Dynamic Adaptive Fault-Tolerance in Stream Computing Systems under Resource Constraints

Zhaojun Wang[1], Dawei Sun[1]*, Xuan Zang[1], Atul Sajjanhar[2], and Rajkumar Buyya[3]

[1] School of Information Engineering, China University of Geosciences, Beijing, 100083, China
wangzhaojuncn@gmail.com; sundaweicn@cugb.edu.cn

[2] School of Information Technology, Deakin University, Waurn Ponds, Victoria, 3216, Australia
atul.sajjanhar@deakin.edu.au

[3] Quantum Cloud Computing and Distributed Systems (qCLOUDS) Lab, School of Computing and Information Systems, The University of Melbourne, Grattan Street, Parkville, Victoria, 3010, Australia
rbuyya@unimelb.edu.au

Abstract. In stream computing systems, fault tolerance and recovery efficiency during task execution are core elements for ensuring system performance. However, existing fault tolerance strategies often overemphasise global fault tolerance performance, leading to significant increases in resource overhead and failing to achieve an effective balance between resource costs and reliability. To address these issues, we propose a dynamic adaptive fault-tolerant strategy named Da-Stream. This paper addresses the following aspects: (1) The high resource costs associated with running both primary and backup copies simultaneously are analysed, and the impact of factors such as operator type, changes in data stream size, and the strength of upstream/downstream dependencies on the fault tolerance requirement level of operators is verified. (2) The stream computing resource model and operator fault tolerance requirement level model are established to evaluate node CPU and memory resource utilisation, satisfy the resource constraints of the fault tolerance strategy, and adaptively assess the fault tolerance requirement levels of different operators. (3) Da-Stream classifies strategies based on factors such as operator type, resource requirements, and dependencies, and dynamically adjusts the fault tolerance strategy at runtime by combining resource usage, fault prediction, and historical scores. (4) Experimental results show that compared with state-of-the-art methods, Da-Stream reduces fault recovery time by 24.3%, increases CPU utilisation by 18.7%, increases memory utilisation by 12.2%, and reduces processing latency by 36.8%.

Keywords: Adaptive fault tolerance strategy · Distributed stream computing · Proactive backup · Resource constraints.

1 Introduction

Stream computing systems are increasingly being applied in fields such as big data processing, real-time analysis, and financial risk control. In these scenarios, the continuity and accuracy of data are of critical importance. Therefore, the fault tolerance mechanism of stream computing systems has become a key factor in ensuring system stability and reliability. Apache Flink, a popular distributed stream computing framework, has been widely adopted in the industry due to its robust fault tolerance capabilities[1, 2]. In addition to Flink, other mainstream stream computing systems also provide their own fault tolerance mechanisms to ensure the reliability and continuity of data processing. Spark[3] achieves end-to-end semantics at least once through RDD lineage and checkpoint mechanisms, and can achieve exactly once processing through idempotent sinks. Storm[4] uses an ACK mechanism for message acknowledgement, while the Trident extension enables support for higher-level semantics. In summary, each system continues to optimise state persistence, efficient scheduling, and fault detection mechanisms, driving fault tolerance mechanisms from static to adaptive, low-overhead, and high-availability directions.

Although Flink's fault tolerance mechanism has been widely applied in practical scenarios, it still faces numerous challenges when dealing with complex stream computing tasks, including: frequent state preservation in high throughput, low-latency scenarios results in significant performance overhead, state management lacks scalability in large-scale tasks and is prone to becoming a bottleneck, complex data flow topology structures that make fault tolerance handling more difficult, the difficulty of balancing data consistency and processing latency, and low recovery efficiency, which requires global rollback and lengthy node recovery times.

Some researchers have explored fault-tolerance strategies in stream computing systems, which can be broadly categorised into active, passive, and hybrid approaches. Active fault tolerance[5–9] improves recovery efficiency by maintaining snapshots or backup copies during runtime, but often faces issues such as high resource overhead, poor scalability, and suboptimal replica placement. Passive fault tolerance[10, 11] relies on checkpoints or logs to restore the system state after a failure. While it reduces resource consumption, it still faces a trade-off between checkpoint frequency and performance, and its efficiency decreases under resource-constraints conditions. Hybrid approaches[12, 13] combine both strategies, using passive techniques under normal conditions and switching to active recovery upon failure. This helps reduce recovery latency but increases overall resource usage and lacks fine-grained replica configuration. To address these issues, recent improvements have focused on optimising checkpoint trigger conditions and intervals to reduce performance overhead, introducing incremental checkpoints and hierarchical storage to enhance state management scalability, and exploring new fault-tolerance models to better support reliability and consistency in complex topologies. More flexible consistency models have also been proposed to balance latency and correctness. To achieve faster recovery, region-based local restart mechanisms and enhanced active fault tolerance tech-

niques have been explored[14]. However, these methods still have shortcomings in addressing the high resource costs required for improved fault tolerance, slow recovery speeds for certain operations under faults, and the lack of adaptive mechanisms that can adjust strategies based on real-time resource conditions and operational state changes.

To address the limitations of static fault-tolerance mechanisms, recent studies [15–20] have proposed adaptive strategies that dynamically adjust fault-tolerance configurations based on system status. These methods aim to improve the balance between reliability and performance by modifying replica counts or checkpoint intervals in real time. However, most of them rely on predefined rules or model-driven adjustments that respond passively to system changes. For example, adjustments are typically triggered only after detecting specific patterns such as high load[15] or continuous failures[17]. While these approaches improve flexibility over static schemes, they often fail to account for rapid shifts in input rates, evolving operator states, or varying failure types. Moreover, some models, such as those based on reinforcement learning[16] or chaos engineering[19], introduce additional overhead or training complexity, limiting their practicality in real-time systems.

Therefore, fault tolerance strategies need to further consider changes in cluster environments and operator states while controlling resource costs in order to achieve low-cost, high-fault-tolerance performance and adaptive fault tolerance methods. To this end, this paper proposes an adaptive fault tolerance strategy under resource constraints. Our contributions are summarised as follows:

1. We verified the impact of operator types, data stream size, and upstream and downstream dependency strength on failure recovery and fault tolerance requirements, and analyzed the varying tolerance demands across different operators.
2. Nodes are classified according to fault tolerance levels: high-level nodes use an active snapshot-based fault tolerance mechanism and synchronise regularly, while other nodes use incremental checkpoints to achieve passive fault tolerance, balancing performance and resource usage.
3. The fault tolerance level of operators is dynamically adjusted based on predicted fault rates, data flows, and resource usage, and passive fault tolerance performance is optimised using a dynamic checkpoint algorithm.
4. Experimental results on the WordCount topology task show that, compared with existing fault-tolerant strategies, Da-Stream effectively improves the overall performance and resource utilisation efficiency of the system.

2 Observation and Motivation

2.1 Observation

To understand the differences in fault tolerance requirements across various operators in a stream processing system, we conducted experiments examining

the effects of operator type, data flow size, and upstream/downstream dependency strength on fault recovery performance. We used a Twitter user behavior dataset and implemented a Top-N streaming application, Top-N topology typically includes data source operators, window aggregation processing operators, Top sorting processing operators, and output operators. Data flows from the data source into the topology, where window operators group the data, perform statistical calculations and sorting based on keywords. Then, the first N elements are retrieved from the calculated data in order. Finally, the output operator sends the top N elements in terms of occurrence frequency to the external storage system.

Operator Type. Operators differ in their recovery behavior due to their functionality and internal state. Stateless operators such as `map` and `filter` recover quickly by simply restarting threads. In contrast, stateful operators like `window` and `aggregation` must reload large volumes of historical state data, resulting in significantly longer recovery times. Figure 1 shows that recovery times for stateful operators are noticeably higher, especially as the amount of retained state increases.

Data Flow Size. We simulated failures every 5 minutes under varying data rates (5k, 15k, 30k events/s) to assess how input stream volume affects recovery. As shown in Figure 2, higher data rates lead to more state accumulation and longer recovery times. This demonstrates that system performance deteriorates when data throughput rises but the fault tolerance strategy remains static.

Dependency Strength. We also evaluated how the degree of upstream and downstream dependencies affects recovery. Operators with stronger dependencies i.e., those that rely on or feed multiple other operators tend to recover more slowly due to the overhead of state synchronization and dependency management. Figure 3 shows that Operator C (strong dependency) has the longest recovery time, followed by B (moderate), and A (weak). Recovery time for strongly dependent operators also increases more sharply over time.

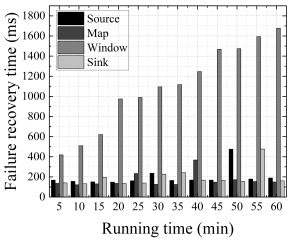

Fig. 1: Analysis of different operator type.

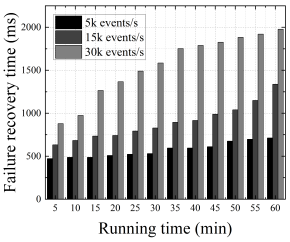

Fig. 2: Analysis of different data flow sizes.

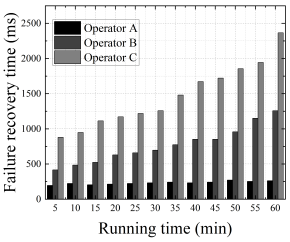

Fig. 3: Analysis of dependency strength

2.2 Motivation

These observations highlight significant variability in operator recovery behavior based on statefulness, input data volume, and topological dependencies. Traditional fault tolerance strategies often treat all operators uniformly or rely on static configurations, failing to account for such differences.

Operators with large state or strong dependencies not only recover more slowly but also exert more pressure on the fault tolerance subsystem, meaning failures in these operators have greater impact on system performance. Similarly, operators processing high-throughput streams face faster state growth, making them more vulnerable to failure and harder to recover.

Therefore, generic fault tolerance methods are inefficient, they either waste resources on low-impact operators or fail to adequately protect critical operators. These findings have led us to propose an adaptive fault tolerance strategy that dynamically adjusts priorities based on operator type, runtime state size, data flow rate, and dependencies. Such a strategy will simultaneously improve recovery efficiency and overall resource utilisation.

3 System Models

We constructed an optimisation model based on the issues identified in the above analysis, including a stream computing fault tolerance model, a resource model, and an operator fault tolerance requirement level model.

3.1 Stream Computing Fault Tolerance Model

During the fault tolerance phase, we optimised the Flink architecture by combining active and passive fault tolerance strategies, adding fault prediction, resource management, priority backup queues, and dynamic adjustment modules. The fault prediction module uses historical fault data for global prediction, dynamically adjusting checkpoint intervals and replica counts. The resource management module evaluates and maintains idle resources. The priority backup queue sorts replica order based on fault tolerance levels. The dynamic adjustment module combines prediction results, node status, and resource conditions to optimise fault tolerance strategies, balancing system latency and fault tolerance. Fault prediction employs a linear regression model, with the process comprising three stages: data preprocessing, model training, and fault prediction. First, input data is standardised and time-sorted, with key features extracted for training. Subsequently, a prediction function is constructed, which can be described as (1).

$$y = W_1X_1 + W_2X_2 + \cdots + W_nX_n + b + \epsilon, \tag{1}$$

where X_i are the feature variables, W_i are their corresponding weights, b is the bias term, ϵ is the error term, and y represents the predicted failure probability of the operator. The model is trained on historical data and optimized using gradient descent to improve prediction accuracy.

3.2 Resource Model

The rationality of resource management and scheduling directly affects the fault tolerance performance and resource utilization efficiency of stream computing systems. Therefore, we propose a resource-scoring-based model to optimise resource allocation under resource constraints and ensure the fault tolerance requirements of critical nodes. The system manages available resource nodes and ranks them by utilisation rate to ensure efficient scheduling. The resource node set is $R = \{R_1, R_2, \cdots, R_n\}$, and each node has four types of resource attributes: CPU, memory, disk I/O, and network bandwidth. The idle rate is calculated as follows:

$$cpu\_idle_i = \frac{cpu^i_{total} - cpu^i_{used}}{cpu^i_{total}}, \tag{2}$$

$$mem\_idle_i = \frac{mem^i_{total} - mem^i_{used}}{mem^i_{total}}, \tag{3}$$

$$disk\_idle_i = 1 - \frac{D^i_{throughput}}{D^i_{max\_throughput}}, \tag{4}$$

$$bw\_idle_i = 1 - \frac{D^i_{used}}{D^i_{total}}. \tag{5}$$

Considering the dynamic fluctuations in resources, we introduce a weighted decay mechanism to give higher weight to recent resource usage. Resource scores are calculated as follows:

$$\begin{aligned} Q_i(t) = \sum_{k=0}^{n} e^{-\lambda k} \Big(& w_1 \cdot cpu\_idle_{i,k} + w_2 \cdot mem\_idle_{i,k} \\ & + w_3 \cdot disk\_idle_{i,k} + w_4 \cdot bw\_idle_{i,k} \Big), \end{aligned} \tag{6}$$

where λ controls the degree of influence of historical data, and w_1,w_2,w_3,w_4 is the resource weight parameter. Nodes are sorted according to the $Q_i(t)$ value, and nodes with high scores are prioritised for allocation to the main task. In addition, to evaluate resource utilisation efficiency, we define the total CPU resource C and memory resource M of the cluster, where the CPU of machine R_i in the cluster is represented as CPU^i_{total} and the memory is represented as MEM^i_{total}. We also define the CPU utilisation rate $Rate_{cpu}$ and memory utilisation rate $Rate_{mem}$, as follows:

$$C = \sum_{i=1}^{n} CPU^i_{total}, \tag{7}$$

$$M = \sum_{i=1}^{n} MEM^i_{total}, \tag{8}$$

$$Rate_{cpu} = \frac{\sum_{i=1}^{n} CPU^i_{used}}{C}, \tag{9}$$

$$Rate_{mem} = \frac{\sum_{i=1}^{n} MEM_{used}^{i}}{M}. \tag{10}$$

3.3 Operator Fault Tolerance Level Model

Traditional static single fault tolerance methods are difficult to adapt to the diverse fault tolerance requirements and dynamic changes of different operators during task execution. To address this issue, we propose an operator fault tolerance requirement hierarchy model that comprehensively assesses fault tolerance priorities based on factors such as operator type, resource consumption, and upstream and downstream dependencies, thereby improving the overall fault tolerance of the system. Operators are divided into two categories: computation-intensive and data-intensive. The former has higher requirements for CPU and memory and relatively higher fault tolerance levels, while the latter has lower computational costs and weaker fault tolerance requirements. To quantify resource usage, the resource score R_i is defined as follows:

$$R_i = \alpha \cdot \frac{d_{CPU}^{i}}{C} + \beta \cdot \frac{d_{MEM}^{i}}{M}. \tag{11}$$

In addition, different operator nodes have different upstream and downstream dependency strengths in the topology, which affect the scope and cost of fault recovery. The dependency strength L_i of a node is defined as the ratio of its upstream and downstream dependencies to the maximum number of dependencies in the task:

$$L_i = \frac{N_{pred}^{i} + N_{succ}^{i}}{\max\left(N_{pred}^{j} + N_{succ}^{j}\right)}. \tag{12}$$

Calculate the fault tolerance requirement level P_i of the operator based on the comprehensive operator type score S_i, resource score R_i, and dependency strength L_i:

$$P_i = w_1 S_i + w_2 R_i + w_3 L_i, \tag{13}$$

where S_i is assigned as follows according to the operator type:

$$S_i = \begin{cases} 0.1, & filteroperator, \\ 0.5, & aggregationoperator, \\ 0.8, & join/statefuloperator. \end{cases} \tag{14}$$

4 Da-Stream: Architecture and Algorithms

According to the above analysis, we propose Da-Stream for adaptive fault tolerance. Da-Stream refines the strategies for different operator groups from multiple dimensions, such as operator type, resource requirements, and upstream and downstream dependency strength. During operation, it also dynamically adjusts the fault tolerance strategies for various operators based on factors such as cluster resource utilisation, fault rate predictions, and historical scores.

4.1 System Architecture

Based on the Flink stream computing architecture, we designed and integrated four major modules: resource management, dynamic fault tolerance adjustment, fault prediction, and system monitoring, to enhance the system's reliability and adaptability, as shown in Figure 4. The resource management module is responsible for real-time monitoring of cluster resource usage and optimises replica allocation strategies accordingly. The dynamic fault tolerance adjustment module dynamically assesses the fault tolerance level of operators and adjusts fault tolerance strategies based on task importance, resource consumption, dependencies, and predicted fault rates, the fault prediction module uses a linear regression model to estimate node fault probabilities, providing decision support for fault tolerance mechanisms. The monitoring module continuously tracks system runtime status, including resource usage, data flow, and slow tasks, and quickly detects node anomalies through a heartbeat mechanism, triggering fault recovery processes. Additionally, the system records historical operational data for subsequent analysis.

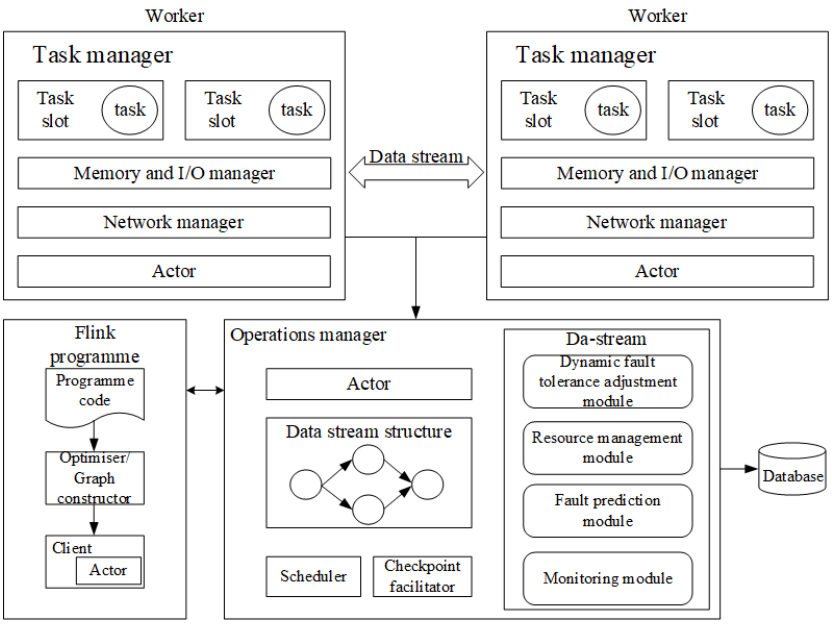

Fig. 4: Architecture of Da-Stream

4.2 Fault Tolerant Methods under Resource Constraints

Traditional active fault tolerance ensures system continuity by configuring primary and backup replicas for operators and quickly switching between them in

the event of a failure. However, this method incurs significant resource overhead, as the primary and backup replicas must process data simultaneously, increasing communication costs and downstream deduplication pressure.

First, drawing inspiration from the checkpoint strategy of passive fault tolerance, we introduce an incremental checkpoint mechanism, periodically persisting task state snapshots to RocksDB. Compared to full checkpoints, the incremental approach effectively reduces data volume and write costs, particularly suitable for tasks with large state sizes and frequent updates.

Second, we utilize cluster resources to enable proactive fault tolerance for operators with stringent reliability requirements. Unlike conventional active-backup approaches, the standby replicas do not engage in real-time computation during normal operation. Instead, they establish upstream and downstream communication channels in advance and are promptly activated to take over tasks upon a node failure. This design effectively reduces runtime resource overhead while ensuring rapid recovery.

To ensure consistency between primary and backup states, the system adopts a chained state replication mechanism. After each checkpoint is completed, the primary replica synchronises the incremental state to the backup replica along the chained structure. Each node can serve as a disaster recovery backup for its preceding node and can directly take over the task in the event of a failure. After recovery, the original primary replica joins the end of the chain and continues synchronisation, forming a closed-loop structure that enhances system resilience and consistency.

Additionally, to avoid delays and data loss caused by global state rollbacks, a local replay mechanism based on offsets is designed. During normal operation, the upstream records the offset of sent data and caches unconfirmed data. After a failure, only data after the offset is replayed, reducing delays and ensuring synchronisation with the downstream state.

To prevent duplicate data during primary-standby switching, downstream operators must perform deduplication. We use a unique identifier method, where upstream data is sent with a globally unique UUID, and the primary-standby processing generates results with UUID-k. Downstream operators cache and compare UUIDs to achieve deduplication. To control cache usage, we introduce a FIFO replacement strategy to improve memory efficiency and processing accuracy.

4.3 Node Fault Tolerance Strategy: Initialisation and Online Adaptation

Initial Fault Tolerance Strategy The fault tolerance strategy for operator nodes needs to balance real-time performance, reliability, and resource efficiency. Therefore, we propose a node fault tolerance strategy classification method based on multi-dimensional feature perception. By quantifying operator importance, resource requirements, and upstream and downstream dependency strength, we initialise the classification of operator nodes for fault tolerance strategies, dividing them into active fault tolerance and passive fault tolerance nodes. Input

all task nodes in the topology $O = \{O_1, O_2, \ldots, O_n\}$, For each task node O_i in O, calculate the backup priority of that node according to formula (13), sort the set O in descending order, iterate through the sorted set, and if the backup priority score exceeds the predefined threshold, update its backup strategy to active backup and add the task node to the active backup node queue. Otherwise, update its backup strategy to passive backup, and finally return the active backup node queue T_{active}.

After the node fault tolerance strategy classification is completed, the system enters the fault tolerance strategy initialisation phase. Fault tolerance strategy initialisation is divided into active fault tolerance initialisation and passive fault tolerance initialisation. For active fault tolerance initialisation, the core processes include resource assessment, replica allocation, and status synchronisation, with specific details described in Algorithm 1. In step 1, we sort the activity fault-tolerant node queue in descending order. In steps 2-5, for each resource node R_i in the resource node set R, we calculate its resource score using formula (6) and then sort them in descending order. Subsequently, for each task node O_i in $T_{sourted}$, In steps 7-10, when the resources exceed the constraints, the replica allocation stops, and other tasks are updated to passive standby replicas. In steps 11-16, we assign the resource node with the highest resource score as its active standby copy. Tasks with higher priority have priority access to high-quality resource nodes. A checkpoint mechanism is adopted to achieve fault tolerance and prevent cluster resource overload.

Algorithm 1 Active Backup Initialization

Input: Active backup node list T_{active}, Resource node list R
Output: Active backup mapping table B

1: Sort T_{active} by P_i descending $\rightarrow T_{sorted}$;
2: **for** each $R_i \in R$ **do**
3: Calculate resource score Q_i by Equation (6);
4: **end for**
5: Sort R by Q_i descending $\rightarrow R_{sorted}$;
6: **for** each task node $O_i \in T_{sorted}$ **do**
7: **if** Remaining resources in $R_{sorted} < AllClusterResource \times 20\%$ **then**
8: *//Insufficient resources, stop allocating*
9: **break**;
10: **end if**
11: $R_j \leftarrow R_{sorted}.peek()$;
12: assignBackupResource(O_i, R_j);
13: $B[O_i] \leftarrow R_j$;
14: *//Synchronize master and replica status*
15: syncStateWithPrimaryBackup(O_i^1, O_i^2);
16: $R_{sorted}.poll(R_j)$;
17: **end for**
18: **return** B

For passive fault tolerance initialisation using the checkpoint mechanism after passive backup nodes are classified by nodes, the initialisation of the checkpoint

interval C_i needs to ensure fault tolerance while maximising execution efficiency. We assume that the task execution cycle T is the total time required to complete the task, which includes the task execution time T_{task}, the checkpoint overhead time $T_{checkpoints}$, and the fault recovery time $T_{recover}$. as shown in equation (15):

$$T_{total} = T_{task} + n_{checkpoints} \cdot T_{checkpoints} + fr \cdot T_{recover}, \tag{15}$$

where fr is the fault recovery probability, and $n_{checkpoints}$ is the number of checkpoints in the task cycle T. Due to the occurrence of checkpoint cycles, it can be expressed as $n_{checkpoints} = \frac{T}{C}$. Task execution efficiency can be defined as the ratio of task execution time to total task time,as described in (16).

$$f(C) = \frac{T_{task}}{T_{total}}. \tag{16}$$

To maximise execution efficiency, we differentiate formula (16) to obtain $f'(C) = 0$. Since $\frac{T}{C^2}$ cannot be zero, therefore, maximising execution efficiency involves minimising the impact of checkpoint overhead and fault recovery time on execution time within the total time. This can be achieved by balancing the checkpoint overhead time and fault recovery time to find the optimal initial checkpoint interval $C_{initial}$, which can be calculated using formula (17).

$$C_{initial} = \frac{T \cdot T_{checkpoints}}{fr \cdot T_{recover}}. \tag{17}$$

Through this derivation, the optimal initial checkpoint interval $C_{initial}$ is obtained to maximise the execution efficiency during task initialisation, thereby completing the initialisation of passive fault tolerance.

Online Fault Tolerance Mechanism The probability of a system failure recurring within a short period of time after an initial failure is high, and the resource status of nodes (such as CPU and memory) also affects operational efficiency and the risk of failure. Therefore, we propose a dynamic backup priority strategy that dynamically adjusts the fault tolerance mode based on the resource usage, failure probability, and data traffic of operators.The system monitors the resource status and failure probability F_i of each operator node in real time. When resource utilisation or F_i is high, the backup priority is increased and the number of replicas is increased. If the resource pressure is low or F_i is low, the number of replicas can be reduced or the checkpoint interval can be extended. In addition, for high-traffic operators, the backup priority should also be increased due to higher recovery costs.Based on this, we determine whether to adopt active fault tolerance, passive fault tolerance, or adjust the checkpoint frequency based on whether the new and old priorities exceed the threshold. The updated priority queue is then output for scheduling in the next cycle.

For adaptive adjustment of fault tolerance strategies, the system regularly evaluates and ranks resource nodes, prioritises the allocation of replicas to high-priority operators that require active backup, and completes status synchronisation and upstream and downstream connections to ensure that they can be

switched at any time. For other operators, the checkpoint interval is dynamically adjusted based on resource utilisation, fault prediction rate, and the number of slow tasks to optimise fault tolerance overhead and execution efficiency.To balance fault tolerance performance and resource consumption, the system reserves some idle resources for active fault tolerance and the rest for fault recovery.

Algorithm 2 Dynamic Active Backup Allocation

Require: Backup priority queue $P' = \{P'_1, P'_2, \ldots, P'_n\}$, dynamic adjustment coefficient α, idle resource node queue R', task nodes $O = \{O_1, O_2, \ldots, O_n\}$
Ensure: Updated backup strategy
1: **while** application not finished **do**
2: Update resource node queue R'
3: **for** each task node $O_i \in O$ **do**
4: **if** backup strategy is active backup **then**
5: **if** O_i has no backup **then**
6: addToResourceQueueO', O_{new}
7: **else**
8: **continue**
9: **end if**
10: **else**
11: $CI_n \leftarrow$ adjustCheckpointInterval$f_r, U_{\text{cpu}}, U_{\text{mem}}, N_{\text{slow}}, CI_{n-1}$
12: **end if**
13: **end for**
14: **for** each task node $O_i \in O_{\text{new}}$ with active backup **do**
15: **if** R' remaining resources $<$ AllClusterResource $\times$ 20% **then**
16: **break**
17: **end if**
18: $R_j \leftarrow R'$.peek()
19: assignBackupResourceP'_i, R_j
20: syncStateWithPrimaryBackupP'_i, R_j
21: R'.pollR_j
22: **end for**
23: updateCheckpointIntervalCI_n
24: waitForNextCycle()
25: **end while**
26: **return** Updated backup strategy

As shown in Algorithm 2. In steps 3-4, first iterate through all operation nodes to determine whether active fault tolerance is enabled. In steps 5-6, if no replica has been assigned, add it to the pending allocation queue. If passive fault tolerance is used, dynamically adjust its checkpoint settings in step 11. In steps 14-22, the system then allocates resources to pending nodes from the idle resource queue in priority order, giving priority to high-priority tasks. When available resources drop below 20% of total resources, allocation stops. After resource binding is complete, the primary and backup replicas synchronise their states

and establish communication. In steps 23-24, the system updates the checkpoint configuration and enters the next monitoring cycle, repeating the process until the application ends.

Adaptive adjustment for passive fault tolerance. The algorithm dynamically adjusts the checkpoint interval by sensing the task running status (including fault rate, CPU and memory usage, and the number of slow tasks) to optimise system performance while ensuring fault tolerance. The resource utilisation rate is equal to the ratio of the resources used by the node to process normal data to the total resources used by the running tasks, where the CPU utilisation rate and memory utilisation rate are as shown in Equations (18) and (19).

$$U_{cpu} = \frac{N_{cpu}}{T_{cpu}}, \tag{18}$$

$$U_{mem} = \frac{N_{mem}}{T_{mem}}. \tag{19}$$

At system startup, the initial interval CI_0 is used. During runtime, the interval is adjusted based on real-time status: when no faults occur, the interval is appropriately extended using formula (20) based on the predicted fault rate f_r. If a fault occurs, the interval is shortened using formula (21) to increase the recovery frequency.If CPU usage U_{cpu} exceeds the threshold C_{const}, the interval is increased according to formula (22) to reduce resource pressure. Similarly, if memory usage U_{mem} exceeds the threshold M_{const}, the interval is adjusted according to formula (23).If the number of slow tasks N_{slow} exceeds the threshold M, extend the interval using formula (24) to mitigate its impact on the system. Finally, the updated checkpoint interval CI_n is written to the system configuration for use in the next round of triggering.

$$\Delta in = CI_{n-1} \cdot (1 - f_r), \tag{20}$$

$$\Delta in = CI_{n-1} \cdot f_r, \tag{21}$$

$$CI_n = CI_{n-1} \cdot \left(\frac{U_{cpu}}{U_{const}} \right), \tag{22}$$

$$CI_n = CI_{n-1} \cdot \left(\frac{U_{mem}}{M_{const}} \right), \tag{23}$$

$$\Delta in = CI_{n-1} \cdot \left(\frac{N_{slow} - M}{M} \right). \tag{24}$$

5 Performance Evaluation

This section introduces the performance evaluation of the Da-Stream system. First, we introduce the experimental setup, and then analyse its effectiveness in terms of fault recovery time, resource utilisation, processing delay, and task execution time.

5.1 Experimental Setup

The Da-Stream system is based on the Flink stream computing framework and is deployed in a cluster environment running the CentOS 7 operating system. The cluster consists of 15 compute nodes, with 1 node deployed for the Flink JobManager, 2 nodes for HDFS and Zookeeper, and the remaining 12 nodes serving as NodeManagers. All nodes have the same hardware configuration, including a 2-core 2.4GHz CPU, 2GB of memory, and a 100Mbps Ethernet interface. We use the Twitter user behaviour dataset and adopt the classic stream computing topology structure WordCount to evaluate system performance. WordCount reads text from the data stream, counts the number of occurrences of each word, and finally prints the results.The topology of WordCount is shown in the Figure 5. In the experiment, we compared Da-Stream with Flink's checkpoint fault tolerance strategy and the current state-of-the-art design scheme A-FP4S[20]. It should be noted that while the experiments are conducted on the WordCount topology, Da-Stream is equally applicable to other streaming topologies.

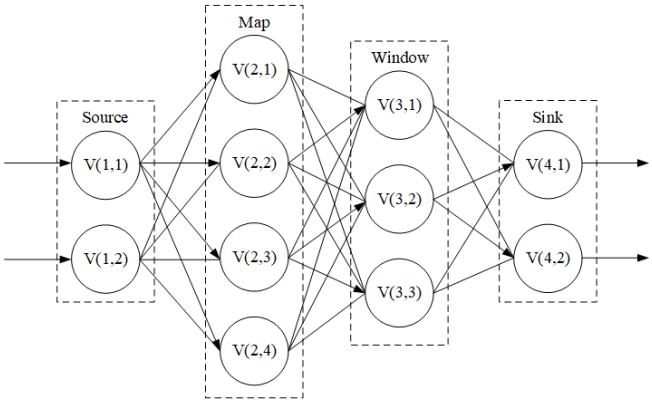

Fig. 5: The instance topology of WordCount

5.2 System Performance Analysis

To evaluate the performance of the fault tolerance strategy in a real operating environment, the experiment simulated failures in Flink jobs in a variety of ways, including using the kill command to forcefully terminate the TaskManager process to simulate node crashes, embedding conditional statements and actively throwing exceptions to simulate local failures during operator execution, and using tc and stress tools to simulate abnormal environments such as network packet loss, latency, or resource overload. In the experiment, fault events were triggered at predefined time points, and the system's response process and recovery duration were recorded using monitoring components such as Prometheus and Grafana, enabling a comprehensive evaluation of the fault tolerance strategy's

fault recovery capabilities. After considering implementation difficulty and control precision, the experiment ultimately adopted an exception method triggered by conditions embedded in custom operators for operator-level fault injection, simulating a fault every 5 minutes and starting the timer when the fault occurred until the task resumed normal operation to measure the fault recovery time. As shown in Figures 6, the results indicate that Da-Stream, which employs a low-overhead active fault tolerance mechanism, can achieve rapid switching during operator failures, with recovery speeds outperforming Flink checkpoint strategies and A-FP4S. In the WordCount topology, Da-Stream reduced the average recovery time by 24.3% compared to A-FP4S.

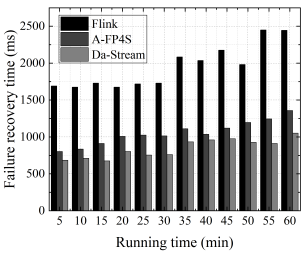

Fig. 6: Failure recovery time on WordCount.

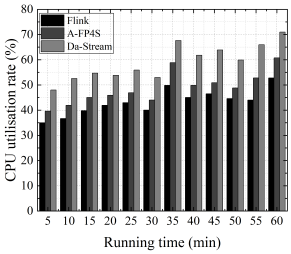

Fig. 7: CPU utilisation on WordCount.

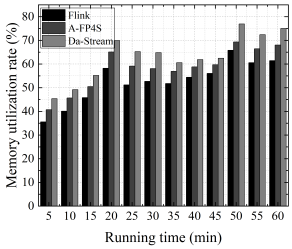

Fig. 8: Memory utilisation on WordCount.

In the Flink stream processing system, CPU and memory are critical resources for task execution, directly affecting system energy consumption and operational costs. To evaluate the resource utilization efficiency of different fault tolerance strategies, we compared the CPU and memory utilization of Da-Stream with Flink's checkpoint mechanism and A-FP4S under normal task execution conditions. As shown in Figures 7-8, the experimental results indicate that the CPU utilisation of the three strategies increases with system load, enters a stable phase, and then fluctuates, with short-term peaks occurring during fault recovery or when computational complexity increases. Since Da-Stream configures hot backup replicas for critical operators and utilises idle resources to synchronise states, its CPU utilisation is overall higher than that of the other two strategies.In terms of memory, Da-Stream periodically persists and synchronises the state of high-fault-tolerance operators to backup copies, resulting in consistently high memory utilisation. Therefore, Da-Stream achieves active fault tolerance through asynchronous state synchronisation, effectively reducing the costs associated with resource redundancy and improving the overall resource utilisation of the cluster. Compared to the A-FP4S fault tolerance strategy, the CPU utilisation and memory utilisation of the Da-Stream fault tolerance strategy are 18.7% and 12.2% higher, respectively.

Processing latency primarily refers to the time from data generation to completion of processing, encompassing all stages such as data transmission, computation, storage, and state updates. The experimental results are shown in Figures 9. Initially, the data stream rate was set to 5,000 tuples/s to simulate a low-load

scenario. After 30 minutes, the rate was increased to 10,000 tuples/s. During the process, simulated failures were input once every ten minutes. Due to the impact of Flink failures on data processing, processing latency increased. Da-Stream used active fault tolerance to achieve fast switching under failure conditions and combined a dynamic adjustment mechanism to maintain low processing latency from start to finish. Compared to Flink, Da-Stream reduced processing latency by 52.3% and compared to A-FP4S, it reduced processing latency by 36.8%.

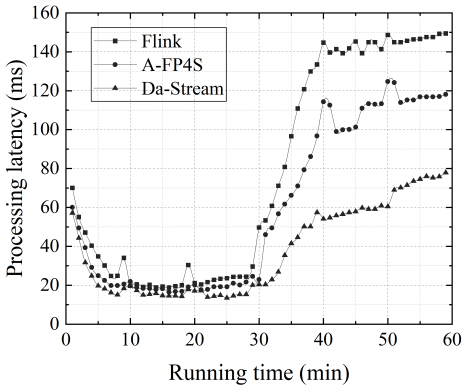

Fig. 9: Processing latency on WordCount.

6 Conclusion

To address the shortcomings of existing fault-tolerant strategies in resource-constraints environments, this paper proposes an adaptive fault-tolerant strategy called Da-Stream. This strategy comprehensively considers task importance, resource requirements, and the strength of dependencies between operators to calculate the fault-tolerance requirement levels of each operator and classify them into different fault-tolerance modes. By introducing a time-aware resource model, it dynamically evaluates the idle resources in the cluster and prioritises active fault tolerance support for operators with high fault-tolerance requirements, while the remaining operators adopt a checkpoint mechanism. Additionally, Da-Stream incorporates an adaptive adjustment mechanism that can update the fault tolerance strategy in real time based on changes in resource usage and operator characteristics, enabling flexible scheduling of resources and replica nodes. This method effectively reduces fault recovery time, lowers processing latency and execution overhead, and significantly improves resource utilization efficiency. Experiments on the WordCount topologies demonstrate that Da-Stream outperforms the existing A-FP4S strategy in both system performance and resource utilisation.

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

References

1. Li, H., Li, J., Duan, X., Xia, J.: Energy-aware scheduling and two-tier coordinated load balancing for streaming applications in apache flink. Future Generation Computer Systems **166**, 107681 (2025)
2. Yasser, T., Arafa, T., ElHelw, M., Awad, A.: Keyed watermarks: A fine-grained watermark generation for apache flink. Future Generation Computer Systems **169**, 107796 (2025)
3. Qi, H., Huang, Z., Chen, Y., Zhang, Y., Gao, Y.: Streamlining trajectory map-matching: a framework leveraging spark and gpu-based stream processing. International Journal of Geographical Information Science **38**(6), 1158–1178 (2024)
4. Zhang, Z., Jin, P.Q., Xie, X.K., Wang, X.L., Liu, R.C., Wan, S.H.: Online nonstop task management for storm-based distributed stream processing engines. Journal of Computer Science and Technology **39**(1), 116–138 (2024)
5. Isukapalli, S., Srirama, S.N.: A systematic survey on fault-tolerant solutions for distributed data analytics: Taxonomy, comparison, and future directions. Computer Science Review **53**, 100660 (2024)
6. Martin, A., Brito, A., Fetzer, C.: Low cost synchronization for actively replicated data streams. In: 2019 9th Latin-American Symposium on Dependable Computing (LADC). pp. 1–10. IEEE (2019)
7. Heinze, T., Zia, M., Krahn, R., Jerzak, Z., Fetzer, C.: An adaptive replication scheme for elastic data stream processing systems. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. pp. 150–161 (2015)
8. Li, H., Wu, J., Jiang, Z., Li, X., Wei, X.: Minimum backups for stream processing with recovery latency guarantees. IEEE Transactions on Reliability **66**(3), 783–794 (2017)
9. Bellavista, P., Corradi, A., Kotoulas, S., Reale, A.: Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In: EDBT. pp. 85–96 (2014)
10. Xu, C., Holzemer, M., Kaul, M., Markl, V.: Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE). pp. 613–624. IEEE (2016)
11. Jayasekara, S., Karunasekera, S., Harwood, A.: Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: theory to practice. Software: Practice and Experience **52**(1), 296–315 (2022)
12. Xu, H., Liu, P., Cruz-Diaz, S., Silva, D.D., Hu, L.: Sr3: Customizable recovery for stateful stream processing systems. In: Proceedings of the 21st International Middleware Conference. pp. 251–264 (2020)
13. Su, L., Zhou, Y.: Tolerating correlated failures in massively parallel stream processing engines. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE). pp. 517–528. IEEE (2016)
14. Cheng, Z., Tang, L., Huang, Q., Lee, P.P.: Enabling low-redundancy proactive fault tolerance for stream machine learning via erasure coding: Design and evaluation. Available at SSRN 4192493 (2024)

15. Zhuang, Y., Wei, X., Li, H., Wang, Y., He, X.: An optimal checkpointing model with online oci adjustment for stream processing applications. In: 2018 27th International Conference on Computer Communication and Networks (ICCCN). pp. 1–9. IEEE (2018)
16. Zhang, Z., Liu, T., Shu, Y., Chen, S., Liu, X.: Dynamic adaptive checkpoint mechanism for streaming applications based on reinforcement learning. In: 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS). pp. 538–545. IEEE (2023)
17. Jayasekara, S., Harwood, A., Karunasekera, S.: A utilization model for optimization of checkpoint intervals in distributed stream processing systems. Future Generation Computer Systems **110**, 68–79 (2020)
18. Cardellini, V., Nardelli, M., Luzi, D.: Elastic stateful stream processing in storm. In: 2016 International Conference on High Performance Computing & Simulation (HPCS). pp. 583–590. IEEE (2016)
19. Geldenhuys, M.K., Pfister, B.J., Scheinert, D., Thamsen, L., Kao, O.: Khaos: Dynamically optimizing checkpointing for dependable distributed stream processing. In: 2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS). pp. 553–561. IEEE (2022)
20. Xu, H., Liu, P., Ahmed, S.T., Da Silva, D., Hu, L.: Adaptive fragment-based parallel state recovery for stream processing systems. IEEE Transactions on Parallel and Distributed Systems **34**(8), 2464–2478 (2023)