Fast Recovery of Correlated Failures in Distributed Stream Processing Engines

Su, Li; Zhou, Yongluan

Published in:

Publication date:
2021

Citation for published version (APA):
Fast Recovery of Correlated Failures in Distributed Stream Processing Engines

Li Su
lisu.sl@alibaba-inc.com
Alibaba Group
China

Yongluan Zhou
zhou@di.ku.dk
University of Copenhagen
Denmark

ABSTRACT
In a large-scale cluster, correlated failures usually involve a number of nodes failing simultaneously. Although correlated failures occur infrequently, they have significant impact on systems’ availability, especially for streaming applications that require real-time analysis, as repairing the failed nodes or acquiring additional ones would take a significant amount of time. Most state-of-the-art distributed stream processing systems (DSPSs) focus on recovering individual failures and do not consider the optimization for recovering correlated failure. In this work, we propose an incremental and query-centric recovery paradigm where the recovery of failed operator partitions would be carefully scheduled based on the current availability of resources, such that the outputs of queries can be recovered as early as possible. By analyzing the existing recovery techniques, we identify the challenges and propose a fault-tolerance framework that can support incremental recovery with minimum overhead during the system’s normal execution. We also formulate the new problem of recovery scheduling under correlated failures and design algorithms to optimize the recovery latency with a performance guarantee. A comprehensive set of experiments are conducted to study the validity of our proposal.

CCS CONCEPTS
• Information systems → Stream management.

KEYWORDS
stream processing, fault tolerance, correlated failure

ACM Reference Format:

1 INTRODUCTION
Distributed Stream Processing Systems (DSPSs), such as Flink [8], Samza [30], Apache Storm [37] and Spark Streaming [45], can be deployed on large computing clusters to process continuous queries over high-velocity data streams. Continuous queries usually run for a very long interval and hence would unavoidably experience various failures, especially when the system is deployed on large-scale clusters [34]. The causes of correlated failures include failures of shared hardware (such as network devices and power facilities), and more importantly, software errors. It was reported that failures caused by software errors exhibit strong temporal correlations and tend to propagate across networks [22, 28, 33, 36, 42]. Previous studies [12, 14, 16, 28, 33, 42, 44] show that, correlated failures, where a number of nodes fail simultaneously or within a short interval, have significant impact on systems’ availability. Google [12] reported that, by using a 120s sliding window to detect correlated failures, around 37% and 8% of individual node failures are part of correlated failures involving at least 2 nodes and 10 nodes, respectively. [42] analyzes the failure traces of 19 different distributed systems with dozens to thousands of nodes, and shows that more than 50% of the downtime of these systems are caused by peak failure periods, which are defined as periods with high hourly failure rate. Facebook [39] has to mitigate 100+ datacenter outages over the last four years, approximately one outage every two weeks.

Tolerating large-scale failures has been implemented as an important feature of messaging systems like Kafka [7], which can actively replicate messages in standby clusters across multiple data centers. Key-value stores such as Cassandra [24] and Megastore [4] can also replicate data across multiple clusters to achieve high availability in case of correlated failures. These technologies can be used to replicate input data across multiple data centers to support fault tolerance for DSPSs upon correlated failures. However, recovering failed queries also requires restoring computation states and coordinating data replays with accuracy guarantee. Correlated failures pose new challenges to fault tolerance in DSPSs, as they exhibit characteristics that are very different from independent failures.

Firstly, correlated failures usually incur unavailability of a large number of nodes. Existing DSPSs, such as Flink, Storm and Spark Streaming, do not consider the capabilities of alive nodes during recovery. The alive nodes could be heavily overloaded to recover all the failed partitions in this case. Although one can scale out the processing cluster by adding new nodes [9] and then perform load balancing [11, 13, 32] after failure recovery is completed, the decoupling of failure recovery, scaling out and load balancing would incur unnecessary latency during the entire process. We envision that a resource-aware recovery mechanism will be an important optimization for recovering correlated failures in DSPSs.

Secondly, it is impractical to assume instant availability of resources for recovering correlated failures. It takes time to repair the failed nodes or to acquire new resources. Moreover, the recovered or
newly allocated nodes may not be available simultaneously, but most likely one by one with significant time gaps between them. Although deploying stream applications using containers helps in achieving fast resource allocation, with the considerations of security and flexible control over the computing environment, it is common practice that containers are deployed on a cluster of physical/virtual machines with limited resources. E.g., Amazon ECS container agents have to run on a cluster of EC2 instances [1]. Correlated failures of physical/virtual machines can result in a shortage of resources for new containers. We conducted experiments (Section 6.1) recording the launching time of in total 180 VMs on Amazon EC2, the results show that the time used for establishing a new node varies from 2 to 6 minutes, even on a cloud service with “virtually” unlimited resources. In this scenario, blocking the recovery until all necessary new nodes become available can result in slow recovery. A viable approach is to gradually recover a part of the failed partitions following the arriving pace of new resources. This approach requires careful scheduling over the recovery order of failed partitions to minimize the latency of resuming final outputs. On the other hand, maintaining a standby resource pool only for the recovery of correlated failures is not cost-effective.

Lastly, as reported in [42], node failures often occur one by one during a peak failure period, with an average interval between failures varying from sub-seconds to hundreds of seconds. Therefore, the system may detect multiple node failures at different time points. In this scenario, existing DSPSs such as Storm and Flink would start multiple recovery processes, each of which rollbacks the state of the entire topology and replays the source input data. However, repeating state rollback and source replay from the same processing progress upon every detected failure is quite expensive, as the prior recovery efforts could be wasted.

In this work, we strive for developing fault-tolerance techniques for correlated failures that bring little performance overhead when the system is at a normal state, and at the same time are compatible with the existing mechanisms of handling independent failures. We first revisit the failure recovery techniques in existing DSPSs to identify the challenges of recovering correlated failures. Based on the analysis, we propose a fault-tolerance mechanism that incurs as little overhead as the previous approaches during a normal processing period. Furthermore, to minimize the recovery latency, we propose an incremental and query-centric recovery paradigm, where the recoveries of failed operator partitions are scheduled to incrementally resume the query outputs as early as possible following the arriving pace of resources. This new paradigm would provide not only a shorter recovery latency and earlier recovery of individual queries, but also a more responsive user interface with a smoother transition from a failed state to a fully recovered one. The contributions of this work are summarized as follows:

- We identify the insufficiencies of fault-tolerance mechanisms in existing DSPSs when recovering correlated failures.
- We propose a new incremental recovery framework that adopts a resource-aware query-centric paradigm to schedule the recovery of operator partitions. Techniques including adaptive buffering and order-preserving processing are proposed to support efficient incremental recovery and guarantee exactly-once processing.
- We formulate the problem of optimizing incremental recovery plan and show that it is NP-Hard. We design a dynamic programming algorithm that can produce the optimal plan with an exponential complexity, and a practical approximate algorithm with polynomial complexity with a performance guarantee.
- We implement our prototype system as a fault-tolerance module in a distributed, low-latency stream processing system built on top of Apache Storm. We conduct experiments with real datasets on Amazon EC2 to study the validity of our proposal.

2 PRELIMINARIES

2.1 System Models

As in mainstream DSPSs, we model a data tuple as a \{key, value\} pair, where the default format of the key is string and the value is a blob that is opaque to the system.

2.1.1 Topology. A topology consists of multiple operators, each containing a user-defined function and subscribing to the output streams of other operators. By denoting operators as vertex and stream subscriptions between operators as directed edges, a topology can be abstracted as a directed acyclic graph (DAG). Figure 1 (left) presents an example topology of a community-based smart city application, which conducts real-time traffic monitoring by aggregating user-reported speed and incident information. Within this topology, \(S_1\) and \(S_2\) are streams of user-reported location and incident events, respectively. \(O_3\) calculates the average vehicle speed on each road segment, and \(O_4\) counts the user-reported incidents on each road segment. The output streams of \(O_3\) and \(O_4\) are joint in \(O_5\) to monitor the average vehicle speed on the road segments where incident happened. As the outputs of \(O_3\), \(O_4\) and \(O_5\) are all useful to users, each of them is designated as an output operator of a logical query. Outputs generated by output operators can be consumed externally, e.g., persisted in a database for further analysis and visualization or used as the input by other applications.

2.1.2 Parallelization & Query Partition. An operator can be parallelized into multiple operator partitions with identical computation logic defined by the user-defined function of the operator. Each input stream of an operator is split into a set of key groupings based on their keys. A union of the same key grouping from each of the input streams of an operator form the complete input of an operator partition. For simplicity, operator partition is also referred to as partition in the rest of this paper. Figure 1 (right) presents a parallelized execution plan of the example topology. For example, the input streams of \(O_3\) are partitioned into two groups based on the road segment IDs, one for each partition of \(O_3\). Within a physical node, partitions from the same operator are executed by one physical task. A task can host multiple partitions from the same operator, and maintain the computation states, input and output buffers for different partitions separately from each other. A physical node can accommodate tasks from multiple operators.
For an output operator, each partition generates a subset of its outputs. Partitions of output operators are referred to as output partitions. We observed that, upon failures, to recover the outputs of an output partition, one has to recover all the failed partitions that are located in the upstream of this output partition. For instance, to recover the output of \( p_1 \), any failed partition in the partition set \( \{ p_{11}, p_{31}, p_{32}, p_{21}, p_{41} \} \) must be recovered.

**Definition 1. Query Partition:** The query partition \( Q_i \) of an output partition \( p_i \) is a set of partitions that satisfies the following: (1) \( p_i \in Q_i \); (2) if \( p_k \in Q_i \), then all the upstream neighboring partitions of \( p_k \) also belong to \( Q_i \).

Query partitions are also referred to as queries hereafter for brevity. There are in total 8 queries in Figure 1, one for each output partition. For example, \( Q_{01} \) consists of \( p_{11}, p_{31}, p_{32}, p_{21}, p_{41} \) and \( p_{31} \). A partition can be shared by multiple queries.

In some applications, outputs of different queries could have different degrees of importance and it could make sense to prioritize the recovery of queries of more importance. For example, in Figure 1, suppose the outputs of \( O_1 \) and \( O_2 \) are partitioned based on their spatial locations, and hence they deliver traffic information of the same areas to the same downstream partition. From the perspective of traffic management and navigation, traffic information of congested roads is more critical than that of the other areas. Upon failure, it is beneficial to prioritize recovering the queries that monitor those critical areas.

Previous studies show that node failures are often correlated, where multiple nodes in the system fail almost simultaneously or within a very short time interval \([12, 14, 16, 28, 33, 42, 44]\). We assume that the crashes of partitions can be detected in time via heartbeat monitoring. In addition, we have no assumption on the spacial and temporal distributions of correlated failures in the system.

### 2.2 Challenges in Fault-Tolerance Design

Data buffering and checkpointing are the key techniques to achieve fault tolerance in DSPSs \([8, 30, 37]\). In the rest of this section, we discuss the common data buffering and checkpointing techniques, and the challenges of extending them to the recovery of correlated failure. In addition, we identify the new problem of recovery scheduling for correlated failures.

#### 2.2.1 Data Buffering

Source buffering and upstream backup are two typical data buffering techniques in DSPSs. With source buffering, the system buffers the input data that have not been completely processed at the sources. This approach is simple to implement and incurs low overhead, therefore it is widely adopted in most existing DSPSs, including Storm \([37]\), Flink \([8]\) and Samza \([30]\). However, with source buffering, the buffered source input would be replayed whenever we recover a newly-detected failed task during a correlated failure. This means that the recovery progress would be blocked until the recovery of the last task is started.

Upstream backup \([9]\) allows each task to buffer its output whose effects on the downstream neighbors have not been included in a checkpoint. With upstream backup, whenever a failed task is restarted, its progress can be resumed by replaying tuples buffered in its upstream tasks. Recovering a failed task would not influence the progress of the normal/recovered tasks, which makes upstream backup more suitable for recovering correlated failures. On the other hand, the overheads of adopting upstream backup include the memory consumption of buffered outputs and the costs of buffer management, e.g., appending new output and trimming the expired.

In this work, we propose adaptive buffering, a new buffering technique that only performs source buffering during normal execution. When a correlated failure is detected, upstream backup is activated to support incremental recovery. The switching between buffering approaches is light-weight and only occurs when a burst of failures is detected. The details of adaptive buffering are presented in Section 4.2.

#### 2.2.2 Checkpointing

To resume the states of the failed tasks, checkpoints of task states should be periodically generated and persisted. Storm \([37]\) checkpoints task states periodically without any synchronization, thus the processing progress of different tasks is inconsistent. Therefore, in Storm, the progress of a restarted task could fall behind that of its downstream neighbors, and thus duplicate elimination is necessary to achieve exactly-once processing.

Other systems, such as Flink \([8]\), attempt to generate consistent checkpoints among the distributed tasks to achieve exactly-once processing. The checkpoints are consistent if they are generated after the tasks have completely processed an identical sequence of input tuples. We can also refer to the set of consistent checkpoints of the topology as a global checkpoint. Upon a node failure, the query can be rollbacked using the global checkpoint, which makes duplicate elimination unnecessary as all stateful tasks have an identical progress.

However, as new task failures could be detected while the recovery of a correlated failure is ongoing, the progress of newly recovered tasks may still fall behind its downstream, which makes duplicate elimination inevitable even with consistent checkpoints. A straightforward solution to avoid duplicate elimination is that, whenever a new task failure is detected, rolling back the states of the topology with the latest global checkpoint and then conducting source replay. However, with this approach, the computation performed between the two replays is wasted. To solve this problem, we propose order-preserving processing to support efficient duplicate elimination. The basic idea of order-preserving processing is to enforce an identical output order for a partition across multiple replays, such that other partitions consuming its outputs can perform duplicate elimination according to the sequence numbers of their inputs. Similar to adaptive buffering, order-preserving processing is only activated during recovery and thus incurs no overhead on normal execution. The details of order-preserving processing are presented in Section 4.3.
2.2.3 Recovery Scheduling. Most existing DSPSs [8, 30, 37, 43] focus on independent failures, they do not optimize the recovery latency of correlated failures. Upon a correlated failure, if the recovery resources arrive incrementally, an intuitive approach is to schedule the failed partitions for recovery individually in a topological order, which is referred to as the operator-centric paradigm. However, this paradigm fails to minimize the latency of resuming the producing of query outputs. Figure 2 presents a simple but illustrative example of this paradigm, where upper stream partitions are recovered before the downstream ones. In this example, we assume the newly deployed recovery nodes arrive sequentially. As one can see, no sink operator partition is recovered until node $N_d$ arrives.

To speed up the recovery of query outputs, we propose a query-centric paradigm where the recovery of failed partitions would be carefully scheduled to incrementally recover the outputs of queries as early as possible. More specifically, if a correlated failure occurs, we gradually increase the number of recovered queries following the arriving pace of new resources. As illustrated in Figure 2, if the query-centric paradigm is adopted, the failed queries are recovered much earlier than the operator-centric approach. In the rest of this section, we formally define the optimization problem of recovery scheduling in correlated failure.

2.3 Problem Definition

During an incremental recovery, if the available resources are not enough, only a subset of the failed partitions will be selected for recovery, which constitutes the first partial recovery plan. Whenever new resources are ready, another partial recovery plan consisting of another set of failed partitions will be scheduled.

While the metric of resource consumption is orthogonal to the algorithms proposed in this paper, we use CPU usage to represent the resource consumption of each partition in our implementation. The task that hosts partition $p_j$ is denoted by $t_j$. Note that $t_j$ may host multiple partitions from the same operator, i.e., the partitions in $t_j$ conduct identical computation defined by the operator. We periodically collect the CPU usage of task $t_j$. The CPU usage of $p_j$ is calculated as follows: (1) calculate the ratio between the number of tuples processed by $p_j$ and the numbers of tuples processed by all the partitions in $t_j$; (2) multiply this ratio with the CPU usage of $t_j$ and use the multiplication as the CPU usage of $p_j$. The CPU usage is denoted as a percentage between 0 and 100%. As our approach is agnostic to the resource metric, one can also use, for example, I/O usage as the measure of resource consumption.

A failed query is considered as recovered if and only if all of its failed partitions are recovered. Users can specify the priority (an integer) for each query partition in a topology. By default the query priority is set to 1. We illustrate how query priority is used when optimizing the recovery plan in Section 5. The formal definition of the problem of recovery scheduling is presented as follows:

**Definition 2. Recovery Scheduling**: Given a topology $T$ and the set of failed partitions $FP$, generate a recovery plan $RP$, $RP \subseteq FP$, under the constraint of the amount of available resources $R$, such that the sum of the priorities of the queries recovered by executing $RP$ is maximized.

The Recovery Scheduling problem is NP-hard. The details of our optimization algorithm are presented in Section 5.

3 SYSTEM OVERVIEW

We implemented our system on top of Apache Storm [37]. Operators are implemented as a layer between the basic Storm operators and the user-defined functions. As presented in Figure 3, we implement Elastic Scheduler and Control Bolt to coordinate incremental recovery. The elastic scheduler works by ensuring that each node in the cluster has exactly one processing task for every operator in the topology. Each processing task maintains the computation states, input and output buffers for multiple partitions independently. The design of isolating partition and task facilitates flexibly migrating partitions within the processing cluster. Every node in the processing cluster accommodates exactly one task for each operator in the topology. Therefore, a failed partition can be recovered on any node within the processing cluster. If new nodes are attached to the cluster, the scheduler starts one task for all the operators on this new node such that any partition processed in the failed nodes can be restarted on the new node.

The control bolt is a system-level operator automatically generated and appended to the user-submitted topology. The statistics manager in the control bolt is responsible for collecting workload statistics for all the partitions and nodes. A timer thread running in the statistics manager periodically sends the requests for workload statistics to the processing tasks, which will then calculate its own CPU usage and the CPU usages of its partitions during the last statistics period, and then send it to the statistics manager. Resource consumptions of partitions are stored in ZooKeeper in case that the control bolt is failed. The fault tolerance coordinator, which is also referred to as coordinator in this paper, detects task failure by checking the heartbeats of workers in the ZooKeeper. Upon a task failure is detected, the coordinator uses the optimization algorithm presented in Section 5 to schedule failure recovery according to the availability of resources. The control bolt is stateless, if failed, it will be restarted by Nimbus and the interrupted recovery scheduling will be resumed.

Tuples are categorized into two types: data tuple and control tuple. All the input tuples and the tuples generated by executing the user-specified functions in operators are data tuple. Data tuples are those that are executed or created by the user-specified functions. The punctuations triggering checkpointing are also recognized as data tuples. Control tuples are used to manipulate the execution of the topology, including the requests for workload statistics, buffer trimming, etc. Within each processing task, two separate input queues are maintained for data tuples and control tuples respectively. Tuples in both queues are processed in the FIFO order. The processing of control tuples is always prioritized in a task whenever its queue of control tuples is not empty.
4 FAULT TOLERANCE

In this section, we first present the details of our checkpointing and buffering techniques. Then we introduce the mechanism of order-preserving processing and explain how the recovery is scheduled. As introduced in Section 2.1, partition is the minimum unit for checkpointing and buffering.

4.1 Checkpointing

We use punctuations to trigger partition checkpointing in an asynchronous approach. Punctuations are generated periodically by the source operators and inserted into the data streams. Each punctuation has a unique sequence number (e.g., $P_k$). Suppose a partition $P_i$ has a number of input streams, $S_1, S_2, \ldots, S_n$. Once a punctuation $P_k$ from $S_j$ arrives at $P_i$, all the tuples $T_{pj}$ from $S_j$ after $P_k$ will be stored in the input buffer before the checkpoint for $P_k$ is generated in $P_i$. After $P_i$ receives a $P_k$ from each input stream, it generates a checkpoint containing the computation state of $P_i$. The fault tolerance coordinator will be acknowledged of the checkpoint for $P_i$. Once the coordinator receives the acknowledgment from each partition about their checkpoints for $P_k$, it is aware of a globally consistent checkpoint of the entire topology, denoted as $cp(P_k)$, is generated. The metadata of the successful consistent checkpoints are backed up in ZooKeeper for fault-tolerance consideration.

![Figure 4: An example of checkpoint punctuation. The partition is denoted as a circle which has two input streams and one output stream. Within each stream, rectangles with $P_i$ represents punctuation whose sequence number is $i$ and rectangles containing an integer denotes a data item. To differentiate from the data in $S_i$, data items in $S_j$ and their intermediate output are marked in gray.](image)

As we have discussed in Section 2.2.1, source buffering is unsuitable for recovering correlated failures, because one has to replay the buffered data from the sources for every newly detected node failure to guarantee exactly-once processing. In this case, only recovering a part of the failed partitions makes little sense, because the recovery of any failed partition would require to redo the whole recovery again. This means that the recovery progress is blocked until there are sufficient resources to recover all the failed partitions. Therefore, maintaining upstream buffers is necessary for incrementally recovering a correlated failure. To avoid the overhead of upstream backup during normal processing, we propose adaptive buffering, where only buffers at the sources are maintained during normal processing. Upon failures, all the partitions except the sinks start buffering their outputs to support incremental recovery.

4.2.1 Enabling Buffering. The control bolt broadcasts recovery messages to all the partitions to start recovery. On detecting a failure, the control bolt activates incremental recovery if at least $X$ node failures are detected since the last global checkpoint is generated. Otherwise it simply uses the blocking recovery method. While the value of $X$ should be determined by the failure statistics of the particular cluster, we use the value of 1 following the observation in [12]. If the control bolt detects a correlated failure, it sets the buffering flag in the recovery message as true. On receiving such a recovery message, $P_i$ turns on adaptive buffering, where all the outputs that $P_i$ produces after the state rollback should be buffered. In other words, by denoting the selected global checkpoint for state rollback as $cp(P_k)$, all the intermediate results generated after punctuation $P_k$ will be buffered. Within $P_i$, for each of its downstream neighboring partitions, denoted by $P_j$, a FIFO queue is maintained to store the outputs that $P_j$ should emit to $P_i$. The buffered data would be spilled to disk if the specified buffer space is full.

4.2.2 Disabling Buffering. Once all the failed partitions are recovered, it is not necessary to continue maintaining the output buffers and the previously buffered outputs should be cleared to release the memory space. The completion of a new global checkpoint indicates that the previous failure has been fully recovered. Therefore, once the control bolt detects that a global checkpoint is successfully generated, it broadcasts control messages to notify all the partitions, except the source ones, to set their buffering flag as false and empty all the output buffers.

![Figure 5: An example of adaptive buffering.](image)

As we have discussed in Section 2.2.1, source buffering is unsuitable for recovering correlated failures, because one has to replay the buffered data from the sources for every newly detected node failure to guarantee exactly-once processing. In this case, only recovering a part of the failed partitions makes little sense, because the recovery of any failed partition would require to redo the whole recovery again. This means that the recovery progress is blocked until there are sufficient resources to recover all the failed partitions. Therefore, maintaining upstream buffers is necessary for incrementally recovering a correlated failure. To avoid the overhead of upstream backup during normal processing, we propose adaptive buffering, where only buffers at the sources are maintained during normal processing. Upon failures, all the partitions except the sinks start buffering their outputs to support incremental recovery.

4.2.1 Enabling Buffering. The control bolt broadcasts recovery messages to all the partitions to start recovery. On detecting a failure, the control bolt activates incremental recovery if at least $X$ node failures are detected since the last global checkpoint is generated. Otherwise it simply uses the blocking recovery method. While the value of $X$ should be determined by the failure statistics of the particular cluster, we use the value of 1 following the observation in [12]. If the control bolt detects a correlated failure, it sets the buffering flag in the recovery message as true. On receiving such a recovery message, $P_i$ turns on adaptive buffering, where all the outputs that $P_i$ produces after the state rollback should be buffered. In other words, by denoting the selected global checkpoint for state rollback as $cp(P_k)$, all the intermediate results generated after punctuation $P_k$ will be buffered. Within $P_i$, for each of its downstream neighboring partitions, denoted by $P_j$, a FIFO queue is maintained to store the outputs that $P_j$ should emit to $P_i$. The buffered data would be spilled to disk if the specified buffer space is full.

4.2.2 Disabling Buffering. Once all the failed partitions are recovered, it is not necessary to continue maintaining the output buffers and the previously buffered outputs should be cleared to release the memory space. The completion of a new global checkpoint indicates that the previous failure has been fully recovered. Therefore, once the control bolt detects that a global checkpoint is successfully generated, it broadcasts control messages to notify all the partitions, except the source ones, to set their buffering flag as false and empty all the output buffers.
## 4.3 Order-Preserving Processing

In Section 2.2.2, we have discussed the necessity and challenges of duplicate elimination for the guarantee of exactly-once processing. During a correlated failure, as node failures may not occur simultaneously, it is possible that a partition (e.g., $p_i$) could be recovered for multiple times. This can result in that the downstream partitions of $p_i$ can be recovered from the source operators, because the effects on the partition states of processing these data have been persisted.

Lastly, it is important to note that output buffers constructed for failure recovery differ from those buffers used in data transferring between partitions. In our system implementation, the latter is immediately trimmed whenever the data are transferred to the downstream partitions.

### 4.3.1 Order-Preserving Processing Algorithm

The algorithm of order-preserving processing is described in Algorithm 1. A FIFO queue $IB[j][k]$ is used to store tuples in batch $B_j$ received from input stream $IS_k$ (lines 5–6). When $p_i$ receives the `Barrier` message from all its input streams, $p_i$ starts processing tuples within this batch in a predefined round-robin order across the input streams (lines 9–14). After $p_i$ has finished processing a batch, it broadcasts the `Barrier` message to all its downstream neighboring partitions.

With order-preserving processing, the output order of $p_i$ is guaranteed to be identical across multiple replays. Therefore, a partition can skip duplicate tuples by checking their local sequence numbers (lines 3–4). The correctness of order-preserving processing is based on the following two assumptions:

**Assumption 1.** If a partition $p_i$ processes the input tuples in an identical order, then it would generate and deliver the output tuples in an identical order.

**Assumption 2.** Messages sent from $p_i$ to $p_j$ are received at $p_j$ in the identical order as they are sent from $p_i$.

Most of the commonly used operators in streaming applications, such as deterministic filtering, aggregate and joins, satisfy Assumption 1. For non-deterministic operators like random filters with non-deterministic filtering predicates, duplicate elimination cannot be done by checking the sequence number of inputs, as the outputs of random filters could be different across different replays even if the inputs are identical. To enforce exactly-once processing for topologies with non-deterministic operators, whenever a new partition failure is detected, the topology state must be rolledback to the latest global consistent checkpoint to avert the appearance of duplicate intermediate outputs. In addition, assumption 2 is held in most existing DSPSs, such as [8, 27, 37], which use in-order message delivery mechanism.

A batch-based approach in stream processing may generally incur additional processing latency [10]. However, as we only use it during failure recovery, i.e., when the system is mainly processing the data that have already arrived and buffered in the system, it would only incur negligible additional latency.

### 4.4 Incremental Recovery

In this subsection we explain how the control bolt coordinates failure recovery. On detecting node failures, with the assumption that the processing cluster is not overloaded before failure, the control bolt deploys $X$ new nodes to scale out the processing cluster, where $X$ is the minimal number of failed nodes which can trigger incremental recovery. In the meantime, with the available resources in alive nodes, the control bolt generates the first recovery plan with the optimization algorithm presented in Section 5. Note that a recovery plan may only consist of a part of the failed partitions if the amount of available resources is insufficient for a complete recovery. Whenever the control bolt detects the arrival of new nodes, if there are still failed partitions waiting for recovery, it generates a new recovery plan and broadcasts it to all the processing tasks.

In the first recovery plan, a boolean flag of state rollback is set as true. This means that, on receiving the first recovery plan, processing tasks rollback the states of alive partitions with the latest global checkpoint $ck(P_k)$. $ck(P_k)$ is also used to restore the states of failed partitions. The boolean flag of state rollback is set as false in all the following recovery plans, which means that only the first recovery

---

### Algorithm 1: Order-Preserving Processing

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input: $p_i$: partition, $t$: input tuple; $InputList$: list of $p_i$’s input streams sorted by the IDs of the source partitions;</td>
</tr>
<tr>
<td>2</td>
<td>$BID$: the id of the current batch,</td>
</tr>
<tr>
<td>3</td>
<td>$i ← t.BatchId;$</td>
</tr>
<tr>
<td>4</td>
<td>$k ← t.LastSeqId;$</td>
</tr>
<tr>
<td>5</td>
<td>if $t.seq ≤ lastSeq[k]$ then</td>
</tr>
<tr>
<td>6</td>
<td>return; $lastSeq[k]$ stores the sequence number of the last processed tuple from input stream $IS_k$; $t$ is a duplicate tuple produced by the upstream partition. */</td>
</tr>
<tr>
<td>7</td>
<td>if $t.type = Data Tuple$ then</td>
</tr>
<tr>
<td>8</td>
<td>$IB[j][k].put(t);$</td>
</tr>
<tr>
<td>9</td>
<td>if $t.type = Barrier$ then</td>
</tr>
<tr>
<td>10</td>
<td>$BO[j] += 1;$</td>
</tr>
<tr>
<td>11</td>
<td>$lastSeq[k] = IB[j][k].size$;</td>
</tr>
<tr>
<td>12</td>
<td>foreach input stream $IS &lt; InputList$ do</td>
</tr>
<tr>
<td>13</td>
<td>while $IB[IS][k].hasNext()$ do</td>
</tr>
<tr>
<td>14</td>
<td>$t' ← IB[IS][k].pull();$</td>
</tr>
<tr>
<td>15</td>
<td>$p_i.process(t');$</td>
</tr>
<tr>
<td>16</td>
<td>$lastSeq[k] ← t'.seq;$</td>
</tr>
<tr>
<td>17</td>
<td>foreach output stream of $p_i$ do</td>
</tr>
<tr>
<td>18</td>
<td>send $Barrier$ message for batch $BID;$</td>
</tr>
</tbody>
</table>

---

The algorithm of order-preserving processing can be done asynchronously by the individual partitions and hence is very lightweight.

### 4.2.4 Buffer Trimming
The expired outputs should be trimmed from the buffers in the source operators to prevent them from growing indefinitely. As mentioned in Section 4.1, during normal processing, the control bolt generates that the checkpoints of all the partitions for a punctuation $P_k$ have been generated, inputs received before $P_k$ can be trimmed from the source operators, because the effects on the partition states of processing these data have been persisted.

Lastly, it is important to note that output buffers constructed for failure recovery differ from those buffers used in data transferring between partitions. In our system implementation, the latter is immediately trimmed whenever the data are transferred to the downstream partitions.

### 4.3 Order-Preserving Processing

In Section 2.2.2, we have discussed the necessity and challenges of duplicate elimination for the guarantee of exactly-once processing. During a correlated failure, as node failures may not occur simultaneously, it is possible that a partition (e.g., $p_j$) could be recovered for multiple times. This can result in that the downstream partitions of $p_j$ receive duplicate data. In addition, as $p_j$ may produce outputs in various orders across different runs of recovery, its downstream partitions cannot purely differentiate duplicates based on the arriving order of the inputs. In this section, we present our solution, referred to as order-preserving processing, to perform duplicate elimination.

Order-preserving processing is switched on or off simultaneously with adaptive buffering. With order-preserving processing switched on, the tuples emitted by the source operators are split into consecutive mini batches and each partition attaches monotonically increasing local sequence numbers to their output tuples. Between consecutive mini batches, each partition of source operators broadcasts a `Barrier` message to its downstream partitions.
plan incurs state rollback. Adaptive buffering and order-preserving processing are switched on in all the alive and restarted partitions after the recovery is started. The progress of failed partitions can be resumed by state restoration with \( ck(P_k) \) and processing the buffered tuples in their upstream neighboring partitions. Algorithm 2 presents the detailed steps about how a recovery plan is executed in processing task, which is self-explanatory.

During the recovery of correlated failures, if \( cp(P_k) \) is the global checkpoint used to perform state rollback, guaranteeing exactly-once processing requires all the source inputs with a sequence number larger than punctuation \( P_k \), being buffered in the source operators. This can be achieved by implementing source operators using a fault-tolerant queuing service such as Apache Kafka [21], or persisting the inputs locally.

In the case that incremental recovery is not triggered, e.g., when recovering an independent failure, adaptive buffering and order-preserving processing will not be activated to avoid unnecessary overheads. Note that, the benefits of incremental recovery also depends on the availability of standby resources. If there are sufficient resources at the beginning of an incremental recovery, all the failed partitions will be recovered in one plan, leaving no space for further plan optimization. In specific, incremental recovery is suitable for queries with multiple sink operators or whose sink operator is parallelized into multiple partitions. For such queries, incremental recovery helps in sequentially resuming the complete outputs of query partitions as soon as possible upon a correlated failure.

5 RECOVERY SCHEDULING

We first design a dynamic programming (DP) algorithm to generate the optimal recovery plan. The basic idea of the DP algorithm is to gradually increment the amount of available recovery resources and enumerate all the possible expansions of the previously generated recovery plans to find out the optimal plan. Details of DP is presented in Algorithm 3.

DP starts by calculating the recovery cost of each failed query (line 3 – 9). Within each iteration of the while loop, we increment the resource usage by 1 unit. We then check each candidate recovery plan \( CP_i \) within \( SC \) to find the query \( Q_k \) whose recovery cost \( C_{ik} \) is equal to the amount of available resources in \( CP_i \) in the current iteration. If we find such a query \( Q_k \), \( CP_i \) is expanded by including all the failed partitions of \( Q_k \). As the newly-recovered partitions could be shared by other failed queries, we update the local recovery costs of such queries in \( CP_i \). The while loop ends when the current resource usage reaches the resource constraint \( R \). Finally, the recovery plan that has the maximal sum of the priorities of the recovered queries in \( RP \) is the maximum among \( SC \).

Algorithm 2: ExecuteRecoveryPlan

<table>
<thead>
<tr>
<th>Input: RP, recovery plan; T, topology;</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreach Task ( t_j ) in topology ( T ) do</td>
</tr>
<tr>
<td>if ( RP \text{ globalRollBack} ) = true then</td>
</tr>
<tr>
<td>foreach failed partition ( p_j ) which should be recovered on ( t_j ) do</td>
</tr>
<tr>
<td>Install partition ( p_j ) on task ( t_j ) with ( RP ) checkpoint;</td>
</tr>
<tr>
<td>( p_j \text{ buffering} ) = true;</td>
</tr>
<tr>
<td>( p_j \text{ orderPreserve} ) = true;</td>
</tr>
<tr>
<td>foreach alive partition ( p_m ) on ( t_j ) do</td>
</tr>
<tr>
<td>clear the input and output buffer;</td>
</tr>
<tr>
<td>rollback the state of ( p_j ) with ( RP ) checkpoint;</td>
</tr>
<tr>
<td>( p_j \text{ buffering} ) = true;</td>
</tr>
<tr>
<td>( p_j \text{ orderPreserve} ) = true;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>foreach failed partition ( p_j ) which should be recovered on ( t_j ) do</td>
</tr>
<tr>
<td>install partition ( p_j ) on task ( t_j ) with ( RP ) checkpoint;</td>
</tr>
<tr>
<td>( p_j \text{ buffering} ) = true;</td>
</tr>
<tr>
<td>( p_j \text{ orderPreserve} ) = true;</td>
</tr>
<tr>
<td>foreach downstream neighboring partition ( p_k ) of ( p_j ) do</td>
</tr>
<tr>
<td>if partition ( p_k ) is scheduled for recovery in ( RP ) then</td>
</tr>
<tr>
<td>( Buffer_{ik} ) = output tuples buffered for ( p_k ) in partition ( p_j );</td>
</tr>
<tr>
<td>send ( Buffer_{ik} ) to the destination task of ( p_k );</td>
</tr>
</tbody>
</table>

Algorithm 3: Dynamic Programming: DP(R, FP)

<table>
<thead>
<tr>
<th>Input: ( R ) Amount of available resources; FP Set of failed partitions;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: ( RP ) Recovery plan;</td>
</tr>
<tr>
<td>foreach Query ( Q_j ) do</td>
</tr>
<tr>
<td>( QT_{ai} ) = ( \emptyset );</td>
</tr>
<tr>
<td>( C_{ai} ) = 0;</td>
</tr>
<tr>
<td>foreach Partition ( p_j ) ∈ FP do</td>
</tr>
<tr>
<td>if ( p_j ) belongs to ( Q_j ) then</td>
</tr>
<tr>
<td>( QT_{ai} ) = ( QT_{ai} \cap { p_j } );</td>
</tr>
<tr>
<td>( C_{ai} ) = ( C_{ai} + \text{Cost}(p_j) );</td>
</tr>
<tr>
<td>while usage ( &lt; R ) do</td>
</tr>
<tr>
<td>usage = usage + 1;</td>
</tr>
<tr>
<td>foreach candidate plan ( CP_i ) in ( SC ) do</td>
</tr>
<tr>
<td>( diff ) = usage - ( \text{Cost}(CP_i) );</td>
</tr>
<tr>
<td>( U_i ) = ( \max { C_{ij} \mid C_{ij} &gt; 0 \text{ &amp; } QT_j \notin CP_i } );</td>
</tr>
<tr>
<td>if ( diff \leq U_i ) then</td>
</tr>
<tr>
<td>foreach Query ( Q_k ) ∈ ( { Q_j \mid C_{ij} = diff } ) do</td>
</tr>
<tr>
<td>( n + \text{, } CP_i = CP_i \cup { QT_{ik} } );</td>
</tr>
<tr>
<td>( SC = SC \cup { CP_i } );</td>
</tr>
<tr>
<td>foreach Query ( Q_k ) do</td>
</tr>
<tr>
<td>update ( QT_{ai} ) and ( C_{ai} );</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>remove ( CP_i ) from ( SC );</td>
</tr>
<tr>
<td>return the recovery plan ( RP, RP \times SC ), where the sum of priorities of the recovered queries in ( RP ) is the maximum among ( SC ).</td>
</tr>
</tbody>
</table>

The time complexity of this algorithm is bounded by \( O(2^M) \), where \( M \) is the number of failed queries.
Algorithm 4: BestDensity($R$, $FP$)

Input: $R$: amount of available resources, $FP$: set of failed partitions;
Output: $RP$: recovery plan;
1. $n \leftarrow 0$; $CP_n \leftarrow \emptyset$; $SC \leftarrow \{CP_n\}$;
   $\triangleright CP_n$ is the initial empty candidate plan; $SC$ is the set of candidate plans;
2. foreach partition $p_i \in FP$ do
   3. $c_i \leftarrow$ the resource consumptions of $p_i$;
   4. $M \leftarrow 0$;
      $\triangleright M$ denotes the number of failed queries;
5. foreach query $Q_i$ do
   6. if $Q_i$ contains failed operator then
      7. $M + 1$ $\triangleright M$ is the resource consumption of recovering all the failed partitions in $Q_i$;
      8. $PD_i \leftarrow Profit_Density(Q_i, FP)$;
      9. find $Q_m$ such that $PD_m \geq \max \{PD_i\}$, where $C_i \leq R$;
      10. $CP_n \leftarrow CP_n \cup \{Q_m\}$; $n + 1$;
   11. for $i = 0$ to $M - 1$ do
      12. $Q_m$ $\triangleright$ $i$th failed query;
      13. for $j = i + 1$ to $M$ do
      14. $Q_n$ $\triangleright$ $j$th failed query;
      15. $\triangleright C_m,n$ $\triangleright$ the resource consumption of recovering $Q_m$ and $Q_n$ concurrently;
      16. if $C_m,n \leq R$ then
      17. $CP_n \leftarrow CP_n \cup \{Q_m, Q_n\}$;
      18. $SC \leftarrow SC \cup \{CP_n\}$;
      19. $n + 2$;
      20. enumerate all the combinations of $i$ failed queries where the sum of their recovery costs is smaller than $R$;
      21. foreach Failed query $Q_m \notin CP_n$ do
      22. $\triangleright CP_i$ $\triangleright$ $i$th plan in $SC$;
      23. found $\leftarrow$ false, maxDens $\leftarrow 0$, index $\leftarrow 0$;
      24. while Cost($CP_i$) $\leq R$ do
      25. foreach Failed query $Q_m \notin CP_i$ do
      26. if Cost($CP_i$) + $C_m \leq R$ & $PD_m \geq \max Dens$ then
      27. $\triangleright index \leftarrow m$, found $\leftarrow$ true, maxDens $\leftarrow PD_m$;
      28. $\triangleright CP_i$ and $PD_i$ denote the cost and profit density of $Q_m$ calculated under the condition that only the queries in $CP_i$ are recovered;
      29. if found then
      30. $CP_i$ $\leftarrow CP_i \cup \{Q_m\}$;
      31. maxDens $\leftarrow 0$;
      32. else
      33. break;
      34. return the recovery plan $RP$, $RP \in SC$, where the sum of priorities of the recovered queries in $RP$ is the maximum among $SC$.

In the above equation, $c_k$ is the resource consumption of $p_k$, and $f_k$ is the number of failed queries that share $p_k$.

For each initial candidate plan $CP_i$, we gradually extend it by adding the failed query that has the largest profit density and a recovery cost smaller than $R - \text{Cost}(CP_i)$. For different candidate plans, the recovery cost and profit density of a query could be different, because the sets of recovered partitions are different in different candidate plans.

BestDensity starts by calculating the recovery cost and profit density of each failed query (line 2 – 8). Next, in case that the amount of recovery resource is only enough for recovering one query, the failed query that has the largest profit density and a recovery cost smaller than $R$ is selected as the first candidate plan $CP_0$ (line 9 – 10). The next step is to extend the set of candidate plans by enumerating all the combinations of 2 failed queries and constructing a candidate plan for each of such combinations (line 11 – 19). Denoting the number of failed queries as $M$, the number of initial candidate plans is smaller than $\frac{M(M-1)}{2} + 1$. Next, each candidate plan in $SC$ is gradually extended by adding the failed query with the largest profit density under the resource constraint (line 21 – 31). In the end, we return the candidate plan with the highest sum of priorities of recovered queries among all the plans in $SC$. The time complexity of this algorithm is $O(M^3 \cdot \log M)$, where $M$ is the number of failed queries. The approximate ratio of Algorithm 4 is given in Theorem 5.1.

**Theorem 5.1.** Algorithm 4 achieves an approximation ratio of $(1 - e^{-2})$, where $d$ denotes the maximal number of queries by which a partition is shared.

Theorem 5.1 states that, by utilizing BestDensity to optimize the recovery plan, we have:

$$\sum_{Q_i \in RQ} prt_i \geq (1 - e^{-2}) \cdot \sum_{Q_i \in RQ} prt_i$$

The recovery plan generated by BestDensity is referred to as $RP$. The set of queries recovered by $RP$ is denoted as $RQ$. $RQ^*$ represents the set of queries recover by plan $RP^*$, which is the optimal plan that has the maximal sum of the priorities of recovered queries.

During the incremental recovery, an upper bound $(e.g., 80\%)$ is set as the maximal CPU usage of each node. After a recovery plan is generated, we always assign the partitions to be recovered on the node with the maximal amount of available resources. This simple but efficient assignment strategy works well as the workloads of operators are split into fine-grained partitions.

**6 EVALUATION**

In this section, we conduct experiments to evaluate incremental recovery by comparing it with existing systems and other variations. In the experiments of recovering correlated failure, we deploy clusters on Amazon EC2 with m3.large instances, the cluster size ranges from 5 to 18. When evaluating the effects of incremental recovery, it is mainly the topology structure that influences the processes of recovery. Benchmarks such as Linear Road [3] and NEXMark [29] mainly focus on the performance characteristics (e.g., throughput and latency) during normal execution. The structures of queries in the both benchmarks are simple (e.g., a small number of operators connected as a chain), and thus cannot well characterize the subtle effects of incremental recovery. We therefore compose two synthetic queries with rich sub-query structures (see Figure 7 and Figure 8) and use them in the experiments. A real dataset consisting of 369,382 tweets is used as the source, which are repeatedly emitted into the source operator to emulate a continuous input source. We use the following systems and variations in our experiments:

- **Storm.** We use Storm as a representative system adopting source replay techniques. We mainly examine how source replay will suffer from multiple repeated rewriters under correlated failures. For this purpose, the conclusion that we draw would be applicable to other systems adopting this technique, such as Flink and Samza.
- **Spark Streaming.** Spark Streaming supports persisting the intermediate outputs, which has a similar effect as upstream backup. This can avoid repeating the expensive source replaying for each node failure. We use this system to motivate that even with upstream buffering, it is still necessary to consider the availability of resources especially under correlated failures. Note that the conclusion would also be applicable to other systems, including Storm, Flink and Samza.
- **Storm (Blocking).** Storm would repeatedly perform source replay to recover the failed tasks under correlated failures. To prevent this
effect from influencing the recovery latency, we block the recovery until all the newly acquired resources are ready for recovery.  
- Incremental. Our prototype system that implements incremental and query-centric recovery.

To evaluate the performance of the proposed optimization algorithms in this paper, we compare BestDensity and the optimal Dynamic Programming algorithm (referred to as DP). We also design a baseline algorithm, which is denoted as OC. OC is implemented as a greedy algorithm that always tries to recover the partition with minimum recovery cost in topological order.

### 6.1 Necessity of Incremental Recovery

![Figure 6: Cumulative distribution of time to acquire new nodes.](image)

Firstly, we attempt to justify the necessity of performing incremental recovery by showing that, upon correlated failures, the availability of new resources takes time and comes in a gradual manner. We conduct this set of experiments on Amazon EC2, where the request of new resources can be issued immediately after failure. We record the time intervals from the moment when the request of new nodes is issued to the moment when the new nodes are ready to host processing tasks. We collect in total 180 samples and depict their cumulative distribution in Figure 6. One can see that, the time cost of acquiring a new node varies from 2 to 6 minutes. This result further consolidates our motivation for incremental recovery even on a cloud service with “virtually” unlimited resources, not to mention when failures occur at different time and/or the administrator needs to fix the software or hardware problems.

![Figure 7: Topology 1](image)

![Figure 8: Topology 2](image)

### 6.2 Incremental Recovery

In this part of experiments, we compare the recovery performance of incremental recovery and blocking operator-centric recovery upon correlated failures. Figure 8 shows the structure of the job topology, which consists of 15 queries. Each operator has 1 partition and the partition of $O_i$ is specified as the output partition for query $Q_i$. The checkpointing interval is set as 10 seconds. On receiving a tweet, the **Parser** emits a tuple for each hashtag within the tweet. Operators $O_1, O_2, O_3$, and $O_4$ conduct sliding-window aggregates, which store hashtag frequency with various window settings and output the updates. Operator $O_4$, $O_{14}$ maintain the states of sliding-window aggregates that they subscribe to. As the input rates of operators increase in the topological order, we set the emitting rate of the Source to 1,000 tweets per second to constrain the workloads of operators near the sinks (e.g., $O_{10}$-$O_{14}$). The experiment is started in a cluster of 10 nodes. To inject a correlated failure, we manually kill the 8 nodes where the output partitions of the 15 queries are deployed, and then deploy 8 new nodes for recovery.

![Figure 9: Storm: Repeated Source Drum.](image)

![Figure 10: Spark Streaming: In-Sufficient Recovery Resources](image)

We also conducted a set of experiments to study how the repeated source replay and insufficiency of computation resource influence the progress of failure recovery in existing systems. We use Storm in the experiments of repeated source replay and Spark Streaming in the experiments of resource insufficiency. The topology used in both experiments is depicted in Figure 7, where the hashtags in tweets are extracted in the parser operators. Each window operator subscribes the output stream of the parser and updates the frequencies of hashtags in its state with a window interval of 5 seconds. The cluster consists of 5 nodes. We use the end-to-end latency as performance metric in this set of experiments.

For the experiment with Storm, the checkpointing interval in Storm is set as 10 seconds, and the input rate is set as 200 tuples per second, such that we can study the influence of repeated source replay on processing latency even when the system is not overloaded. The message timeout interval is set as 30 seconds, which means that a tuple is replayed from the source if it has not been completely processed within 30 seconds after it is emitted. We run the experiments with 1, 2 and 3 consecutive node failures. The interval between any two consecutive failures is 60 seconds. The results are presented in Figure 9, where the x-axis denotes the timestamps of source tuples and the y-axis denotes the end-to-end latency. One can see that, the latency during recovery increases with the number of consecutive worker failures, as the recovery processes before the last failure are wasted due to repeated source replays and input tuples are queued before the last replay is started.

In the next experiment, we examine how Spark Streaming performs when node failures incur insufficiency of resources. We set the batch interval as 1 second, the checkpointing interval as 10 seconds. The input rate is set as 5,000 tuples per second, with which the system is not overloaded. As one can see in Figure 10, killing 1 node brings obvious increment on the end-to-end latency, which is caused by the rescheduling of failed RDDs. After killing 2 nodes, the latency starts increasing unboundedly, as the system is heavily overloaded and the size of buffered inputs keeps growing. This result shows that an eager recovery paradigm without considering the availability of resources cannot work well for in the case of correlated failures.
of this query. Denoting $l_b$ as a query’s latency before failures and $l_r$ as that after failures, its relative latency, $RL$, is calculated as $\frac{l_r}{l_b}$. After query $Q_i$ is recovered, $RL_i$ would gradually approximate 1. Within a time interval $\Delta T$, if the average $RL_i$ of $Q_i$ is smaller than $\Theta (\Theta = 1.2)$, $Q_i$ is considered as an Available Query, which means it has been recovered to a normal state.

In Figure 11, Incremental denotes our approach where the BestDensity algorithm is used to optimize the recovery plans. Furthermore, we use BestCase to denote the case where all the new nodes arrive after failure. More specifically, in the experiments of BestCase, we deploy 8 nodes before failure and keep them standby, the recovery of all the failed queries are started 3 minutes after the failure. Storm (Blocking) represents the case where the recovery is blocked until the last newly deployed node becomes available.

As one can see in Figure 11, BestCase always outperforms the other two approaches. This is because BestDensity starts recovering all the failed partitions earlier than the other approaches. On the other hand, Storm (Blocking) has the worst recovery performance as it blocks recovery after all the new nodes become available, which results in that Storm (Blocking) has more input tuples buffered than BestCase and Incremental before the recovery is started. The performance of incremental recovery is better than Storm (Blocking), as the failed queries are gradually recovered following the pace of resource acquiring. This experiment shows that, compared to blocking recovery, incremental recovery incurs lower recovery latency and takes less time to recover the failed queries.

To further study the performance of incremental recovery, we conducted recovery experiments with various settings of node failures based on the job topology in Figure 8. To have more flexibility on controlling the location and timing of node failures, we conduct this set of experiments under the local mode of Storm.

We first study if the location of failed partitions in topology influences recovery latency. We test with two failure scenarios: (1) partitions in 4 upstream operators, i.e. $O_8$, $O_9$, $O_2$ and $O_4$, are failed; (2) partitions in 3 downstream operators, i.e. $O_{12}$, $O_{13}$ and $O_{14}$, are failed. The input ratio is 100 tuples per second. The checkpointing interval is 10 seconds. Figure 12 presents the relative latency of the failed queries.

![Figure 11: Recovery of correlated failure where all the partitions in the 15 aggregate operators ($O_{15}, O_{14}, ..., O_1$) are failed.](image)

![Figure 12: Relative latency of failed queries after 2 node failures.](image)

Figure 13: Incremental Recovery: relative latency of normal and failed queries after 2 nodes are killed. (a) Operators are involved into the failure in reverse-topological order: $\{O_{14}, O_{13}, ..., O_0\}$; (b) Operators are involved into the failure in topological order: $\{O_0, O_1, ..., O_{14}\}$.

As one can see, Incremental outperforms Storm (Blocking) in both scenarios, as the latter blocks recovery until all the newly deployed nodes arrive. Figure 13 presents the relative latency of the failed queries and normal queries after incremental recovery is started. In Figure 13(a), the failures of downstream partitions have no influence on the latency of normal queries before the recovery is started. A fluctuation on the latency of normal queries occurs after the recovery is started. This is incurred by the operations of sending buffered tuples from the alive partitions to the recovered ones. If failures are located in the upstream of the topology, all the queries in the topology are failed. As shown in Figure 13(b), there is no output produced from any query until the recovery is started, because failures in the upstream partitions block the processing of the downstream ones. Once the recovery is started, the latencies of both failed and normal queries gradually return to a normal level.

Besides the location of failures, we also conduct experiments of incremental recovery with various number of failed nodes. Figure 14 presents the latency of failed queries using incremental recovery with different numbers of node failures. We expand the set of failed operators in both a topological (Figure 14(b)) and a reverse-topological order (Figure 14(a)). As one can see, in both failure settings, it takes a longer time for the failed queries to return to a normal state with more failed nodes. The fluctuations of relative latency are incurred by the gradually started recovery of failed operators, as the previously recovered partitions are requested to send the replay tuples to their downstream neighbors.

**6.3 Optimizing Recovery Plan**

To evaluate the proposed optimization algorithms, we implement a query generator which can generate topologies with various specifications, such as the number of queries, the size of each query, the distribution of query priority and the skewness of partition sharing frequency. For each set of the topology specifications, we generate 100 synthetic topologies. As the time complexity of the dynamic programming (DP) algorithm increases exponentially with the number
of queries, we set the number of queries in all the synthetic topologies as 18 so that DP can be complete in time. In Figure 15, BestDensity denotes the BestDensity algorithm that optimizes the recovery schedule. DP denotes the Dynamic Programming optimization algorithm. OC is an operator-centric optimization algorithm, which recovers the operator with the minimum recovery cost in topological order.

Figure 15(a) shows that the performance of BestDensity approximates DP in all the settings. This verifies our heuristic that queries with higher Profit Density should be assigned with higher priority during incremental recovery. As OC schedules the recovery in the operator-centric paradigm, it may add a partition that has the smallest recovery cost into the recovery plan, instead of the one that completes the recovery of a failed query. Therefore, OC has the worst performance in all the settings. Figure 15(a) shows that the performance of all the three algorithms are degraded by doubling the maximal sharing frequency of partition. This is because a query is only considered as recovered if and only if all its failed partitions are recovered. However, failed partitions with high share frequency could be shared by more queries. Figure 15(b) shows that, by increasing the skewness of partition sharing, the performance of BestDensity increases. As OC does not consider partition sharing, its performance decreases with higher skewness of partition sharing. The results presented in Figure 15(c) indicate that, compared to the case that query priorities are randomly distributed, there is a slight performance degradation of both BestDensity and DP when the priority of query proportionally increase with its recovery cost. This is because there exist queries that have low recovery costs but high priorities when the query priorities are randomly distributed.

7 RELATED WORK

Fault tolerance techniques in DSPSs can be generally categorized into two types [19]: passive and active approaches. Passive techniques include checkpointing [8, 15], upstream backup [2, 9, 26, 30, 31] and source buffering [8, 37]. Active approaches [5, 6, 35] employ hot-standby replicas to achieve faster failure recovery with higher resource consumption.

The authors of [20] propose the technique of delta checkpoints to compress the size of checkpoints. The work in [25] integrates fault tolerance with scaling operations in DSPSs such that the computation state stored in a checkpoint can be utilized to speedup workload migration. [6] studies the dynamic assignment of computation resources between primary and active replicas to optimize the trade-off between throughput and output quality with the existence of node failure. [45] proposes a hybrid approach that switches between active and passive standby modes to optimize recovery from transient failures. [38] selects a passive/active replication plan for each operator individually according to its estimated recovery cost. The work in [33] recovers operator states by replaying a distinct selection of source events, and therefore has to replay a large number of events when the states depend on all the historical stream data.

Parallel recovery [9, 43] partitions the workloads of a failed computation component into multiple pieces and restarts them in parallel to achieve fast recovery. Parallel recovery cannot solve the challenges posed by correlated failures in DSPSs. The incremental recovery proposed in this work is orthogonal to parallel recovery, though combining them could improve recovery efficiency.

Load balancing techniques [11, 13, 32] are explored to dynamically re-balance the distribution of workload in the processing cluster. The scheduling algorithm for incremental recovery proposed in this work prevents from overloading any node during recovery.

Spark Streaming [45] abstracts the processing of input streams as a sequence of RDDs. RDDs lost caused by failures can be recomputed following its lineage of generation. RDDs can be persisted to emulate upstream backup. As Spark Streaming adopts a lazy strategy on RDD transformation, it has no control over the order on how the failed RDDs are reconstructed. Drizzle [40] is a low-latency batch-based DSPS implemented on Spark Streaming. Our query-centric scheduling approach can be used on Spark Streaming and Drizzle to optimize the recovery order of failed RDDs upon correlated failures.

Storm [37] uses source buffering and checkpoint for fault tolerance, and does not guarantee exactly-once processing. Samza [30] achieves fault tolerance by adopting upstream backup and delta checkpoints. As duplicate processing may appear during failure recovery, Samza can only guarantee at-least-once processing. Trident [38] is a high-level abstraction built on top of Storm, which utilizes batching and anchoring techniques to guarantee exactly-once processing. Both Flink [8] and Naiad [27] rely on the techniques of consistent checkpoint and source buffering to guarantee exactly-once processing. However, the blocking recovery incurred by source buffering makes it unsuitable for recovering correlated failure. Our checkpointing scheme is similar with the approach used in Flink [8]. MillWheel [2] also adopts upstream backup. It assigns each input tuple a globally unique ID and guarantees exactly once processing with precise duplicate elimination. This approach incurs runtime overhead of maintaining upstream backup during normal state. All the above systems do not consider the optimization of recovering correlated failures.
Previous researches [16, 28, 34] indicate that there exist large-scale correlated failures in clusters and the scale of the failure is usually proportional to the number of physical nodes in the cluster. Although correlated failures are less frequent than the single node failures, they could obviously weaken the system availability [12] and incur significant latency to the DSFS applications. The work in [17] shows that, instead of simultaneous failing of multiple nodes, the failures of nodes involved within a correlated failure could span a short interval. Borg [41] studies how to reduce the possibility of correlated failures by assigning computation components of a job topology across multiple failure domains such as racks and power domains. Our approach addresses an orthogonal problem and is compatible with the resource allocation strategy proposed in Borg. [35] presents a hybrid fault-tolerance framework to recover correlated failure. In this framework, a subset of tasks are actively replicated such that they can be immediately recovered to produce tentative outputs after correlated failure. This paper focuses on minimizing the query latency on recovering correlated failures, which is orthogonal to the work in [35].

8 CONCLUSION

In this paper, we present a fault-tolerance framework for DSFSs that incrementally schedules the recovery of correlated failures. With incremental recovery, failed partitions are gradually recovered according to the arriving pace of recovery resources, such that the outputs of failed queries could be resumed as early as possible. We propose a query-centric approximate algorithm that takes partition sharing into consideration to optimize the scheduling of failure recovery. Experiments show that our approximate algorithm greatly outperforms the operator-centric one, especially when the recovery resources are limited and the recovery costs of partitions are skewed. Compared with the blocking recovery, queries failed in correlated failures could be recovered much faster with incremental recovery.

REFERENCES