Concurrency and Computation: Practice and Experience

WILEY

RESEARCH ARTICLE

Toward High-Availability Distributed Stream Computing Systems via Checkpoint Adaptation

Dawei Sun[1] | Jia Peng[1] | Ting Zhu[1] | Jonathan Kua[2] | Shang Gao[2] | Rajkumar Buyya[3]

[1]School of Information Engineering, China University of Geosciences, Beijing, China | [2]School of Information Technology, Deakin University, Melbourne, Australia | [3]Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

Correspondence: Dawei Sun (sundaweicn@cugb.edu.cn)

Received: 3 April 2024 | **Revised:** 19 May 2025 | **Accepted:** 7 June 2025

Funding: National Natural Science Foundation of China, Grant/Award Number: 62372419; Fundamental Research Funds for the Central Universities, Grant/Award Number: 265QZ2021001.

Keywords: checkpoint adaptation | distributed systems | high availability | mutifactor awareness | stream computing

ABSTRACT

The importance of fault tolerance strategies for distributed streaming computing systems becomes more evident due to the increased diversity of failures. Checkpointing is considered a general and efficient method for ensuring fault tolerance. However, determining the checkpoint interval poses a challenge: shorter checkpoint intervals lead to higher overhead, while longer intervals result in extended fault recovery time. Therefore, optimizing the checkpoint interval becomes crucial for the efficient operation of streaming applications. There has been relatively limited exploration and analysis of optimal checkpoint interval settings in the context of stream computing. Many existing works considered adjusting this interval based on a single factor. This article proposes a checkpoint adaptive strategy with high availability, named Ca-Stream. It considers multiple factors when adjusting checkpoint intervals. Specifically, it addresses the following aspects: (1) Using linear regression to predict the system's fault rate and dynamically adjusting the checkpoint interval based on these predictions. (2) Monitoring CPU time and memory consumption per task to dynamically trigger checkpoints, achieving high reliability, especially in resource-constrained scenarios. (3) Detecting task execution times on nodes and volume of input data for tasks to identify slow tasks within the cluster. Experiments conducted on a Flink system demonstrate Ca-Stream's benefits. It reduces checkpoint consumption time by over 38%, system recovery latency by 33%, CPU occupancy by up to 47%, and memory occupancy by 37% compared to Flink's approaches.

1 | Introduction

Distributed stream computing systems play an important role in extracting valuable insights from large volumes of real-time data streams. These systems find applications in diverse areas such as Internet of Things data processing, clickstream analysis, network monitoring, fraud detection, spam filtering, and news processing [1, 2]. In these areas, failures may have serious consequences, leading to highly unreliable results due to prolonged delays [3]. Adopting effective fault tolerance strategies is imperative to handle failures in a timely manner. Given the diversity of failure scenarios and types, it becomes crucial to embrace fault tolerance strategies with high availability.

Currently, the majority of stream computing systems primarily rely on replication recovery, checkpoint recovery, and Data-stream-based linear recovery to improve their fault tolerance [4]. Replication recovery enables fast fault recovery through

active backup task switching, but its resource consumption is nearly doubled [5]. In contrast, checkpoint recovery significantly improves system efficiency, offering a common fault-tolerant model in big data flow computing environments for wide range of applications. Checkpoint recovery economizes on resource costs by periodically checking the status of task processing. Data-stream-based linear recovery is slow and resource-intensive in geographically distributed network settings, especially in situations where lengthy linear graph are involved and the system lacks the capability to simultaneously address multiple faults. In the stream computing engines using checkpoint recovery, setting the optimal checkpoint interval is key to ensuring the efficiency of stream applications. The risk of longer checkpoint intervals comes at the cost of longer fault recovery time, while too short a checkpoint interval can significantly increase the additional overhead during fault-free operation.

While numerous checkpoint-based fault tolerance techniques are available, there is still a significant challenge in developing optimal checkpoint intervals for stream computing engines for the following reasons. First, existing fault tolerance mechanisms usually analogously and periodically trigger checkpoints, but failures do not occur periodically in real-life scenarios [6]. Therefore, determining how to set checkpoint intervals to effectively accommodate faults is a key question. The second challenge is that the frequency with which checkpoints are executed affects the system performance (e.g., utilization, latency), as the checkpointing process consumes resources and time that could otherwise be allocated to actual computation [7]. The third challenge is that the loss of tasks with longer execution times in a job is more likely to delay the job's completion [8].

To address the aforementioned challenges, we propose Ca-Stream, a strategy capable of adjusting the checkpoint interval based on multiple factors. We make the following contributions:

1. Using linear regression to predict the system's fault rate, and dynamically adjusting the checkpoint interval based on the prediction. When the predicted fault rate rises, a dynamically decreased checkpoint interval is triggered. When the predicted fault rate decreases, a dynamically increased checkpoint interval is initiated.
2. Monitoring the CPU time and memory consumed by each task to dynamically trigger checkpoints, achieving high reliability, especially in resource-constrained scenarios. When the CPU time or memory consumption exceeds a predefined threshold, the checkpoint interval is iteratively increased.
3. Detecting task execution times on nodes and volume of input data for tasks to identify slow tasks within the cluster. When the number of slow tasks exceeds a specified threshold, the checkpoint interval is increased, allowing the system to prioritize the execution of these slow tasks.

The subsequent sections are organized as follows: Section 2 presents a review of related work. Section 3 introduces the model construction of Ca-Stream. Section 4 presents the system architecture of Ca-Stream and the implementation details of the fault-tolerant algorithms. Section 5 reports experiments conducted to evaluate Ca-Stream's performance and compares it with Flink's existing checkpointing mechanism. Finally, Section 6 concludes this article.

2 | Related Work

In this section, we will summarize the existing fault-tolerance mechanisms of stream process frameworks and the existing high-availability strategies.

2.1 | Fault Tolerance Mechanisms in Stream Processing Frameworks

Currently, a multitude of stream processing frameworks exist that individually employ distinct fault-tolerance mechanisms to ensure the stability and reliability of their systems. These frameworks adopt varying strategies for error recovery and resilience, thereby catering to the critical nature of continuous data flow and real-time computation without compromising on performance or data integrity.

Spark Streaming is a common stream computing system that includes Spark SQL, MLlib [9], and Spark Streaming. Spark Streaming provides an advanced declarative API and supports languages such as Java, Python, and Scala [10]. It abstracts the processing logic of input streams into a series of resilient distributed datasets [11], which can be recomputed.

Storm [12] adopts checkpointing and source buffering for fault tolerance. Additionally, it utilizes a heartbeat mechanism to monitor the health states of nodes and employs a fault recovery and migration mechanism to ensure the continuity of data flow and processing.

Trident [13] serves as a higher-level abstraction for Storm. It simplifies the construction process of topologies and incorporates advanced operations such as aggregation, windowing, and state management. Trident employs anchoring techniques and batch processing to guarantee processing only once.

Samza was initially developed as a stream processing solution for LinkedIn [14], and contributed to the community alongside LinkedIn's Kafka [15, 16]. Samza achieves fault tolerance through incremental checkpointing and upstream backup. Furthermore, it utilizes a heartbeat mechanism and state monitoring to detect node health and adopts a task-switching mechanism to handle faulty nodes.

Flink et al. [17] rely on source buffering and consistent checkpoints to ensure processing only once. Additionally, Flink et al. employ periodic restarts and end-to-end timeout mechanisms to ensure system reliability.

MillWheel [18] also utilizes upstream backup. It assigns a globally unique ID to each input tuple, ensuring their precise processing and eliminating duplicates. This method incurs runtime overhead for maintaining upstream backups in a fault-free state.

2.2 | High Availability Strategies for Stream Computing Systems

2.2.1 | Fault Tolerance Methods

The fault tolerance methods for stream computing are usually divided into three categories: active backup, passive backup, and linear recovery based on data flow.

Active backup aims to reduce the impact of potential failures through predefined behaviors. In the process of active backup, a set of completely independent hot failover nodes [19] processes the same data stream in parallel with the primary nodes. Input data is transmitted to both sets. When one or more primary nodes fail, the system immediately switches to the set of secondary nodes [20]. Systems such as Flux use active backup for fault tolerance [4]. PLBFT, proposed by James, actively handles failures in cloud computing by balancing load fault tolerance [21].

In the process of passive fault tolerance, each node in the pipeline maintains a buffer in memory that stores a copy of records forwarded to downstream nodes since the last checkpoint. All nodes periodically transmit their state checkpoints to remote storage, such as Hadoop Distributed File System (HDFS) [22], while maintaining a set of backup nodes. Various systems have used passive replication, such as Trident [23] and TimeStream [4].

To achieve lower resource overhead and faster recovery, researchers have introduced linear recovery based on data streams, commonly used in Spark-based systems where the recent state is stored in the memory of each node [24]. Linear recovery based on data streams usually depends on the persistent storage of data stream logs, where all transactional operations that modify the data are recorded in a time-ordered sequence. By replaying these log entries, the system's linear recovery is achieved.

As active backup requires twice the resources of passive backup, passive backup is considered more resource-efficient. Linear recovery process based on data streams may be particularly slow. Therefore, passive backup emerges as a common fault-tolerant model in big data stream computing environments [25, 26]. Checkpoint-based fault tolerance mechanisms are often used in passive fault tolerance methods [27, 28].

When employing checkpoint-based fault tolerance methods, a key aspect of ensuring the efficiency of submitted stream applications lies in determining an optimal checkpoint interval. The optimization of checkpoint intervals has been widely studied in the field of high-performance computing [29, 30]. To address the high overhead caused by checkpoint operations, Parasyris proposed a differential checkpoint approach, which writes only the updated data to the checkpoint file [31]. Zhao implemented a differential incremental checkpoint optimization that records only state deltas and reduces global checkpoint frequency. However, it has overlooked the overhead of cross-node log synchronization in distributed environments. Furthermore, its transaction-consistent global checkpoints have struggled to balance strong data consistency with high availability [32]. A new checkpointing system that considers system-level power consumption, available memory and bandwidth, and checkpoint frequency, was introduced to make informed resource and data management decisions [33]. Kermarrec implemented a two-level checkpointing algorithm [34], where a memory checkpoint is efficiently established, and a persistent checkpoint is established at a much lower frequency but capable of tolerating permanent and power cut failures. Gossman proposed an I/O aggregation strategy for asynchronous multilevel checkpointing to achieve efficient fault tolerance, yet it fails to deeply study the resource competition between applications and background I/O threads [35]. Frank introduced a newly designed cost function to determine the optimal checkpoint interval. This algorithm takes into account several input parameters, including the number of nodes utilized by the application and the runtime of the application. Furthermore, this algorithm is capable of converging optimal results even for jobs with a high probability of failure [36]. Mushtaq presented a PBTS-FT model, which combines replication and checkpointing techniques, flexibly choosing to apply either technique separately or in tandem. This adaptability allows the system to enhance its fault tolerance based on the nature and circumstances of the failures. However, with the expansion of the system scale, the computational costs associated with maintaining and updating task priorities, as well as managing replication and checkpointing, are likely to increase substantially. This increase may exert a negative influence on the overall performance of the system [37].

Despite the advancements in checkpointing methods mentioned above, there has been relatively limited exploration and analysis of optimal checkpoint interval settings in the context of stream computing.

2.2.2 | Optimization of Checkpoint Intervals

Geldenhuys proposed an automatic analysis and runtime prediction method to model the performance and availability of distributed stream computing jobs. This approach is intended for scenarios where jobs handle static workloads, meaning the throughput remains constant over time [38]. The need to improve checkpoint intervals to enhance the system computing efficiency was emphasized by Zhuang [39]. Since different checkpointing policies should be applied in different scenarios, Marzouk proposed a set of preliminary rules that determine how and when to execute which checkpointing policy [40].

A checkpoint model called PANDA [8] was proposed to address the failure of memory data analysis frameworks. The proposed checkpoint model uses three checkpoint methods to initiate checkpoints, but it does not consider the size of the input data volume or enforce a checkpoint budget. Phoebe, an active method for automatically tuning distributed stream computing jobs on dynamic workloads, was proposed in [41]. This method can automatically adjust configuration parameters to ensure stable services and maintain consistency with the Quality of Service (QoS) goal of recovery time.

Gupta proposed a fault-tolerant approach of just-in-time checkpointing [42]. Only saves checkpoints for critical states and data, enabling error recovery by redoing at most one mini-batch of work. However, in the event of multi-node failures, relying solely

TABLE 1 | Comparison between Ca-Stream and related work.

Fault tolerance	FATM [6]	Khaos [45]	Panda [8]	Jayasekara [46]	Ca-Stream
Predictiton	✓	✓	×	×	✓
Dynamic	✓	✓	✓	×	✓
QoS aware	×	✓	×	×	✓
Vertex feature aware	×	×	✓	×	✓
Objective	Failure aware	QoS aware	Resource aware	Improving performance	Multiple factor aware
Limitation	Focus solely on failure rate	Fixed threshold	Focus solely on failure rate	Ignore dynamic workflows	Rely on historical data
Simulator	Unspecified	Flink	Custom environment	Theoretical model	Flink

on just-in-time checkpointing cannot fully address the issue. Although the paper mentions that low-frequency periodic checkpointing can be combined to handle such situations, it does not elaborate on how to efficiently coordinate these two checkpointing mechanisms in practical applications.

A dynamic checkpoint interval adjustment algorithm rooted in reinforcement learning was introduced by Zhang [43]. This algorithm dynamically fine-tunes checkpoint intervals by consistently gathering data on environmental state indicators and evaluating feedback from the environment. Jayasekara et al. [7] developed an expression to determine the checkpoint interval, aiming to minimize overhead during checkpoint establishment and optimize the interval.

The En-CHORE method, introduced by Sigdel et al., aims to decrease checkpointing overheads by selectively omitting specific checkpoints within each active sequence. This is conducted prior to identifying the optimal intercheckpoint interval for efficient checkpoint control, making it particularly well-suited for systems that accommodate prolonged task duration [44]. Akber introduced a failure-aware adaptive fault tolerance model (FATM), which improves the system utility factor by reducing the frequency of checkpoints and the additional workload associated with checkpoints [6].

MK Geldenhuys [45] introduced a new approach, Khaos, which focuses on automatic runtime optimization for fault-tolerant configurations in distributed stream processing jobs. Jayasekara [46] proposed an expression for system resource utilization and found the optimal checkpoint interval based on this expression. The method is especially applicable when system's size increases. Benoit utilizes a dynamic heuristic method of setting thresholds and a dynamic programming algorithm integrated with time discretization to ascertain the number and intervals of checkpoints. Nevertheless, due to the high complexity of the dynamic programming algorithm, it is challenging to acquire the optimal solution [47].

Based on the above discussion, it can be seen that existing optimization methods for checkpoint intervals do not simultaneously consider the distribution of failures and the impact of checkpoint intervals on system performance. They often overlook factors such as node computing capabilities and data transfer volume. In practice, it is beneficial to predict fault rates and set corresponding checkpoint intervals accordingly. Furthermore, monitoring various resource utilization metrics and dynamically adjusting checkpoint intervals based on resource utilization is necessary. Lastly, fine-tuning checkpoint intervals based on nodes' computing capabilities and tasks' data transfer volume is crucial. For example, tasks with lower computing capabilities better have longer checkpoint intervals to allow them to complete their computation as quickly as possible.

Considering all these factors, we propose a checkpoint adaption strategy with high availability, called Ca-Stream, for stream computing environments. Ca-Stream considers factors such as failure distribution, relevant failures, impact of checkpoints on performance, and number of slow tasks in the cluster. It suggests setting different checkpoint intervals and triggering checkpoints immediately in the event of failures based on varying fault rates, resource utilization, and number of slow tasks. This approach aims to minimize checkpointing overheads and system recovery latency as much as possible. The comparison between Ca-Stream and related work is presented in Table 1.

3 | Problem Statement

Current adjustment of checkpoint intervals lacks consideration for QoS. Using Flink as an example, its existing checkpoint-based fault tolerance mechanism fails to consider factors such as fault rates and the impact on system performance after setting checkpoints, as well as the number of slow tasks. Checkpoint interval settings should take these factors into account for the following reasons:

1. Actual system faults do not occur periodically, while checkpoints in the system are triggered periodically. This sharp contrast makes periodic checkpoints inefficient [6].
2. The frequency of triggering checkpoints can affect system performance, such as utilization and throughput, as this process consumes computing time and resources [18].
3. A significant number of slow tasks can impact program runtime, and setting frequent checkpoints for slow tasks will further affect their computing speed [8]. Longer checkpoint intervals for slow tasks can reduce the impact of checkpoint

TABLE 2 | Symbols and descriptions.

Symbols	Descriptions
$G(V, E)$	Directed acyclic graph of streaming job topology where V represents the set of vertices and E represents the set of edges
T_{chpt}	Total delay introduced by checkpoint operations on streaming computing systems
t_{chpt}	Average delay introduced by each checkpoint on streaming computing systems
t_{rs}	Tuple reprocessing time to restore operator state from the previous checkpoint to the state prior to the failure
t_{r}	Restart time from the operator state of the previous checkpoint
T_{norm}	Normal job processing time without checkpointing operations or failures
T_{rev}	Total time required for the system to recover from failure
ci_k	kth checkpoint interval corresponding to each task
CI_k	kth checkpoint cycle corresponding to each task
u_k	Real-time efficiency of the operator within the k_{th} time window

operations on resources and time, thereby accelerating program runtime.

Table 2 lists primary symbols and their descriptions used in this article.

3.1 | Qos-Aware Checkpoint Optimization Model

Our Ca-Stream integrates a QoS-aware checkpoint optimization model to ensure high system availability. This model consists of a database and a QoS-aware agent. Its structural detail is shown in Figure 1. The database stores historical fault events that are used as inputs to the QoS-aware agent. The agent consists of three modules: a fault-aware module, a resource-aware module, and a vertex feature-aware module. The fault-aware module within the agent utilizes historical fault events to predict future fault rates. This prediction is performed once at regular intervals for the entire cluster and is subsequently used to adjust checkpoint intervals during program operation. The information, including the fault rate predicted by the fault-aware module, the resource utilization perceived by the resource-aware module, and the task execution time and upstream data transfer volume perceived by the vertex feature-aware module, is transmitted to the QoS-aware checkpoint coordinator, which schedules the initiation of checkpoints.

This fault-aware module employs linear regression to predict the likelihood of potential failures and adjusts the prediction algorithm based on the evaluation of its predictive results. Figure 2 illustrates this prediction process. Fault rates are predicted in three stages: Data Preprocessing, Model Training, and Fault Prediction.

At the Data Preprocessing stage, the input dataset is first standardized and ordered sequentially. After that, appropriate features (e.g., resource utilization and task failure frequency) are extracted from the input dataset for prediction purposes.

At the Model Training stage, the corresponding prediction function is trained using the standardized training dataset. Assume the input dataset contains x records: $\{y_p, Z_p : 1 \le p \le x\}$, where y_p represents a feature vector, Z_p is the corresponding fault rate, and $Z_p \in \{0, 1\}$ [6]. The prediction function maps each feature vector to the corresponding failure event and predicts the fault probability.

At the Fault Prediction stage, the prediction function uses the trained linear regression model to forecast the future fault rates in upcoming time windows.

3.2 | Checkpoint Recovery Cost Model

The runtime of a stream processing job is considered to be infinite. As shown in Figure 3, the normal processing time T_{norm} of a job (without checkpointing operations or failures) can be seen as a series of discrete time intervals divided by periodic checkpoints. For simplicity, let the checkpoint interval within a given execution time T be a constant value, denoted ci. Checkpointing introduces an additional delay in normal logic processing, known as checkpoint execution time overhead. The total time spent setting up checkpoints T_{chpt} in normal processing time T_{norm} can be determined by multiplying the time spent setting up an individual checkpoint t_{chpt} by the number of checkpoints, represented by Equation (1),

$$T_{\text{chpt}} = t_{\text{chpt}} * \frac{T_{\text{norm}}}{T * ci} \tag{1}$$

The recovery time for each failure consists of two parts: the time to reprocess the tuples to restore the state of the operator from the saved state at the previous checkpoint to its state before failure, denoted as t_r, and the restart time from the state of the operator saved at the previous checkpoint, denoted as t_{rs} [48]. t_{rs} includes the time to detect the fault and the time to restore the operator state to the previous checkpoint. For the checkpoint cycle $CI_k = ci_k + t_{\text{chpt}}$, let the number of failures be $n(CI_k)$, then the time T_{rev} required to recover a failure within the time period T_{norm} can be calculated as Equation (2),

$$T_{\text{rev}} = n(CI_k) * (t_{\text{r}} + t_{\text{rs}}) * \frac{T_{\text{norm}}}{ci_k} \tag{2}$$

For each checkpoint cycle CI_k within the given computation segment T_k, we can now estimate the expected number of interruptions caused by failures.

Based on Daly's first-order model and assuming the exponential term is small, $ci_k + t_{\text{chpt}} \ll$ the mean time between failures (MTBF), we can use Equation (3) to estimate the expected number of failures $n(CI_k)$.

$$n(CI_k) \approx \frac{ci_k + t_{\text{chpt}}}{MTBF} \tag{3}$$

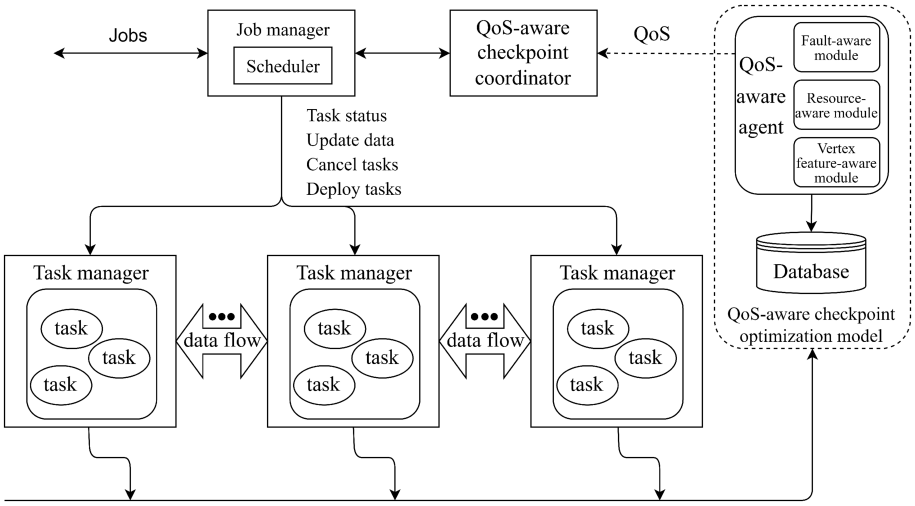

FIGURE 1 | Structure of the QoS-aware checkpoint optimization model.

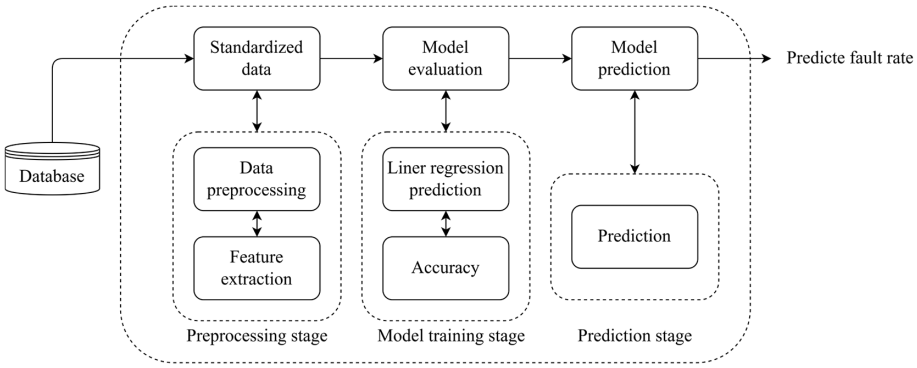

FIGURE 2 | Fault prediction pipeline of the fault-aware module.

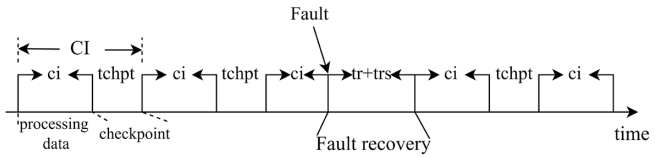

FIGURE 3 | Operator runtime model.

Substituting Equation (3) into Equation (2), we have Equation (4):

$$T_{\text{rev}} = \frac{ci + t_{\text{chpt}}}{MTBF} * (t_{\text{r}} + t_{\text{rs}}) * \frac{T_{\text{norm}}}{ci} \tag{4}$$

3.3 | Adaptive Checkpoint Interval Model

The checkpoint interval is a key point in achieving a balance between cost and recovery time during fault-tolerant execution. In scenarios with a given stable input data rate, a statically optimal checkpoint interval can significantly enhance the efficiency of operators throughout the entire runtime. However, in real-world scenarios, the arrival rate of streams is variable [49], and workload peaks can occur at any time. Due to the fluctuation of workload, the fault recovery overhead in terms of recovery time also varies. It is evident that periodic checkpoints may not always achieve the optimal balance. To achieve maximum processing efficiency, an adaptive optimal checkpoint interval is required to address this issue, namely the online optimal checkpoint interval problem in stream computing systems.

To address the challenge of handling continuous workload fluctuations, we propose a refined metric u for operator efficiency, known as the real-time efficiency of operators. As shown in Figure 4, the total execution time of a task is divided into a set of offline time segments denoted as $T_1, T_2, \ldots, T_k$. Time interval $[t_k, t_{k+1}]$ denotes the kth computation segment, during which the task maintain a stable checkpoint interval ci_k. Let CI_k denote the checkpoint cycle within the specified computation segment T_k. The real-time efficiency of operators whthin the time segment T_k can be measured by the ratio of the checkpoint interval as a percentage of the sum of the checkpoint cycle and the delay in recovering from a failure during that cycle, represented by Equation (5),

$$u_k = \frac{ci_k}{ci_k + t_{\text{chpt}} + n(CI_k) * (t_{\text{r}} + t_{\text{rs}})} \tag{5}$$

Based on this, the problem of determining the optimal checkpoint interval in the context of big data stream computing can

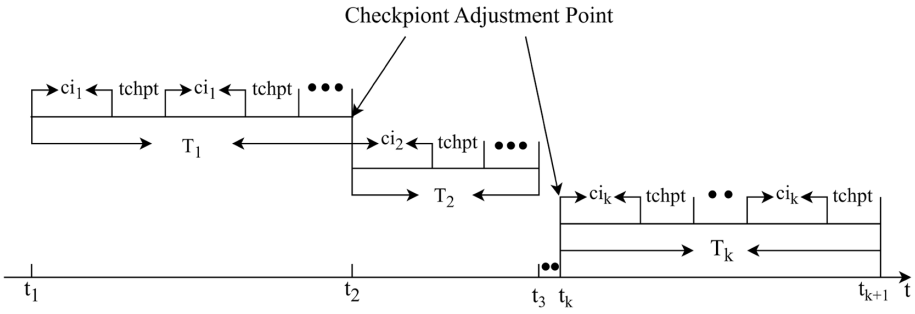

FIGURE 4 | Adaptive checkpoint interval model.

be further described as follows: Given a flow topology graph $G(V, E)$, where each operator v ($v \in V$) has a state, the goal is to find a checkpoint interval CI that maximizes the real-time efficiency of operators. As reprocessing tuple time t_r varies with the input rate, ci should be dynamically adjusted during task execution.

In this adaptive checkpoint interval model, the optimal checkpoint interval is considered as a real-time refinement of checkpoint intervals within each time segment T_k. The objective is to achieve a global optimum using a greedy algorithm for heuristic solutions, making a locally optimal choice for each time segment.

In general, a data stream application is modeled as a data flow graph, where vertices represent computational nodes (i.e., operators) and edges represent the data flow between these operators. Upon receiving data streams from upstream operators, each operator runs its specified processing logic and forward the data to downstream operators. As a result, every data tuple d ($d \in D$) traverses the data flow graph, forming a processing pipeline.

Checkpoint-based runtime fault-tolerance overhead encompasses the cost of setting checkpoints and the recovery cost during fault restoration. Increasing the checkpoint interval increases the recovery overhead while reducing checkpoint setting costs, and vice versa. Therefore, selecting an optimal checkpoint interval is crucial. Using $CI_1 \ldots CI_n$ to represent the 1st through nth checkpoint intervals, we try to find the best checkpoint interval CI_k for each discrete time segment. This can be achieved by first optimizing the checkpoint interval model based on the time-varying reprocessing time under changing workloads. We propose a method to rapidly find this optimal interval.

For a given task i within a computational segment T_k, the average time to failures within this checkpoint interval ci_k can be considered at half a checkpoint cycle. $tbeg^k$ is the opening time of the current checkpoint interval ci_k. Then, the expected failure occurrence time of the ci_k is the start time of the checkpoint cycle CI_k plus the failure occurrence time within ci_k, given as Equation (6),

$$tf^k = tbeg^k + \frac{ci_k + t_{\text{chpt}}}{2} \tag{6}$$

The data reprocessing volume after a failure can be modeled by Equation (7),

$$Wo^k = \int_{tbeg^k}^{tf^k} v(t)dt \tag{7}$$

where, $v(t)$ is the fluctuating input rate, and $tbeg^k$ represent the start time of current checkpoint interval ci_k, tf^k represent the expected failure occurrence time.

After that, we calculate the reprocessing tuple time t_r of checkpoint interval ci_k by dividing the data volume Wo^k by the maximum processing rate v_m of the operator.

$$tr(CI_k) = \frac{Wo^k}{v_m} \tag{8}$$

For simplicity, we use $\overline{v_k}$ to represent the average input rate over the time segment from t_k to t_{k+1}, then the volume of faulty data to be reprocessed can be determined by the average input time and average failure occurrence time(i.e., half of the checkpoint interval), expressed as $\overline{v_k} * (ci_k + t_{\text{chpt}})/2$. Substituting this value for Wo^k, the equation for checkpoint interval ci_k's recovery time t_r in Equation (8) can be replaced by Equation (9),

$$tr(ci_k) = \overline{v_k} * \frac{ci_k + thcpt}{2v_m} \tag{9}$$

Replacing $n(CI_k)$ and T_k with the recovery cost Equation (2), we have Equation (10),

$$T_{\text{rev}} = T_k * (ci_k + chpt) * \frac{t_r(ci_k) + t_{\text{rs}}}{ci_k + MTBF} \tag{10}$$

By substituting Equations (9) and (3) for $n(CI_k)$ and $tr(ci_k)$ in Equation (5), and taking the derivative of u_k, the optimal checkpoint interval can be obtained. For task i, the optimal checkpoint interval for the next duration $[t_k, t_{k+1}]$ with average input rate $\overline{v_k}$ is given by Equation (11),

$$(ci_k)^{opt} = \sqrt{2v_m * t_{\text{chpt}} * \frac{MTBF + t_{\text{rs}}}{\overline{v_k}} + t_{\text{chpt}}^2} \tag{11}$$

4 | Ca-Stream: Architecture and Checkpoint Mechanisms

4.1 | System Architecture

As shown in Figure 5, the proposed Ca-Stream includes a QoS-aware checkpoint optimization model to support the checkpoint adaptive strategy for high availability of distributed stream computing systems. Using Flink as our test bed, Ca-Stream integrates this optimization model into the architecture of

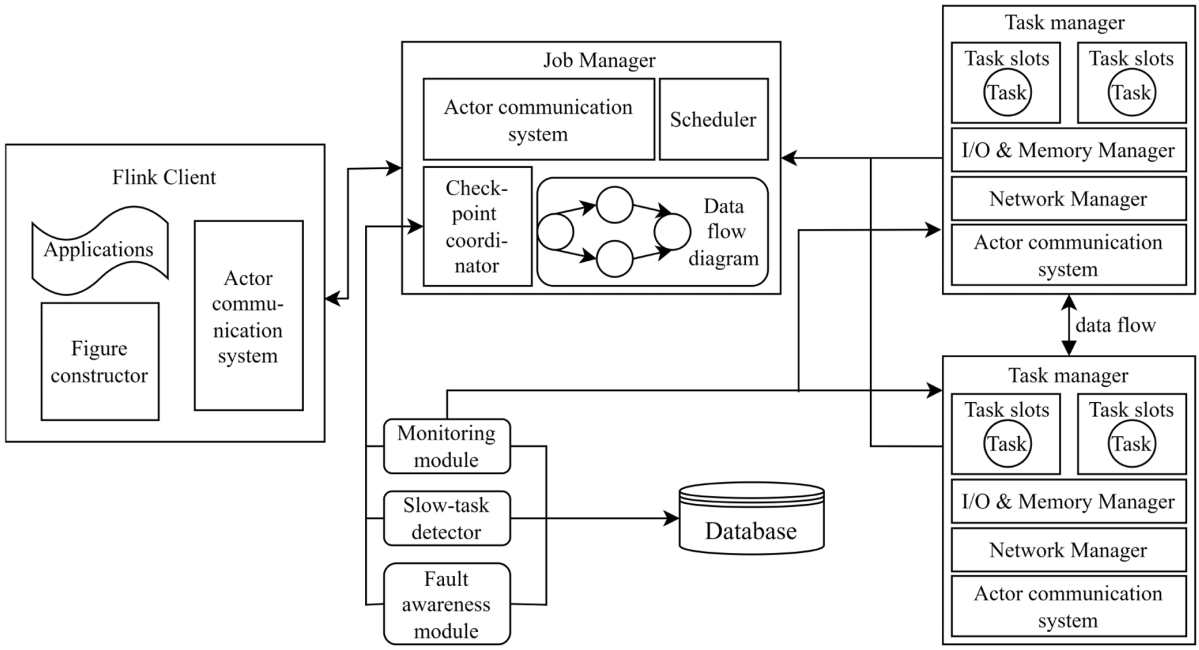

FIGURE 5 | System architecture of Ca-Stream.

Flink. The model consists of a database and a QoS-aware agent. The QoS-aware agent includes three modules: fault-aware, resource-aware, and vertex feature-aware.

The fault-aware module is used to predict the fault rate and send it to the QoS-aware agent. The QoS-aware agent dynamically adjusts the checkpoint interval based on the predicted fault rate it receives.

The resource-aware module is used to sense the real-time running status of stream processing nodes, specifically referring to real-time CPU and memory usage of tasks. This module typically employs a heartbeat mechanism to detect node's status.

The Vertex feature-aware module contains a slow task detector, which perceives the execution time and input data volume of tasks on nodes, thereby determining whether a task is considered slow. Only tasks with longer execution times and smaller input data volumes are classified as slow tasks.

The Database is used to temporarily store predicted fault rates, perceived CPU and memory usage, task execution duration, and input data volumes. Once these data have been processed, they are deleted.

Additionally, the traditional checkpoint coordinator in Flink is replaced with our coordinator possessing multi-feature awareness. This new checkpoint coordinator dynamically adjusts checkpoint intervals by receiving predictions of fault rates from the fault-aware module, various resource utilization information monitored by the monitoring module, and the number of slow tasks determined by the slow task detector. This improves the system reliability, availability, and efficiency, enabling automatic fault tolerance in the event of node failures or anomalies, and supporting continuous execution and high-quality output of streaming processing tasks.

The Actor communication system is a lightweight message-passing framework integrated within Ca-Stream. Each job manager and task manager includes an actor module that enables asynchronous, nonblocking communication through message exchanges. This architecture facilitates parallel decision-making and efficient coordination among components, such as triggering backups, initiating recovery, and updating runtime policies. Crucially, it avoids tight coupling and synchronization overhead, allowing Ca-Stream to scale efficiently and remain responsive to dynamic workload variations and fault-handling demands. The system architecture of Ca-Stream is depicted in Figure 5.

We also dynamically adjust the priorities of these three types of checkpoint mechanisms based on the real-time system status and predicted fault rates. When the resource utilization is excessively high, we elevate the priority of the resource-aware checkpoint mechanism to ensure system processing speed and task completion progress. In the event of an increased fault rate, we raise the priority of the fault-aware checkpoint mechanism to ensure system fault tolerance. If the number of slow tasks is excessively high, we increase the priority of the vertex feature-aware checkpoint mechanism to expedite slow task processing.

Additionally, we employ a joint decision-making method that comprehensively considers the outputs of fault-aware, resource-aware, and vertex feature-aware modules. For example, when the resource-aware module detects high resource utilization, the system triggers corresponding fault-tolerance strategies while balancing whether to increase the density of fault-aware checkpoints to address potential failures.

In the following sections, we elaborate on the details and algorithm implementations of these three mechanisms.

4.2 | Fault-Aware Checkpoint Mechanism

In a big data streaming computation cluster with fault-aware checkpoints, a QoS-aware checkpoint coordinator is used to initiate checkpoints based on the predicted fault rates. In typical big data streaming environments, the checkpoint coordinator periodically triggers checkpoints without considering the potential distribution of faults. However, the QoS-aware checkpoint coordinator initiates checkpoints based on fault awareness. If the fault rate is low, it dynamically increases the checkpoint interval, while it decreases the interval if the fault rate is high. This QoS-aware checkpoint coordinator receives input about the predicted fault rate from the QoS-aware agent and sends a customized schedule of checkpoint initiation times to the scheduler.

Considering that most faults occur before the average time between faults, the QoS-aware checkpoint coordinator dynamically adjusts the checkpoint intervals in regions with different fault densities of the application. The QoS-aware checkpoint coordinator initiates the application's execution with a fixed checkpoint interval. Subsequently, it optimistically assumes that no faults will occur in the near future. Therefore, it continuously increases the checkpoint interval based on the predicted fault rate (fr). This leads to monotonically increasing checkpoint intervals, such as $ci_1 < ci_2 \ldots < ci_n$. The QoS-aware checkpoint coordinator gradually increases checkpoint intervals until a failure occurs. If a failure occurs, it decreases the checkpoint interval to reduce the fault recovery time. Research on actual fault data and fault predictions indicates that shortly after a previous failure, the probability of subsequent failures is very high. Therefore, to mitigate potential faults, the QoS-aware checkpoint coordinator does not increase the checkpoint interval after a failure but decreases it.

The value of predicted fault rate fr plays a crucial role in determining changes in checkpoint intervals and, ultimately, the checkpoint frequency. The fr value ranges from 0 to 1, where 0 represents no fault, and 1 represents fault. When $fr = 1$, the behavior of fault-aware checkpoints is similar to the fixed checkpoint interval model because of the high fault rate, and any increase in checkpoint intervals may potentially increase fault recovery time. In this case, it is advisable to gradually decrease the checkpoint interval. On the other hand, when $fr = 0$, it represents the lowest fault rate and demonstrates the potential to increase checkpoint intervals to improve the efficiency of the application.

In situations where the fault probability is relatively low, increasing the checkpoint interval can increase utilization and reduce checkpoint costs. However, the scenario where $fr = 0$ is likely to double the increase in checkpoint intervals, which can easily lead to higher fault recovery overhead. Therefore, we impose a limit on the value of fr, with a minimum fault rate ($fr_{\min}$) of 0.25, to limit the exponential increase in the checkpoint interval. The failure-aware checkpoint agent allows a minimum value of fr of 0.25, and all values below this threshold are set to 0.25.

ALGORITHM 1 | Fault-aware Checkpoint Algorithm.

Input: Predicted Fault Rate (fr), Initialized checkpoint Interval (ci_0), Minimum fault rate ($fr_{\min}$);
Output: Failure aware checkpoints;
1: **if** $fr < fr_{\min}$ **then**
2: Set $fr = fr_{\min}$
3: **end if**
4: **while** Application not finished **do**
5: Use periodic checkpoint interval ci_0;
6: **if** Not fail **then**
7: **Non-uniform-Interval()**{ // *increase checkpoint interval until fault occurs*
8: Calculate $\Delta i_n = ci_{n-1} * (1 - fr)$;
9: Next checkpoint interval $ci_n = ci_{n-1} + \Delta i_n$;
10: Triggering checkpoint time $t_n = ci_n + ci_{n-1}$;}
11: **end if**
12: **if** failure occur **then**
13: Restart execution from last checkpoint;
14: **Non-uniform-Interval()**{ // *decrease checkpoint interval until minimum*
15: Calculate $\Delta i_n = ci_{n-1} * fr$;
16: Next checkpoint interval $ci_n = ci_{n-1} - \Delta i_n$;
17: Triggering checkpoint time $t_n = ci_n + ci_{n-1}$;}
18: until $ci_n = ci_{min}$;
19: **end if**
20: **end while**

Algorithm 1 describes the detailed procedure for the QoS-aware checkpoint coordinator to adjust the checkpoint interval based on predicted fault rates. Let t_0 represent the periodic checkpoint time. This algorithm initially runs the application with a fixed checkpoint interval (ci_0) and initiates regular checkpoints at time t_0 (Line 5). It then optimistically assumes that no faults will occur in the near future. Therefore, it begins to increase the checkpoint interval to trigger checkpoints (Line 7). It increases (ci_0) using Δi_1, where $\Delta i_1 = (ci_0)$ * (1-fr) (Line 8), and continues to increase each subsequent checkpoint interval (ci_n) at this rate. In this iterative process, the checkpoint intervals gradually increase, and checkpoints are initiated accordingly. Once the algorithm calculates the size of the checkpoint interval, it gradually increases the checkpoint initiation times as the intervals grow. The initiation time for the nth checkpoint is given as $t_n = ci_n + ci_{n-1}$ (Line 10). The algorithm dynamically increases the checkpoint intervals using this approach and eventually decreases the frequency of checkpoint initiation. The checkpoint intervals continue to increase until a failure occurs.

In the event of a failure, the application resumes from the last persisted checkpoint (Line 13) and decreases (ci_m) using Δi_m, where $\Delta i_m = ci_m * fr$ (Line 15). To tolerate potential faults, it maintains this rate to further reduce the checkpoint interval. Similarly, it iteratively decreases the checkpoint intervals and initiates checkpoints as the intervals decrease. Using this method, it dynamically decreases the checkpoint intervals and reduces the fault recovery time. The checkpoint intervals gradually decrease until they reach the minimum checkpoint interval.

4.3 | Resource-Aware Checkpoint Mechanism

The resource-aware checkpoint primarily focuses on sensing resource utilization for fault tolerance, including CPU usage and memory usage. Let's first consider the impact of CPU resources on checkpoint intervals. The resource-aware module is used to continuously improve the distributed streaming processing job based on user-defined QoS constraints. A threshold $Cconst$ ($0 < Cconst \leq 1$) is set for the maximum proportion of CPU time used, which defines an upper limit on the CPU time proportion used by each machine. Violating this agreement triggers a system reconfiguration for new checkpoint intervals [45]. Care must be taken when selecting new checkpoint interval values because reducing the checkpoint interval will decrease fault recovery time but increase checkpoint overhead and vice versa. If the CPU usage time proportion on a node exceeds the $Cconst$ threshold, the checkpoint interval is increased to reduce the checkpoint operation's impact on the CPU, allowing all CPU usage for task execution to make tasks execute more quickly. Given a CPU utilization rate U_{cpu}, if the CPU usage rate exceeds the $Cconst$ threshold, the next checkpoint interval can be calculated by Equation (12),

$$ci_n = ci_{n-1} * \frac{U_{\text{cpu}}}{Cconst} \tag{12}$$

U_{cpu} is equal to the ratio of CPU time used for normal logical processing in a task to the total CPU time during its execution.

$$U_{\text{cpu}} = \frac{N_{\text{cpu}}}{T_{\text{cpu}}} \tag{13}$$

The maximum threshold for memory usage, $Mconst$ ($0 < Mconst \leq 1$), is set to determine the impact of memory on checkpoint intervals. If the memory usage exceeds the defined threshold $Mconst$, the checkpoint interval will be increased to reduce the memory usage during checkpoint operations. This ensures that all available memory is used for executing tasks, allowing the tasks to run more efficiently and complete faster. Memory utilization rate, U_{mem}, can be defined as the ratio of the memory usage for normal logical processing within a task to the total available memory during the execution process.

$$U_{\text{mem}} = \frac{N_{\text{mem}}}{T_{\text{mem}}} \tag{14}$$

If the memory utilization rate exceeds the threshold $Mconst$, the next checkpoint interval can be calculated by Equation (15),

$$ci_n = ci_{n-1} * \frac{U_{\text{mem}}}{Mconst} \tag{15}$$

Algorithm 2 provides a detailed description of resource-aware checkpoint mechanism.

This Algorithm 2 starts by running the application with a fixed checkpoint interval (ci_0) and initiates regular checkpoints at time t_0 (line 4). When the monitoring module detects that the CPU utilization exceeds the threshold $Cconst$ or the memory utilization exceeds $Mconst$, it begins to initiate checkpoints at irregular intervals (line 6). It calculates the new checkpoint interval using either Equation (12) or Equation (15) (line 7). In this way, it iteratively increases the checkpoint interval and starts checkpoints as the interval increases. Once the size of the checkpoint interval is determined, the algorithm continues to increase the checkpoint start times with the increased interval. The time to start the n^th checkpoint is given by $t_n = ci_n + ci_{n-1}$ (line 8). Through this method, it dynamically increases the checkpoint interval while also reducing the checkpoint initiation frequency. The checkpoint interval gradually increases until both resource utilization rates fall below their respective thresholds.

Once both resource utilization rates fall below their thresholds, the current value of the checkpoint interval (ci_n) is reset to the fixed checkpoint interval (ci_0) (line 11).

ALGORITHM 2 | Resource-aware Checkpoint Algorithm.

Input: Periodic checkpoint interval ci_0, CPU occupancy rate U_{cpu}, Memory occupancy rate U_{mem}, CPU occupancy threshold $Cconst$, Memory occupancy threshold $Mconst$;
Output: QoS-aware checkpoints;
1: Calculate the CPU occupancy rate U_{cpu} by Equation (13);
2: Calculate the Memory occupancy rate U_{mem} by Equation (14);
3: **while** Application not finished **do**
4: Use Periodic checkpoint interval ci_0;
5: **if** $U_{\text{cpu}} > Cconst$ or $U_{\text{mem}} > Mconst$ **then**
6: **Non-uniform-Interval()**{ // *If U_{cpu} or U_{mem} exceeds threshold, increase checkpoint interval by corresponding equation*
7: Calculating next checkpoint interval by Equation (12) or Equation (15);
8: Triggering checkpoint time $t_n = ci_n + ci_{n-1}$;
9: **end if**
10: **if** $U_{\text{cpu}} \leq Cconst$ and $U_{\text{mem}} \leq Mconst$ **then** // *Once both resource utilization rates fall below their thresholds, the ci_n is reset to the fixed ci_0*
11: Recover checkpoint interval to ci_0;
12: **end if**
13: **end while**

4.4 | Vertex Feature-Aware Checkpoint Mechanism

In production environments, it is often challenging to avoid having hot machines where workloads are concentrated, leading to issues like intense flushing and mixed deployment clusters. Such machines may experience high workloads and busy input/output operations, which can result in significantly slower data processing tasks, ultimately jeopardizing job completion times.

The vertex feature awareness mechanism introduced in this section aims to identify abnormal machine nodes that contribute to slow job execution due to various issues such as hardware problems, occasional IO congestion, high CPU loads, etc. These issues can cause tasks running on these nodes to perform much slower than tasks on other nodes, consequently increasing the overall job execution time.

To implement this mechanism, Flink uses a slow task detector to identify slow tasks. The machine nodes where slow tasks are detected are considered abnormal machine nodes and are added to the machine node blacklist. The scheduler then creates new task instances for these slow nodes and deploys them on non-blacklisted machine nodes. However, these operations are typically designed for batch processing jobs and need to be extended to accommodate stream processing jobs.

We extend the interface of the slow task detector to detect the execution time and data transfer volume of stream processing tasks. Tasks with longer execution times and lower data consumption are identified as slow tasks. The objective is to reduce the time spent on checkpointing for slow tasks, thereby reducing checkpoint overhead and, more importantly, lowering resource consumption and task execution times. Throughout the execution process, the resource-aware module continuously observes the CPU and memory utilization in the environment.

Currently, slow tasks are detected using a runtime-based slow task detector, which periodically collects statistics on all completed tasks. It's important to note that this approach combines runtime with the actual input data volume. If there's data skew, tasks with significantly different data volumes but similar computing capabilities won't be detected as slow tasks, thus avoiding unnecessary resource wastage caused by false detection.

Algorithm 3 determines slow tasks based on task execution time and the amount of data transmitted by upstream tasks, counting the number of slow tasks. Only nodes where tasks have both a long execution time and a small data transfer volume are considered slow tasks.

ALGORITHM 3 | Slow task Election Algorithm.

Input: Total number of tasks m, Task List $taskList$, Historical task execution time for previous batches $execution[task\_i][batch\_pre]$, duration of task execution $duration[task\_i]$, task transmission of node $[task\_i]$, Total number of slow tasks N_{slow}

Output: number of slow tasks

1: **for** $task\_i = 1 \rightarrow m$ **do**
2: $duration[task\_i]$ = computeTaskExecutionDuration→ $(execution[task\_i][])$; // *identification of tasks with long execution time*
3: **end for**
4: $tasksList1$ = sortTasks$(m, duration[])$;
5: **for** $task\_i = 1 \rightarrow m$ **do**
6: $taskTrans[task\_i]$ = computeTaskTransmission $(execution[task\_i][])$; // *identification of tasks with small data transfer volumes*
7: **end for**
8: $tasksList2$ = sortTasks$(m, taskTrans[])$;
9: slowTasksList = selectSlowTasks$(s, taskList1, tasksList2)$; // *tasks are listed in tasksList1 and taskList2 are considered to be slow tasks*
10: N_{slow} = Count$(slowTasksList)$;
11: **return** N_{slow}

Flink version 1.16 introduced a Predictive Execution mechanism consisting of three key components. The first one is the Slow Task Detector. It periodically performs checks, taking into account the data processed by tasks and their runtime, to assess whether a task qualifies as a slow one. When it identifies a task as slow, it notifies the scheduler. The second component is the scheduler. Upon receiving a notification of a slow task, it informs the Blacklist Handler to mark the machine running the slow task. Additionally, as long as the number of instances running slow tasks doesn't exceed the user-configured limit, the scheduler creates and deploys new instances for them. When one instance completes its task, the scheduler terminates other instances running the same task. The third component is the Blacklist Handler, which Flink uses to blacklist machines. After a machine is blacklisted, no further tasks will be deployed on that machine. To support Predictive Execution, Flink version 1.16 introduced a soft-blacklist approach where tasks already running on a blacklisted machine can continue running without being canceled. It's important to note that these mechanisms are designed primarily for batch processing, not for stream processing. We focus on stream processing jobs in this study.

We build upon the Predictive Execution mechanism and introduce a Vertex Feature-Aware Checkpointing Strategy. It specifically detects tasks that are significantly slower than the rest of the job, as indicated by the Slow Task Detector. The strategy monitors the number of slow tasks in the environment, denoted as N_{slow}. If N_{slow} exceeds a predefined threshold, M, it increases the checkpointing interval, reducing the time spent by slow machines on checkpointing and allowing them to concentrate more on task execution. When the Slow Task Detector observes that N_{slow} has returned to normal levels, the checkpoint interval is reverted to its default value. Algorithm 4 provides a detailed description of the Vertex Feature-Aware Checkpoint algorithm.

This Algorithm 4 begins by running the application with a fixed checkpoint interval (ci_0) and initiates periodic checkpoints at time t_0 (line 2). When the Slow Task Coordinator detects that the number of slow tasks exceeds the threshold M, it starts initiating checkpoints at irregular intervals (line 4). It uses Δi_1 to increase

ALGORITHM 4 | Vertex Feature-aware Checkpoint Algorithm.

Input: Slowtask number N_{slow}, Initialized checkpoint interval ci_0

Output: Vertex feature-aware checkpoints

1: **while** Application not finished **do**
2: Use Initialized checkpoint interval ci_0
3: **if** $Nslow > M$ **then**
4: **Non-uniform-Interval()**{ // *increase the checkpoint interval until the Nslow* $\leq M$
5: Calculate $\Delta i_n = ci_{n-1}$*$[(Nslow - M)/M]$;
6: Next checkpoint interval $ci_n = ci_{n-1} + \Delta i_n$;
7: Triggering checkpoint time $t_n = ci_n + ci_{n-1}$;}
8: **end if**
9: **if** $Nslow \leq M$ **then**
10: Recover checkpoint interval to ci_0;
11: **end if**
12: **end while**

ci_0, where $\Delta i_1 = ci_{n-1} * [(Nslow - M)/M]$ (line 5), and continues to increase each checkpoint interval (ci_n) based on Δi_n. In this way, it iteratively increases the checkpoint interval while also reducing the frequency of checkpoint initiation. Once the size of the checkpoint interval is calculated, the algorithm continuously increases the checkpoint start time with the increasing checkpoint interval. The time to start the n^th checkpoint is $t_n = ci_n + ci_{n-1}$ (line 7). Through this method, it dynamically increases the checkpoint interval while reducing the frequency of checkpoint initiation. The checkpoint interval gradually increases until the number of slow tasks falls below the threshold. After the number of slow tasks drops below the threshold, the current value of the checkpoint interval (ci_n) is reset to the fixed checkpoint interval (ci_0) (line 10).

Algorithms 3 and 4 are sequential, with the former detecting slow tasks and counting their numbers, and the latter dynamically adjusting the checkpoint interval based on the number of slow tasks.

After introducing the QoS-aware checkpoint optimization strategy, the specific checkpoint cost is analyzed. The QoS-aware checkpoint coordinator significantly changes the checkpoint cost by continuously changing the checkpoint interval. This fluctuating checkpoint interval is vulnerable to high recovery overhead or high checkpoint costs because, compared to a fixed checkpoint interval during the rollback period, it either requires more rework or triggers more checkpoints. While the QoS-aware checkpoint coordinator considers fault rates, resource utilization, and number of slow tasks to change the checkpoint intervals, failures or resource shortages may occur during program execution, potentially affecting system performance. Anticipating failures or sensing resource scarcity incurs a cost for fault recovery.

To measure checkpoint overhead, two aspects are considered: the space used to store checkpoint data (S_{ck}) and the time taken to perform checkpoint operations (T_{ck}). These two parts are weighted differently to evaluate the metric F.

$$F = \alpha * S_{ck} + (1 - \alpha) * T_{ck} \tag{16}$$

In Equation (16), weight α can take values within the range (0, 1). When the value of the weight α is very close to 1, it indicates that the user places greater importance on space cost. Conversely, when α approaches 0, it implies that the user places greater importance on program runtime.

4.5 | System Implementation

In the process of implementing the checkpoint adaptive strategy for high availability in the Flink stream computing system, modifications are made to the `'startTriggeringCheckpoint'` method in the `'CheckpointCoordinator'` class of the `'flink-runtime'` project by extending the method to incorporate custom triggering logic. This method serves as the entry point for responding to checkpoint operations.

Before implementing the checkpoint adaptive strategy for high availability, it's essential to determine the characteristics of the stream compute nodes, such as task transfer data volume, task execution time, and resource utilization on nodes. In the extended `'startTriggeringCheckpoint'` method, a new `'calCheckpointScheduleTrigger'` method is added to calculate the execution parameters for checkpoint operations. This includes the following steps:

1. Using linear regression algorithms to predict fault rates, and designing dynamically adjustable checkpoint intervals based on the prediction results.
2. Calculating CPU utilization, memory utilization, task execution time, and task data transfer volume, and designing dynamically adjustable checkpoint intervals based on the calculated results.
3. Invoking the `'rescheduleTrigger'` method in the `'CheckpointCoordinator'` class to adjust the timing for the next checkpoint trigger based on the calculated checkpoint interval.

These optimization operations ensure the stability and reliability of the system, especially in the face of failure scenarios, where they can better handle and ensure data consistency and integrity. This will improve system performance and stability, providing better guarantees for the normal operation of applications.

5 | Performance Evaluation

This section begins by providing details about the hardware and software configurations of the experimental environment. Then, it describes the experimental setup. Finally, a comparative analysis is conducted between the proposed strategy and existing checkpoint mechanisms, focusing on checkpoint interval changes, checkpoint overhead, system recovery latency, task execution time, and resource utilization. This comparison aims to demonstrate the achievements of the proposed strategy in terms of resource efficiency and performance improvement in streaming computing systems.

5.1 | Experimental Environment and Parameter Setup

To validate the effectiveness of the checkpoint adaptive strategy with high availability, experiments are conducted using the Flink big data streaming computing system. HDFS is used as the external storage system for checkpoints, and a Zookeeper cluster provides coordination services. The cluster comprises 10 virtual machine nodes in a school data center, with 2 nodes dedicated to HDFS and Zookeeper, 1 node serving as the job manager, and the remaining 7 nodes as task manager nodes. Each task manager process has as many task slots as the number of CPU cores on the physical node. In the experiments, the monitoring environment is established using Prometheus, Pushgateway, and Node Exporter, where Prometheus is an open-source monitoring solution. Flink version 1.16, released on July 30, 2021, is used for the experiments. For detailed information about the hardware and software configurations, please refer to Tables 3 and 4.

To validate the effectiveness of the proposed strategy, experiments are conducted to compare the proposed checkpoint adaptive

TABLE 3 | Software configurations.

Software	Versions
OS	Ubuntu 20.04.1 64bit
Flink	Apache-Flink-1.16
JDK	Jdk1.8 64bit
Zookeeper	Zookeeper-3.4.14
Redis	Redis-7.0.9

TABLE 4 | Hardware configurations.

Hardware	Configuration
CPU	Intel core i7
Memory	32GB
Disks	40GB HDD
Network card	100Mbps

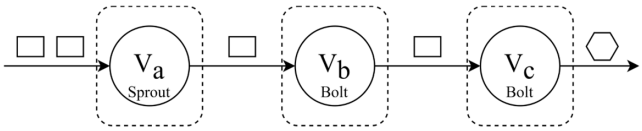

FIGURE 6 | WordCount topology.

strategy with high availability (Ca-Stream) against the current Flink periodic checkpointing mechanism (considered the most advanced ***periodic checkpoint mechanism (PCI) for article writing). The comparison covers five aspects: checkpoint interval changes, checkpoint overhead, system recovery latency, task execution time, and resource utilization.

Two test topologies, WordCount and SocketWindowWordCount in Figures 6 and 7, are used in the experiments. The WordCount topology represents a simple word counting application, and the SocketWindowWordCount topology is more complex application, designed for streaming window word counting from a web socket source.

To predict the fault rate in the system, failure data are obtained from the publicly available Failure Trace Archive database. The CPU usage threshold is set at 10% based on the runtime characteristics of tasks, and the memory usage threshold is set at 50%, also based on task runtime characteristics. This experimental setup allows for a comprehensive analysis and evaluation of the proposed Ca-Stream in comparison to the existing periodic checkpointing mechanism in Flink.

In the experiment, after initiating predictive execution and detecting slow tasks that trigger the predictive execution, Flink's user interface (UI) presents predictive execution instances on the job page under the subtask section of the node information. The Flink UI also displays the currently blacklisted task managers on the Task Managers & Overview page. The number of slow tasks can be determined by calculating the number of blacklisted task managers on the Flink page. In the experiment, the threshold for the number of slow tasks is set to 5, based on the runtime

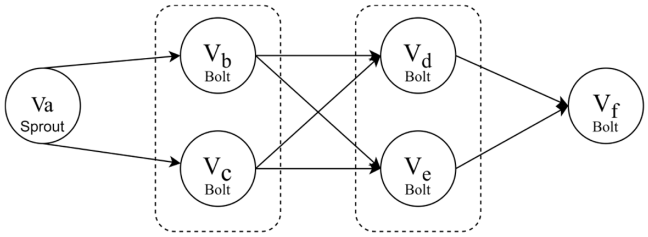

FIGURE 7 | SocketWindowWordCount topology.

characteristics of tasks. When the number of slow tasks exceeds this threshold, the checkpoint interval is increased; otherwise, it is reverted to the checkpoint interval corresponding to PCI.

When the number of slow tasks remains constant, the checkpoint interval is dynamically adjusted based on the fault rate and resource usage. Generally, if measures are not taken to reduce the number of slow tasks, such as migrating slow tasks to faster machines, the number of slow tasks remains constant. It has been verified through experimentation that as the complexity of the topology and the degree of operator parallelism increase, the number of slow tasks also increases. For instance, when using the WordCount topology with a parallelism of 1, the number of slow tasks is 1. When using the SocketWindowWordCount topology with a parallelism of 1, the number of slow tasks increases to 3. When using the SocketWindowWordCount topology with a parallelism of 2, the number of slow tasks increases to 10.

The proposed strategy can adaptively change the checkpoint intervals in the task execution process, possessing real-time improvement capabilities. However, the periodic checkpoint method can only support fixed interval checkpoints. This section compares the checkpoint interval changes between these two methods. It is worth noting that Flink does not start checkpoints by default, which is an important configuration aspect. The comparison of checkpoint interval changes aims to assess the adaptive capabilities of the proposed strategy compared to the periodic checkpoint method in terms of fault tolerance, resource utilization, and the number of slow tasks. It is important to note that the periodic checkpoint method in the experiment has a fixed checkpoint interval of 5 s, as introduced in the previous section.

Regarding the adaptive change in checkpoint intervals, each time a checkpoint is triggered, the current time is recorded, and then the time of the last checkpoint trigger is manually subtracted to obtain the current checkpoint interval. Figure 8 displays the variation in checkpoint intervals under both approaches. The proposed strategy can adaptively change checkpoint intervals based on the linear regression algorithm's predictions of fault rates, monitored resource utilization, and detected slow task count changes.

When the fault rate is relatively low, the proposed strategy increases the checkpoint interval; conversely, it decreases the interval. When resource utilization exceeds the threshold defined in the strategy, it increases the checkpoint interval. when resource utilization falls below the threshold, it reverts to the fixed 5-second checkpoint interval used in periodic checkpoints. When the number of slow tasks exceeds a certain threshold, it increases

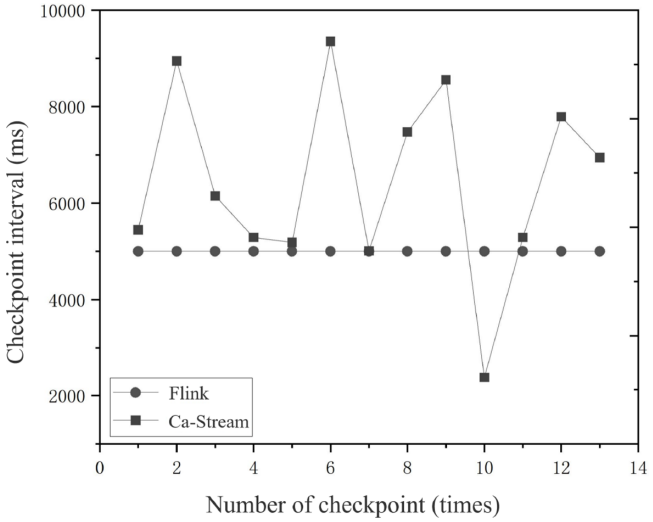

FIGURE 8 | Checkpoint interval comparison.

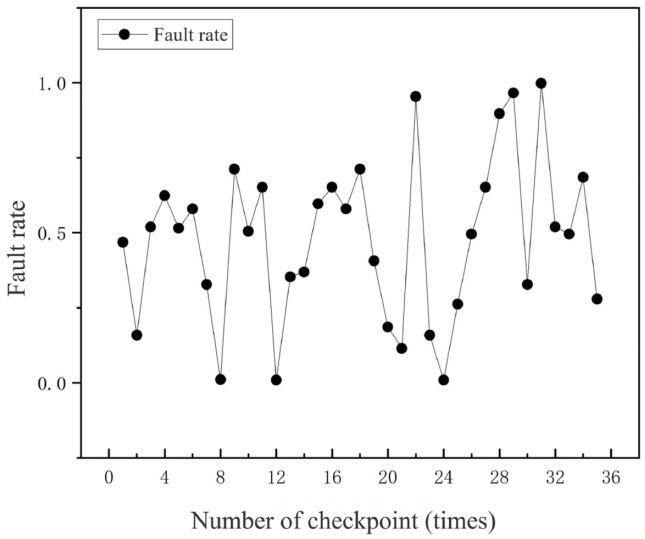

FIGURE 9 | Variations in fault rates.

the checkpoint interval; however, when the number of slow tasks falls below the threshold, it reverts to the fixed 5-second checkpoint interval used in periodic checkpoints.

From Figure 8, it is evident that the checkpoint intervals set by Ca-Stream are mostly greater than those of Flink, which significantly reduces checkpoint overhead. During the $10th$ checkpoint operation, the checkpoint interval of Ca-Stream becomes smaller than that of Flink. This is because the predicted fault rate for the 10_{th} checkpoint operation is relatively high, resulting in a drastic reduction in Ca-Stream's checkpoint interval.

In the experiment, the actual failure data obtained from the Failure Trace Archive database is used as a training dataset to predict the fault rate. The results of predicting the fault rate are illustrated in Figure 9. The x-axis represents the number of checkpoint operations, while the y-axis represents the predicted fault probability values. This figure depicts the predicted fault rates for the initial 36 checkpoint operations. From the graph, it can be observed that the fault rate is highly unstable and fluctuating continuously.

Figure 10 presents a comparison of the experimental results between Ca-Stream and Flink as the number of checkpoint operations changes. In alignment with the predicted fault rates during the program execution, the variation in fault rates has a noticeable impact on checkpoint intervals. The proposed Ca-Stream adjusts checkpoint intervals based on the predicted fault rate—increasing the interval when the fault rate is low, and decreasing it when the fault rate is high. This graph also shows that checkpoint intervals increase as the fault rate decreases, and decrease as the fault rate increases. However, periodic checkpoints of Flink cannot adjust checkpoint intervals based on fault rates and maintain a fixed value. Therefore, the proposed strategy is more flexible.

We compare checkpoint overhead from two aspects: the space used to store checkpoint data and the time spent on checkpoint execution. Figure 11 illustrates the time required to save checkpoint data for both strategies over 24 checkpoint operations. As shown in Figure 11, the time to save checkpoint data for both

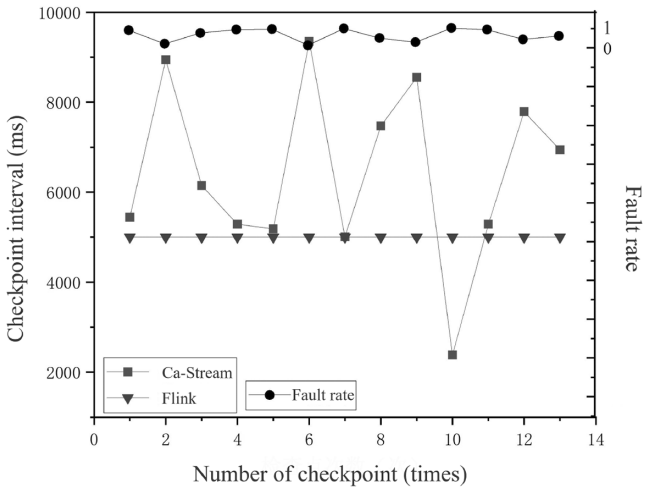

FIGURE 10 | Checkpoint intervals vary with fault rates.

strategies is dynamically changing. In the majority of cases, the time to save checkpoint data for the proposed checkpoint adaptive strategy is less than that for the PCM. After calculations, it is determined that the proposed checkpoint adaptive strategy reduces the time to save checkpoint data by an average of approximately 13.32%, with the best performance achieving a reduction of over 38%.

Checkpoint data size is a parameter that can be used to demonstrate the effectiveness of fault tolerance strategies. During checkpoint initiation, checkpoint data is saved multiple times in a reliable storage area. Storing checkpoint data in reliable storage requires space, and it should be retrievable from the storage in case of failures for recovery. If a significant amount of checkpoint data needs to be stored, the storage capacity should be sufficiently large. Therefore, comparing checkpoint data size is crucial for evaluating fault tolerance strategies. The comparison graph of checkpoint data size is shown in Figure 12. From Figure 12, it is evident that the checkpoint data size for both strategies is dynamically changing. In each instance, the checkpoint data size for the

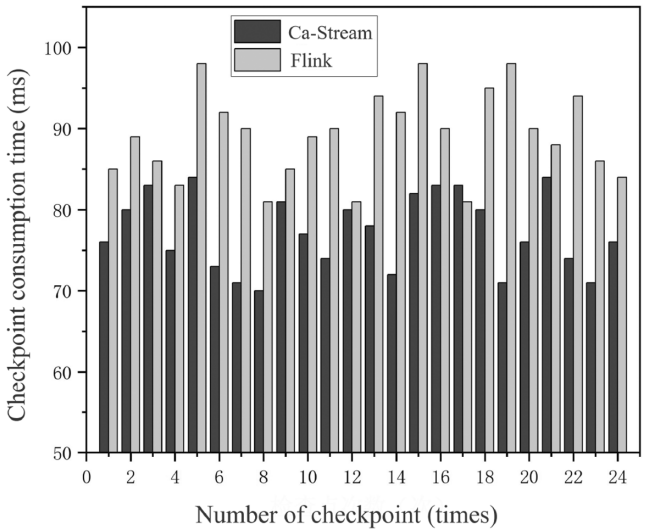

FIGURE 11 | Checkpoint consumption time.

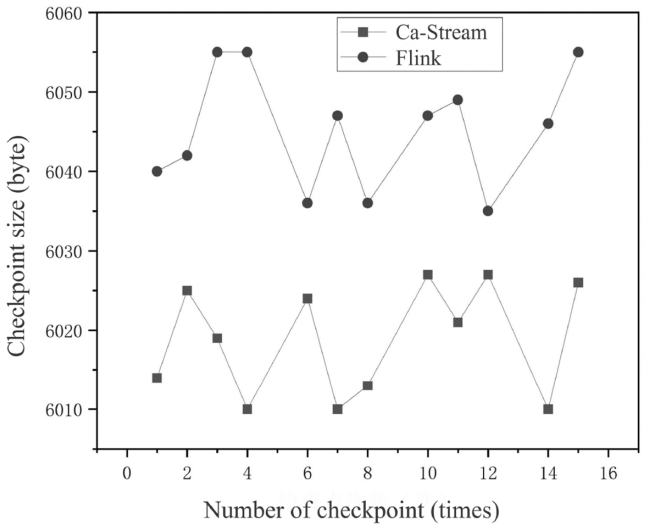

FIGURE 12 | Checkpoint data size.

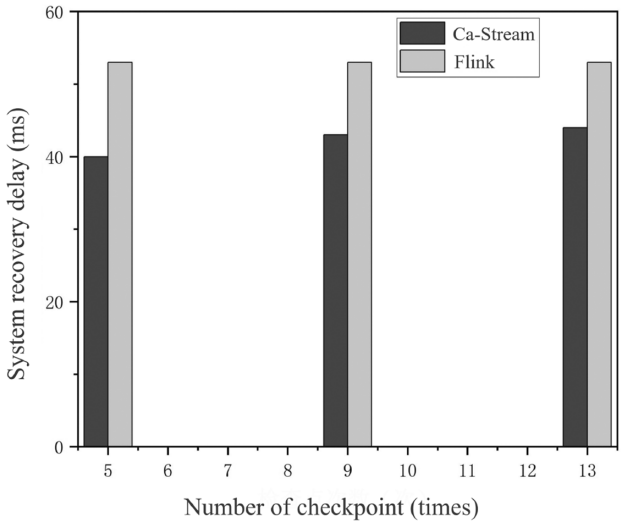

FIGURE 13 | System recovery delay.

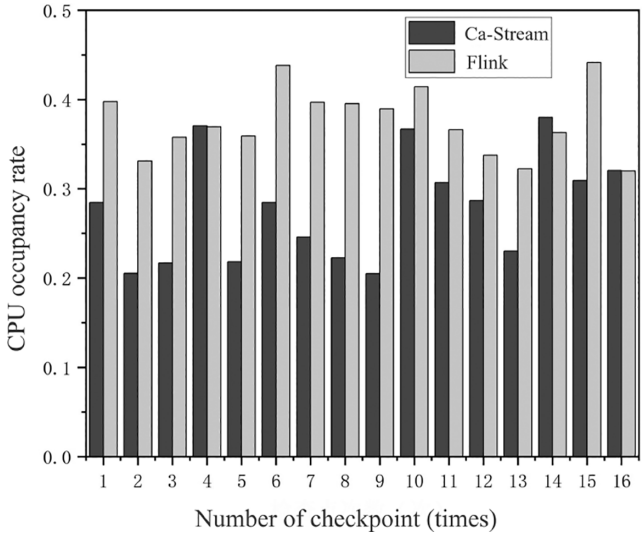

FIGURE 14 | CPU occupancy rate.

proposed Ca-Stream is smaller than that for Flink's corresponding checkpoint data size.

5.2 | System Recovery Latency

Figure 13 provides a statistical summary of system recovery latency corresponding to the two strategies during 13 checkpoint operations. From the graph, it is evident that the system undergoes 3 instances of failure recovery. Figure 13 illustrates that, with an increasing number of checkpoint operations, the system's recovery latency gradually increases for the proposed Ca-Stream. In contrast, the recovery latency for the PCM in Flink remains constant, as the checkpoint interval is fixed. Furthermore, the system's recovery latency corresponding to the Ca-Stream is consistently lower than that of the PCM in Flink. On average, the proposed Ca-Stream reduces system recovery latency by approximately 20.13%, with a maximum reduction of about 33% observed under optimal conditions.

5.3 | Resource Occupancy Rate

As we all know, CPU and running memory are essential resources for computation. The proposed strategy, after optimization, demonstrates the ability to utilize fewer resources. Let's compare Ca-Stream with the current PCM in terms of CPU utilization and memory consumption during normal task execution logic processing.

Figure 14 displays the CPU utilization for both strategies during 16 checkpoint operations. From Figure 14, it's evident that CPU utilization for both strategies is dynamically changing. Moreover, in the majority of checkpoint operations, CPU utilization corresponding to Ca-Stream is lower than that of the PCM.

Through calculations, it can be determined that the proposed Ca-Stream reduces CPU utilization by approximately 25.75%, compared to the CPU utilization in Flink's periodic checkpoint

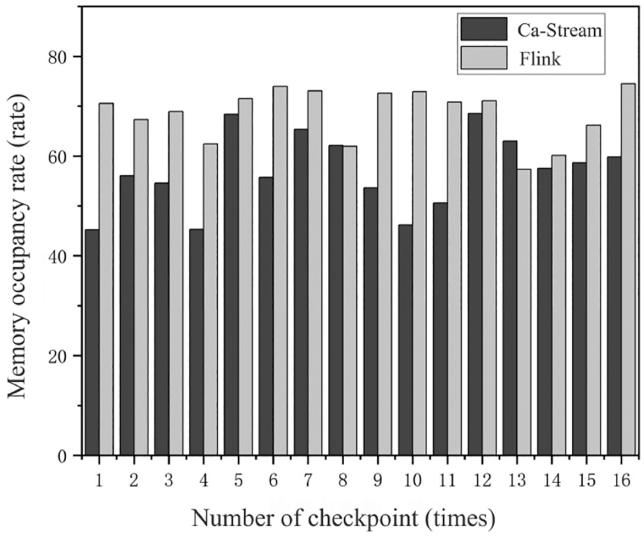

FIGURE 15 | Memory occupancy rate.

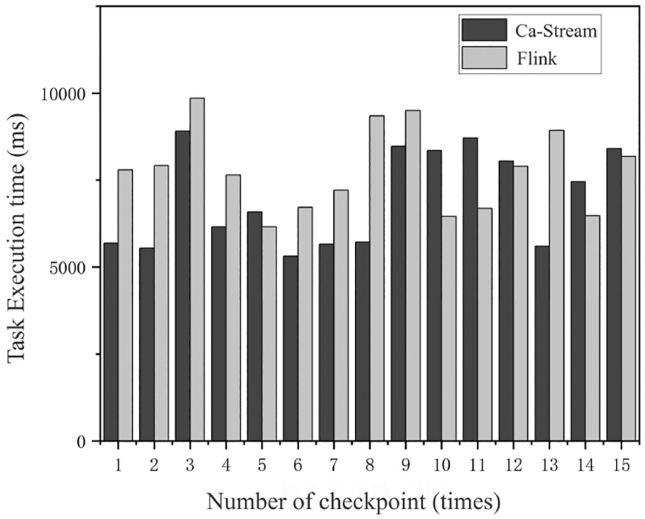

FIGURE 16 | Task execution time.

mechanism, with a reduction of over 47% observed under optimal conditions.

Figure 15 illustrates that memory utilization for both strategies is dynamically changing. The figure shows the monitored memory utilization during 16 checkpoint operations. It's evident from Figure 15 that, in the majority of checkpoint operations, the memory utilization corresponding to the proposed Ca-Stream is lower than that of the periodic checkpoint mechanism. Both strategies exhibit dynamic changes in memory utilization.

Through calculations, it can be determined that the proposed Ca-Stream reduces memory utilization by approximately 16.83% compared to the memory utilization in Flink's periodic checkpoint mechanism, with a reduction of approximately 37% observed under optimal conditions.

5.4 | Task Execution Time

Due to the increase in checkpoint intervals when there are more slow tasks, as described in this article, less time and resources are spent on checkpoint operations. This allows more time and resources to be dedicated to data computing, consequently reducing task execution time. Figure 16 depicts the dynamic changes in task execution time for both strategies as the number of checkpoints increases. This figure presents the task execution time for 15 checkpoint operations.

From Figure 16, it's evident that the task execution time corresponding to the proposed Ca-Stream is generally lower than that of Flink's periodic checkpoint mechanism. Through calculations, it can be determined that the proposed multifeature-aware fault-tolerant strategy, Ca-Stream, reduces the average task execution time by approximately 11.66% compared to the average task execution time in Flink's periodic checkpoint mechanism, with a reduction of over 39% observed under optimal conditions.

6 | Conclusion and Future Work

This article initially employs a linear regression algorithm to predict the fault rate in a stream computing system based on data from publicly accessible databases. Subsequently, dynamic adjustments are made to the initial checkpoint intervals. Then, by considering the characteristics of perception nodes, specifically referring to the execution duration of tasks on nodes, task data transmission volume, and resource utilization, the initial checkpoint intervals are adaptively adjusted. Through experiments involving factors such as fault rates, task CPU utilization, memory utilization, and the number of slow tasks in a distributed cluster, comparisons are made between the targeted checkpoint mechanism and the proposed fault-aware checkpoint strategy on the Flink system.

Ca-Stream exhibits a notable reduction in checkpoint data storing time by approximately 17.8% compared to the traditional synchronous checkpointing approach. The average task execution time is reduced by 11.66%, indicating less interference from fault-tolerance operations during standard processing. Furthermore, Ca-Stream reduces CPU utilization by 47% and memory usage by 37% during fault recovery scenarios under optimal conditions. The system recovery delay is also shortened by 33%, attributed to Ca-Stream's lightweight checkpointing and rapid restoration mechanism. These improvements collectively confirm the effectiveness of our strategy in enhancing both runtime performance and resource utilization. The experimental results indicate that the proposed strategy outperforms the current checkpoint mechanism, improving the checkpoint data storing time and task execution time. Additionally, it demonstrates marked decreases in CPU utilization, memory utilization, and system recovery delay.

The future improvement on this study will be carried out mainly from three aspects. First, perceive more factors and comprehensively evaluate the importance of nodes. Second, focus on integrating slow tasks and scheduling strategies. Finally, use reinforcement learning methods to further optimize the system.

Acknowledgments

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

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

1. H. Nasiri, S. Nasehi, and M. Goudarzi, "A Survey of Distributed Stream Processing Systems for Smart City Data Analytics," in *Proceedings of the International Conference on Smart Cities and Internet of Things* (ACM, 2018), 1–7.

2. H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of Distributed Stream Processing Frameworks for IoT Applications in Smart Cities," *Journal of Big Data* 6 (2019): 1–24.

3. D. G. G. R. G. García, "A Comparison on Scalability for Batch Big Data Processing on Apache Spark and Apache Flink," *Big Data Analytics* 2 (2017): 1–11.

4. P. Liu, H. Xu, D. Da Silva, Q. Wang, S. T. Ahmed, and L. Hu, "Fp4s: Fragment-Based Parallel State Recovery for Stateful Stream Applications," in *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE, 2020), 1102–1111.

5. T. Distler, "Byzantine Fault-Tolerant State-Machine Replication From a Systems Perspective," *ACM Computing Surveys* 24, no. 1 (2021): 1–37.

6. S. M. A. Akber, H. Chen, and H. Jin, "FATM: A Failure-Aware Adaptive Fault Tolerance Model for Distributed Stream Processing Systems," *Concurrency and Computation: Practice and Experience* 33, no. 10 (2021): e6167.

7. S. Jayasekara, S. Karunasekera, and A. Harwood, "Optimizing Checkpoint-Based Fault-Tolerance in Distributed Stream Processing Systems: Theory to Practice," *Software: Practice and Experience* 52, no. 1 (2022): 296–315.

8. B. Ghit and D. Epema, "Better Safe Than Sorry: Grappling With Failures of In-Memory Data Analytics Frameworks," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (ACM, 2017), 105–116.

9. B. Quinto and B. Quinto, "Introduction to Spark and Spark MLlib," in *Next-Generation Machine Learning With Spark: Covers XGBoost, LightGBM, Spark NLP, Distributed Deep Learning With Keras, and More* (Springer, 2020), 29–96.

10. S. Tang, B. He, C. Yu, Y. Li, and K. Li, "A Survey on Spark Ecosystem: Big Data Processing Infrastructure, Machine Learning, and Applications," *IEEE Transactions on Knowledge and Data Engineering* 34, no. 1 (2022): 71–91.

11. A. Téllez-Velázquez and R. Cruz-Barbosa, "A Spark Image Processing Toolkit," *Concurrency and Computation: Practice and Experience* 31, no. 17 (2019): e5283.

12. X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya, "E-Storm: Replication-Based State Management in Distributed Stream Processing Systems," in *Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, vol. 2017 (IEEE, 2017), 571–580.

13. To QC, J. Soto, and V. Markl, "A Survey of State Management in Big Data Processing Systems," *VLDB Journal* 27, no. 6 (2018): 847–872.

14. E. Fernandes, A. C. Salgado, and J. Bernardino, *Big Data Streaming Platforms to Support Real-Time Analytics* (ICSOFT, 2020), 426–433.

15. R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar, "KAFKA: The Modern Platform for Data Management and Analysis in Big Data Domain," in *Proceedings of the 2017 2nd International Conference on Telecommunication and Networks (TEL-NET)* (IEEE, 2017), 1–5.

16. T. Aung, H. Y. Min, and A. H. Maw, "Coordinate Checkpoint Mechanism on Real-Time Messaging System in Kafka Pipeline Architecture," in *Proceedings of the 2019 International Conference on Advanced Information Technologies (ICAIT)* (IEEE, 2019), 37–42.

17. A. Katsifodimos and S. Schelter, "Apache Flink: Stream Analytics at Scale," in *Proceedings of the 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)* (IEEE, 2016), 193.

18. C. Y. Lin, L. C. Wang, and S. P. Chang, "Incremental Checkpointing for Fault-Tolerant Stream Processing Systems: A Data Structure Approach," *IEEE Transactions on Emerging Topics in Computing* 10, no. 1 (2020): 124–136.

19. J. Fang, P. Chao, R. Zhang, and X. Zhou, "Integrating Workload Balancing and Fault Tolerance in Distributed Stream Processing System," *World Wide Web* 22 (2019): 2471–2496.

20. N. Hagshenas, M. Mojarad, and H. Arfaeinia, "A Fuzzy Approach to Fault Tolerant in Cloud Using the Checkpoint Migration Technique," *International Journal of Intelligent Systems & Applications* 14, no. 3 (2022): 18–26.

21. S. M. Attallah, M. B. Fayek, S. M. Nassar, and E. E. Hemayed, "Proactive Load Balancing Fault Tolerance Algorithm in Cloud Computing," *Concurrency and Computation: Practice and Experience* 33, no. 10 (2021): e6172.

22. D. S. Kumar and M. A. Rahman, "Simplified HDFS Architecture With Blockchain Distribution of Metadata," *International Journal of Applied Engineering Research* 12, no. 21 (2017): 11374–11382.

23. K. Gao, T. Nojima, H. Yu, and Y. R. Yang, "Trident: Toward Distributed Reactive SDN Programming With Consistent Updates," *IEEE Journal on Selected Areas in Communications* 38, no. 7 (2020): 1322–1334.

24. D. Sun, S. Gao, X. Liu, F. Li, X. Zheng, and R. Buyya, "State and Runtime-Aware Scheduling in Elastic Stream Computing Systems," *Future Generation Computer Systems* 97 (2019): 194–209.

25. X. Wang, C. Zhang, J. Fang, R. Zhang, W. Qian, and A. Zhou, "A Comprehensive Study on Fault Tolerance in Stream Processing Systems," *Frontiers of Computer Science* 16 (2022): 1–18.

26. Y. Fang, Q. Chen, and N. Xiong, "A Multi-Factor Monitoring Fault Tolerance Model Based on a GPU Cluster for Big Data Processing," *Information Sciences* 496 (2019): 300–316.

27. A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault Tolerance in Iterative-Convergent Machine Learning," *Proceedings of the 36th International Conference on Machine Learning* 97 (2019): 5220–5230.

28. M. Amoon, N. El-Bahnasawy, S. Sadi, and M. Wagdi, "On the Design of Reactive Approach With Flexible Checkpoint Interval to Tolerate Faults in Cloud Computing Systems," *Journal of Ambient Intelligence and Humanized Computing* 10 (2019): 4567–4577.

29. J. Ren, K. Wu, and D. Li, "Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)* (IEEE, 2020), 237–247.

30. J. Villamayor, D. Rexachs, and E. Luque, "A Fault Tolerance Manager With Distributed Coordinated Checkpoints for Automatic Recovery," in *Proceedings of the 2017 International Conference on High Performance Computing & Simulation (HPCS)* (IEEE, 2017), 452–459.

31. K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint Restart Support for Heterogeneous HPC Applications," in *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, vol. 2020 (IEEE/ACM, 2020), 242–251.

32. J. Zhao, H. Liu, S. Zhang, et al., "Fast Parallel Recovery for Transactional Stream Processing on Multicores," in *Proceedings of the IEEE 40th International Conference on Data Engineering (ICDE)* (IEEE, 2024), 1478–1491.

33. J. John, I. D. N. Araya, and M. Gerndt, "iCheck: Leveraging RDMA and Malleability for Application-Level Checkpointing in HPC Systems," in *Proceedings of the 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)* (IEEE, 2023), 467–474.

34. A. M. Kermarrec and C. Morin, "HA-PSLS: A Highly Available Parallel Single-Level Store System," *Concurrency and Computation: Practice and Experience* 15, no. 10 (2003): 911–937.

35. M. J. Gossman, B. Nicolae, and J. C. Calhoun, "Scalable I/O Aggregation for Asynchronous Multi-Level Checkpointing," *Future Generation Computer Systems* 160 (2024): 420–432.

36. A. Frank, M. Baumgartner, R. Salkhordeh, and A. Brinkmann, "Improving Checkpointing Intervals by Considering Individual Job Failure Probabilities," in *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE, 2021), 299–309.

37. S. U. Mushtaq, S. Sheikh, and S. M. Idrees, "Enhanced Priority Based Task Scheduling With Integrated Fault Tolerance in Distributed Systems," *International Journal of Cognitive Computing in Engineering* 6 (2025): 152–169.

38. M. K. Geldenhuys, L. Thamsen, and O. Kao, "Chiron: Optimizing Fault Tolerance in Qos-Aware Distributed Stream Processing Jobs," in *Proceedings of the 2020 IEEE International Conference on Big Data (Big Data)* (IEEE, 2020), 434–440.

39. Y. Zhuang, X. Wei, H. Li, Y. Wang, and X. He, "An Optimal Checkpointing Model With Online OCI Adjustment for Stream Processing Applications," in *Proceedings of the 2018 27th International Conference on Computer Communication and Networks (ICCCN)* (IEEE, 2018), 1–9.

40. S. Marzouk and M. Jmaiel, "A Survey on Software Checkpointing and Mobility Techniques in Distributed Systems," *Concurrency and Computation: Practice and Experience* 23, no. 11 (2011): 1196–1212.

41. M. K. Geldenhuys, D. Scheinert, O. Kao, and L. Thamsen, "Phoebe: QoS-Aware Distributed Stream Processing Through Anticipating Dynamic Workloads," in *Proceedings of the 2022 IEEE International Conference on Web Services (ICWS)* (IEEE, 2022), 198–207.

42. T. Gupta, S. Krishnan, R. Kumar, et al., "Just-In-Time Checkpointing: Low Cost Error Recovery From Deep Learning Training Failures," in *Proceedings of the 19th European Conference on Computer Systems* (ACM, 2024), 1110–1125.

43. Z. Zhang, T. Liu, Y. Shu, S. Chen, and X. Liu, "Dynamic Adaptive Checkpoint Mechanism for Streaming Applications Based on Reinforcement Learning," in *Proceedings of the 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)* (IEEE, 2023), 538–545.

44. P. Sigdel, X. Yuan, and N. F. Tzeng, "Realizing Best Checkpointing Control in Computing Systems," *IEEE Transactions on Parallel and Distributed Systems* 32, no. 2 (2021): 315–329.

45. M. K. Geldenhuys, B. J. Pfister, D. Scheinert, L. Thamsen, and O. Kao, "Khaos: Dynamically Optimizing Checkpointing for Dependable Distributed Stream Processing," *Annals of Computer Science and Information Systems* 30 (2022): 553–561.

46. S. Jayasekara, A. Harwood, and S. Karunasekera, "A Utilization Model for Optimization of Checkpoint Intervals in Distributed Stream Processing Systems," *Future Generation Computer Systems* 110 (2020): 68–79.

47. A. Benoit, L. Perotin, Y. Robert, and F. Vivien, "Checkpointing Strategies for a Fixed-Length Execution," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE, 2024), 508–518.

48. H. Li, J. Wu, Z. Jiang, X. Li, and X. Wei, "Minimum Backups for Stream Processing With Recovery Latency Guarantees," *IEEE Transactions on Reliability* 66, no. 3 (2017): 783–794.

49. X. Wei, Y. Zhuang, H. Li, and Z. Liu, "Reliable Stream Data Processing for Elastic Distributed Stream Processing Systems," *Cluster Computing* 23 (2020): 555–574.