mathcal{R}_{i,u,v}(w)$;

A split rule is represented as (k, q_k), where $k \in \mathcal{K}^h_{i,u,v}(w)$ is a high-frequency key and q_k is the selected split factor for this key. The split factor specifies how many virtual keys are created for k. Rt-Stream selects q_k from a bounded candidate set $\mathcal{Q}$. A candidate factor $q \in \mathcal{Q}$ denotes a possible number of virtual keys, and $q = 1$ means that the key is not split. Thus, q is a candidate value being tested, while q_k is the final value selected for key k.

For low disturbance, Rt-Stream searches $\mathcal{Q}$ in ascending order and selects the first effective factor. Let $\rho_{\max}$ denote the configured upper bound on the estimated traffic share of any virtual key derived from a high-frequency key k. For a candidate factor q, it is accepted only when $f_{i,u,v}(k,w)/q \leq \rho_{\max}$. Since task imbalance $\Gamma^{task}_{i,u,v}(w)$ has already been checked by Algorithm 1, this rule selects the smallest split factor that can reduce the concentration of a high-frequency key without introducing unnecessary virtual keys.

Algorithm 2 summarizes this split-rule planning process and outputs the planned rule set $\mathcal{R}_{i,u,v}(w)$. At runtime, the *Virtual Router Bolt* applies active split rules to incoming tuples. For a tuple with original key k, if no active split rule exists, the *Virtual Router Bolt* sets $r = k$. If an active rule (k, q_k) exists, the *Virtual Router Bolt* chooses a bucket identifier $b \in \{1, \ldots, q_k\}$ according to recent route loads and creates the virtual key $r = (k, b)$. Tuples are grouped by r,

while downstream stateful or aggregation operators still use the original key k. Therefore, Rt-Stream changes only the routing key used for task assignment and preserves the original-key semantics required by stateful processing.

For each split-planning request, Algorithm 2 checks at most K_h high-frequency keys and at most $|\mathcal{Q}|$ candidate split factors for each key. Thus, the worst-case split-planning cost for one key-partitioned edge in one control window is $O(K_h|\mathcal{Q}|)$. At runtime, tuple routing only requires an active-rule lookup and bucket selection, so the average overhead per tuple is $O(1)$ with a hash-table rule store. Since K_h and $|\mathcal{Q}|$ are bounded configuration parameters, the overhead is predictable and independent of the total key-space size.

D. Rule Lifecycle Management

Rt-Stream applies planned split rules through a bounded lifecycle. Each rule is bound to a specific topology, key-partitioned edge, and original key, and records the selected split factor and an expiration window set by the **Rt-Stream Controller**. This expiration window applies only to this rule rather than to all topologies. If a planned rule is not present in the rule store, the controller installs it as an active rule. If the rule already exists, the controller refreshes it by extending its expiration window.

A split rule is temporary. If it is not refreshed before expiration, Rt-Stream removes it from the active rule set. Expiration only prevents the rule from being applied to future tuples; it does not trigger task migration or state migration. Tuples emitted while a rule is active still carry their original key, and downstream stateful or aggregation operators use the original key to maintain state and combine partial results.

This lifecycle supports low-disturbance recovery control. Useful rules can be reused during recurring recovery pressure, while inactive rules are removed after recovery pressure disappears. As a result, Rt-Stream prevents temporary grouping changes during recovery from becoming permanent grouping policies.

V. Performance Evaluation

We evaluate Rt-Stream on Apache Storm with multi-topology workloads derived from a real-world dataset. The evaluation focuses on post-surge recovery behavior. We aim to answer the following questions:

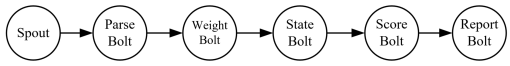
(1) Can Rt-Stream reduce the recovery time of co-running topologies after input-rate surges? (Section V-B)

(2) Can Rt-Stream reduce latency during recovery while avoiding persistent post-recovery degradation? (Section V-C)

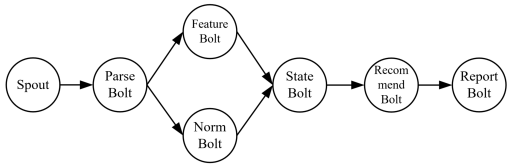
(3) Can Rt-Stream improve system throughput and resource utilization? (Section V-D and Section V-E)

A. Experimental Settings

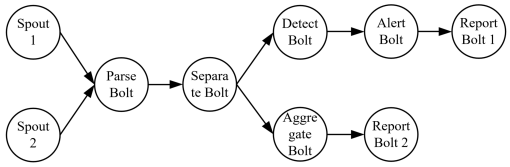
Rt-Stream is evaluated on a distributed Apache Storm cluster with one management node running Nimbus and ZooKeeper, and 31 compute nodes running Supervisors. Each compute node hosts two workers. Table II lists the software configuration.



(a) Topology 1: Category trend monitoring



(b) Topology 2: Customer recommendation



(c) Topology 3: Risk control

Fig. 7. Representative benchmark topologies

TABLE II
Software configuration

Software	Version
OS	Ubuntu 24.04.2 64bit
Storm	Apache-Storm-2.8.0
Zookeeper	Zookeeper-3.9.3
JDK	Jdk-21.0.2 64bit
MySQL	mysql-server-8.0
Kafka	Apache-Kafka-3.8.0

TABLE III
Key Parameters

Parameter	Value
H	15
K_h	100
τ_L	100 ms
$A_{\min}$	0.95
$\rho_{\max}$	0.05
$\mathcal{Q}$	{1,2,4,8}
τ_{Γ}	1.8

We construct a multi-topology workload using three representative topologies derived from the Taobao UserBehavior dataset [22]. The three topologies represent typical stream analytics scenarios, including category trend monitoring topology (Fig. 7(a), Topology 1), recommendation topology (Fig. 7(b), Topology 2), and risk control topology (Fig. 7(c), Topology 3). They contain different keyed stateful operators and branch structures, allowing us to evaluate recovery-lag mitigation under heterogeneous topology logic and skewed key distributions. Table III summarizes key parameters.

The basic runtime statistics are collected in fixed monitoring windows through Storm metrics and counters embedded in the topology. The input rate $\Lambda_i(w)$ is counted at the spout, the acknowledged throughput $\lambda_i(w)$ is counted from successfully completed tuples, and the end-to-end latency $L_i(w)$ is calculated from tuple timestamps recorded at the spout and sink bolt. The key fraction $f_{i,u,v}(k,w)$ and task input rate $x_{i,u,v,\ell}(w)$ on key-partitioned edges are reported by the *Virtual Router Bolt*. These statistics are recorded for each topology and used to calculate the recovery metrics in Section III-B. Resource utilization are sampled from each compute node and averaged over the cluster.

We compare Rt-Stream with representative baselines based

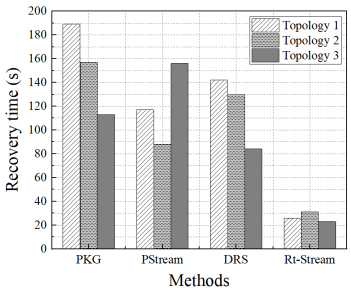


Fig. 8. Recovery time across methods

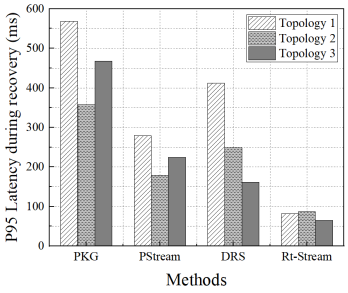


Fig. 9. P95 latency during recovery across methods

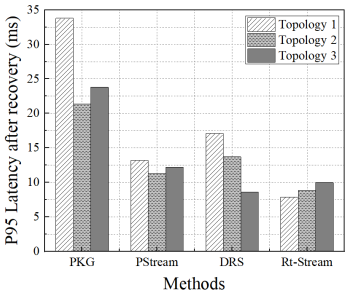


Fig. 10. P95 latency after recovery across methods

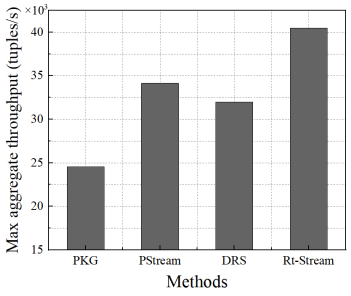


Fig. 11. Maximum aggregate throughput

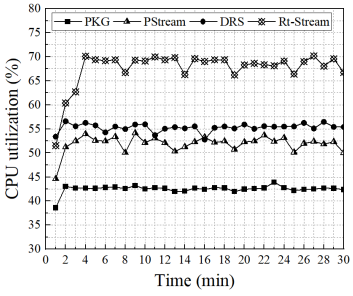


Fig. 12. CPU utilization over time

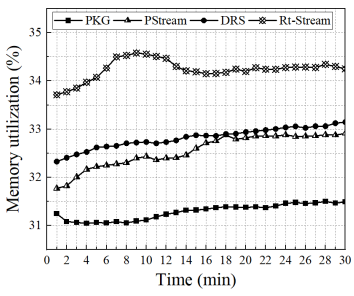


Fig. 13. Memory utilization over time

on grouping and scaling. PKG [7] and PStream [13] represent skew-aware grouping strategies that improve load balance under skewed streams. DRS [17] represents an elastic scaling method and is included to examine whether physical reconfiguration alone is sufficient for mitigating post-surge recovery lags. All methods are evaluated under the same cluster configuration and workload patterns.

We use six metrics to evaluate recovery effectiveness and resource utilization. *Recovery time* (Eq. (12)) measures the time required for a topology to return to the healthy region after a surge. *P95 latencies* [21] during and after recovery are the 95th percentile end-to-end latencies measured before and after recovery confirmation, respectively. *Maximum aggregate throughput* measures the largest total input rate that the co-running topologies can sustain without long-term degradation. *CPU and memory utilization* are used to analyze whether the system effectively uses cluster resources during execution [23].

B. Recovery-Time Reduction

Rt-Stream achieves the shortest recovery time among all evaluated methods. As shown in Fig. 8, Rt-Stream consistently recovers faster than PKG, PStream, and DRS on the three co-running topologies. Compared with these baselines, Rt-Stream reduces the recovery time by 64.8%–77.8%.

This result indicates that post-surge recovery lags cannot be fully addressed by load balancing or resource adaptation alone. PKG and PStream improve key distribution under skewed streams, but their decisions are not driven by recovery signals. DRS can adjust execution resources, but its reconfiguration is less direct for short recovery periods dominated by localized backlog caused by selected high-frequency keys. Rt-Stream instead applies grouping intervention only when recovery symptoms and task imbalance on key-partitioned edges appear together, enabling faster recovery with limited disturbance.

C. Latency and Residual Degradation

Rt-Stream achieves the lowest P95 latency during recovery. As shown in Fig. 9, Rt-Stream achieves the lowest P95 latency during recovery on all three topologies, reducing it by 51.6%–70.6% compared with these baselines. This improvement shows that Rt-Stream not only shortens the recovery duration, but also lowers the tail latency experienced during the degraded recovery period.

Fig. 10 further shows that Rt-Stream does not introduce persistent latency degradation after recovery. The post-recovery P95 latency of Rt-Stream remains comparable to the baselines on all three topologies and stays far below the latency threshold $\tau_L = 100$ ms. This indicates that the temporary virtual-key grouping used during recovery does not keep the topology in a degraded latency state after recovery. The reason is that Rt-Stream splits only selected high-frequency keys and manages split rules through refresh and expiration. Once recovery pressure disappears, inactive rules expire automatically, preventing temporary grouping intervention from becoming a permanent overhead.

D. System Throughput

Rt-Stream achieves the highest maximum aggregate throughput under bursty workloads. As shown in Fig. 11, Rt-Stream sustains approximately 40,500 tuples/s, improving the maximum aggregate throughput by 18.7%–65.0% over all baselines. This result indicates that Rt-Stream improves the

system's ability to absorb input surges. PKG and PStream may still leave part of the operator capacity underutilized, and DRS may react too coarsely for localized keyed bottlenecks. Rt-Stream directly redistributes tuples from selected high-frequency keys over existing task instances, which improves the effective use of existing task capacity without relying on physical reconfiguration.

E. Resource Utilization

Rt-Stream achieves more effective resource utilization during execution. As shown in Figs. 12 and 13, Rt-Stream achieves higher CPU and memory utilization than the baselines, indicating that the allocated resources are used more effectively during post-surge recovery. For CPU utilization, Rt-Stream reduces local bottlenecks caused by high-frequency keys and enables more downstream tasks to participate in useful processing. For memory utilization, virtual-key routing distributes tuple processing, input queues, and mergeable partial states across more task instances instead of concentrating them on a few overloaded tasks. Therefore, the higher resource utilization reflects better use of existing task capacity, rather than additional physical resource consumption.

VI. Conclusion and Future Work

This paper studies post-surge recovery lags in distributed stream computing systems. We propose Rt-Stream, a recovery-aware and low-disturbance virtual-key grouping framework that identifies recovery-risk topologies, diagnoses task imbalance on key-partitioned edges, and redistributes tuples of selected high-frequency keys with limited disturbance. By deriving routing keys from original keys and managing split rules through refresh and expiration, Rt-Stream mitigates recovery lags without high-cost runtime task migration or operator reconfiguration.

We implement Rt-Stream on Apache Storm and evaluate it with multi-topology workloads. Rt-Stream outperforms representative grouping- and scaling-based baselines by shortening recovery time, reducing recovery P95 latency, avoiding persistent post-recovery latency degradation, improving maximum aggregate throughput and resource utilization.

Our future work will explore recovery-lag mitigation for large-model-native distributed stream computing systems, where streaming inference requests, tool calls, and agent workflows may introduce new forms of skewness, burstiness, and recovery lags.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 62372419; the Fundamental Research Funds for the Central Universities of China under Grant No.265QZ2021001.

References

[1] Apache, "Apache Storm," 2026, project website. [Online]. Available: https://storm.apache.org/.

[2] Apache, "Apache Heron," 2026, project website. [Online]. Available: https://heron.apache.org/.

[3] Apache, "Apache Flink," 2026, project website. [Online]. Available: ht tps://flink.apache.org/.

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

[5] Y. Mao, J. Zhao, S. Zhang, H. Liu, and V. Markl, "MorphStream: Adaptive scheduling for scalable transactional stream processing on multicores," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[6] G. Van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020.

[7] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini, "Partial key grouping: Load-balanced partitioning of distributed streams," *arXiv preprint arXiv:1510.07623*, 2015.

[8] J. Tan, L. Yang, Y. Guo, Z. Zuo, and X. Xiao, "A distributed skewed stream processing system based on scoring high-frequency key perception: J. tan et al." *The Journal of Supercomputing*, vol. 81, no. 9, p. 1073, 2025.

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

[10] Q. Shi, X. Li, H. Zheng, T. Yang, Y. Wang, and M. Xu, "Heavylocker: Lock heavy hitters in distributed data streams," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, 2025, pp. 1244–1255.

[11] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024.

[12] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2015, pp. 797–802.

[13] 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, 2020.

[14] D. Sun, Z. Chen, W. Lv, S. Gao, and J. Rong, "A frequency-aware grouping strategy for stateful operators in distributed stream processing systems," in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2023, pp. 126–131.

[15] J. Liu, R. B. Basat, L. De Wardt, H. Dai, and G. Chen, "Disco: A dynamically configurable sketch framework in skewed data streams," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 4801–4814.

[16] S. Chen, D. Zuo, and Z. Zhang, "Flexsp:(1+ β)-choice based flexible stream partitioning for stateful operators," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 732–741.

[17] 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 Transactions on networking*, vol. 25, no. 6, pp. 3338–3352, 2017.

[18] L. Zhang, W. Zheng, K. Zheng, H. Zhu, C. Li, and M. Guo, "Bayesian-driven automated scaling in stream computing with multiple qos targets," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 7, pp. 1251–1267, 2024.

[19] 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)*. IEEE, 2019, pp. 1042–1052.

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

[21] S. Liu, J. Weng, J. H. Wang, C. An, Y. Zhou, and J. Wang, "An adaptive online scheme for scheduling and resource enforcement in storm," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1373–1386, 2019.

[22] Alibaba Cloud Tianchi, "User behavior data from taobao for recommendation (userbehavior)," Tianchi Dataset, 2026. [Online]. Available: https://tianchi.aliyun.com/dataset/649

[23] X. Liu and R. Buyya, "D-storm: Dynamic resource-efficient scheduling of stream processing applications," in *2017 IEEE 23rd International conference on parallel and distributed systems (ICPADS)*. IEEE, 2017, pp. 485–492.