Technical Report: Incremental Recovery in Distributed Stream Processing Systems

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

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

ABSTRACT

In a large-scale cluster, correlated failures usually involve a number of nodes failing simultaneously. Although correlated failures occur infrequently, they have significant effect 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 failure and do not consider the optimization for recovering correlated failure. In this work, we propose a 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.

1 INTRODUCTION

Distributed Stream Processing Systems (DSPSs), such as Flink [6], Samza [26], Apache Storm [33] and Spark Streaming [39], 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 [30]. Therefore, fault tolerance is an indispensable functionality in DSPSs.

The causes of correlated failures include failures of shared hardwares (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 [20, 25, 29, 32, 38]. Previous studies [10, 12, 15, 25, 29, 38, 40] show that, correlated failures, where a number of nodes fail simultaneously or within a short interval, have significant effect on systems' availability. Google [10] 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. [38] 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 [35] 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 [5], which can actively replicate messages in standby clusters across multiple data centers. Key-value stores such as Cassandra [21] and Megastore [2] 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 [7] and then perform load balancing [9, 11, 28] after failure recovery is completed, the decoupling of failure recovery, scaling out and load balancing would incur unnecessarily 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. For instance, we conducted experiments (Section 7.1) 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 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 just for the recovery of correlated failures is not cost-effective, because these resources have to stay idle most of the time.

Lastly, as reported in [38], 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 a 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 differences between correlated failures and independent failures for DSPSs and analyze 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. The recovery framework employs new techniques including adaptive buffering and order-preserving processing to support efficient incremental recovery and to 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 multiple sets of 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.

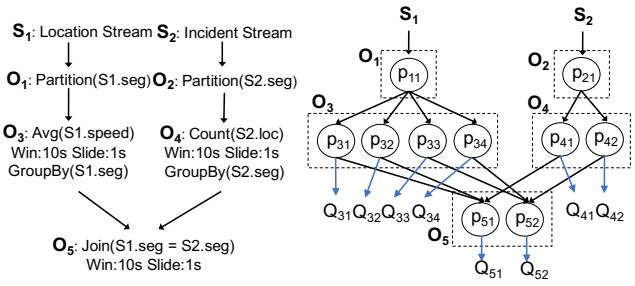


Figure 1: Example topology (left) and its execution plan (right). O_3, O_4 and O_5 are output operators.

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_5 are partitioned into two groups based on the road segment IDs, one for each partition of O_5. 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_{51}, any failed partition in the partition set $\{p_{11}, p_{31}, p_{21}, p_{41}\}$ must be recovered. Therefore, we further partition a logical query into query partitions for fine-grained recovery scheduling.

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_{51} consists of p_{11}, p_{21}, p_{31}, p_{41} and p_{51}. 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 are more critical than that of the other areas. Upon failure, it is beneficial to prioritize recovering the queries that monitor those critical areas. Therefore, we will consider query priorities when optimizing the recovery plan.

2.2 Challenges in Fault-Tolerance Design

Data buffering and checkpointing are the key techniques to achieve fault tolerance in DSPSs [6, 26, 33]. 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 [33], Flink [6] and Samza [26]. 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 [7] 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 overhead of adopting upstream backup includes the memory consumption of the output buffers and the cost of buffer management including appending new output and trimming the old ones.

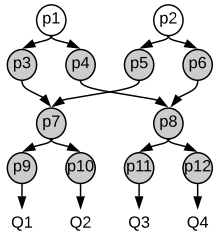
In this work, we propose adaptive buffering, a new buffering technique that only performs source buffering during normal execution. When a correlated failure is detected, upstream backup is activated to support incremental recovery. The switching between buffering approaches is light-weight and only 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 computation states of the failed tasks, checkpoints containing tasks' states should be periodically generated and persisted. For instance, Storm [33] generates checkpoints of the tasks periodically without any synchronization, thus the processing progress of different tasks at the time of checkpointing could be inconsistent. Therefore, in Storm, the progress of a restarted task could

fall behind that of its downstream neighbors. Thus duplicate elimination is necessary to achieve *exactly-once* processing.

Some systems, such as Flink [6], 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 topology state would 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 peers. This characteristic of correlated failures 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.



Node	Operator-Centric		Query-Centric	
	Schedule	Recovered Queries	Schedule	Recovered Queries
N1	{p3, p4}	n/a	{p3, p5}	n/a
N2	{p5, p6}	n/a	{p7, p9}	{Q1}
N3	{p7, p8}	n/a	{p4, p10}	{Q2}
N4	{p9, p10}	{Q1, Q2}	{p6,p8}	n/a
N5	{p11, p12}	{Q3, Q4}	{p11,p12}	{Q3, Q4}

Figure 2: An example illustrating operator-centric and query-centric recovery paradigms. Shadowed circles represent the failed partitions. N_i denotes the i'th node which becomes available for recovery. It is assumed that each node is capable of recovering 2 partitions. The table presents how the recovery is scheduled with the two paradigms, respectively.

2.2.3 Recovery Scheduling. Most existing DSPSs [6, 26, 33, 39] 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 one after another. As one can see, no sink operator partition is recovered until node N_4 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

In an incremental recovery scheme, if the amount of available resources is not enough to recover all the failed queries, only a subset of the failed partitions will be selected for recovery, which constitutes the first partial recovery plan. Whenever new recovery resources are acquired, another partial recovery plan consisting of some selected remaining failed partitions will be scheduled for execution.

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_i 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_i is calculated as follows: (1) calculate the ratio between the number of tuples processed by p_i 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_i. 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. A query is assigned a real number as its priority for recovery. In our implementation, it is an integer between 1 and 10 with the default

value as 1. 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 ⊆ 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 proof is presented in Appendix D.1. The details of our optimization algorithm are presented in Section 5.

3 SYSTEM OVERVIEW

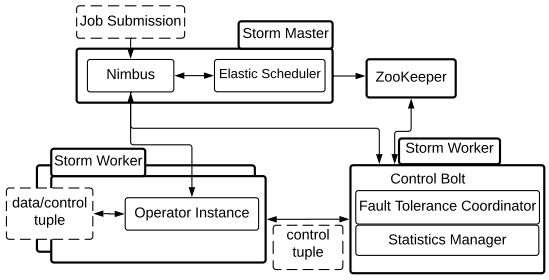


Figure 3: System Framework

We implemented our system on on top of Apache Storm [33]. 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

Similar to the work in [6], we use punctuations to trigger checkpointing of partitions in an asynchronous approach. By default, 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, $S_1, S_2, \cdots, S_n$. Once a punctuation P_k from S_i arrives at p_i, all the tuples p_i receives from S_i after P_k are stored in the input buffer before the checkpoint for P_k is generated in p_i. After p_i receives P_k from all of its input streams, it can confirm that p_i has processed all the inputs before P_k. Then a checkpoint containing the computation state of p_i is generated, and the fault tolerance coordinator is acknowledged. The coordinator tracks the checkpointing progress of the topology. Once the coordinator is acknowledged that all the partitions have completed the checkpointing for P_k, it is aware that a globally consistent checkpoint of the entire topology for P_k, denoted as $cp(P_k)$, is generated. This information of the successful consistent checkpoints are backed up in ZooKeeper by the coordinator for fault tolerance.

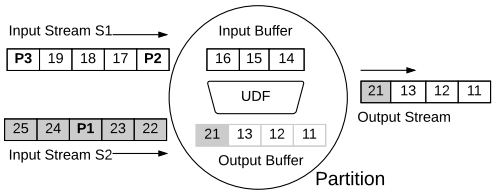


Figure 4: An example of checkpoint punctuation. The partition is denoted as a circle which has two input streams and one output stream. Within each stream, rectangles with P_i represents punctuation whose sequence number is i and rectangles containing an integer denotes a data item. UDF is the user-defined function.To differentiate from the data in S_1, data items in S_2 and their intermediate output are marked in gray.

Figure 4 presents an example about the input data and punctuations processed within a partition. In Figure 4, P_1 from S_1 has already arrived, hence the tuples from S_1, i.e. tuples 14, 15, and 16 are put into the input buffer, while the tuples from stream S_2 are still processed on their arrival. Once the punctuation P_1 from stream S_2 is received, the partition generates and stores the checkpoint for punctuation P_1. Then the partition starts processing the buffered input tuples in a FIFO manner.

4.2 Adaptive Buffering

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 2 following the observation in [10]. 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_i should emit to p_j. 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.

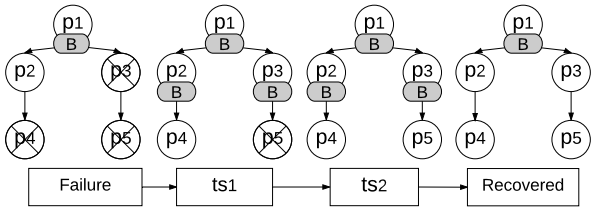


Figure 5: An example of adaptive buffering.

Figure 5 presents an example of adaptive buffering. Before the failure is detected, only the partition in the source operator (i.e., p_1), has output buffer. When the failures of p_3, p_4, and p_5 are detected, p_3 and p_4 are restarted at timestamp ts_1 and the output buffers are turned on in partition p_2 and p_3. All the partitions can continue their processing instead of being block to wait for the recovery of p_5. At ts_2, when new resources arrive and p_5 is restarted, it will process the tuples buffered in partition p_3. After a new global checkpoint is successfully generated, the output buffers are disabled in all the non-source partitions.

4.2.3 Overhead. In general, dynamically enabling and disabling upstream buffers requires synchronization among the parallel partitions to ensure correctness. Therefore, it may incur synchronization overhead. However, adaptive buffering only enables the upstream buffers at the moment that all the partitions have to be rollbacked to the latest consistent checkpoint to recover failures, which means their progress are already synchronized. Therefore our approach does not incur additional synchronization overhead. Furthermore, the disabling of buffering only involves deleting the FIFO queues,

Algorithm 1: Order-Preserving Processing

```
Input: p_i: parition; t: input tuple; InputList: list of p_i's input streams
       sorted by the IDs of the source partitions;
BID: the id of the current batch;
1  j ← t.BatchId;
2  k ← t.InputStreamId;
3  if t.seq ≤ lastSeq[k] then
4      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.            */
5  if t.type == Data Tuple then
6      IB[j][k].put(t);
       /* IB[j][k] is the input buffer for batch B_j from input
          stream IS_k;                                         */
7  else if t.type == Barrier then
8      BO[j] + +;
       /* BO[j] counts the number of received Barrier messages for
          batch B_j;                                           */
9  if BO[BID] == InputList.size; then
       /* when all the Barrier messages for batch BID have
          arrived;                                             */
10     foreach input stream IS_l ∈ InputList do
11         while IB[BID][l].hasNext() do
12             t' ← IB[BID][l].pull();
13             p_i.process(t');
14             lastSeq[l] ← t'.seq;
15     foreach output stream of p_i do
16         send Barrier message for batch BID;
```

which 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, after the coordinator detects 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 the 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 challenge of duplicate elimination to guarantee exactly-once processing. As correlated failures may trigger the replaying of a partition more than once, hence its downstream partitions would receive duplicate input data in this case. In addition, as a partition may produce outputs with various orders across different replays, its downstream partitions cannot easily detect duplicates based on the arriving order of the inputs. In this section, we present our solution, referred to as *order-preserving processing*, for 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 a set of consecutive, non-overlapping 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.

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 ($line\ 5 - line\ 6$). When p_i receives the *Barrier* message of the same batch from all its input streams, p_i starts processing tuples within this batch in a predefined round-robin order across the input streams ($line\ 9 - line\ 14$). 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 ($line\ 3 - line\ 4$).

The validity of adopting 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 can not 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 rollbacked 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 [6, 24, 33], which use in-order message delivery mechanism.

A batch-based approach in stream processing may generally incur additional processing latency [8]. 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.

Algorithm 2: ExecuteRecoveryPlan

```
Input: RP: recovery plan; T : topology;
foreach Task t_i in topology T do
    if RP.globalRollback == true then
        foreach failed partition p_j which should be recovered on t_i do
            Install partition p_j on task t_i with RP.checkpoint;
            p_j.buffering ← true;
            p_j.orderPreserve ← true;
        foreach alive partition p_j on t_i do
            clear the input and output buffer;
            rollback the state of p_j with RP.checkpoint;
            p_j.buffering ← true;
            p_j.orderPreserve ← true;
    else
        foreach failed partition p_j which should be recovered on t_i do
            install partition p_j on task t_i with RP.checkpoint;
            p_j.buffering ← true;
            p_j.orderPreserve ← true;
    foreach partition p_j in task t_i do
        foreach downstream neighboring partition p_k of p_j do
            if partition p_k is scheduled for recovery in RP then
                Buf_j^k ← output tuples buffered for p_k in partition p_j;
                send Buf_j^k to the destination task of p_k;
```

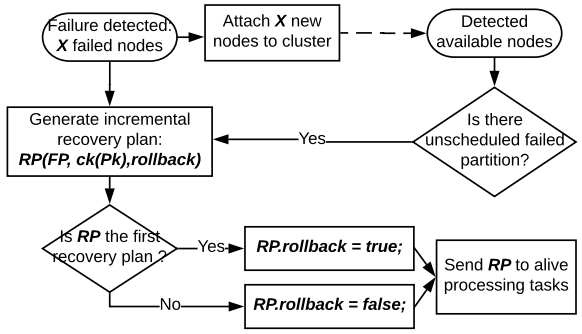


Figure 6: Flowchart of Failure Recovery. FP **is the set of failed partitions scheduled for recovery,** $ck(P_k)$ **denotes the latest global checkpoint.**

4.4 Incremental Recovery

Figure 6 depicts a flowchart showing 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 $\boldsymbol{X}$ new nodes to scale out the processing cluster, where $\boldsymbol{X}$ is the number of failed nodes. 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 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 [19], or persisting the inputs locally. We prove that our approach can guarantee exactly-once processing during recovery in Appendix D.3.

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 the DP algorithm are presented in Appendix B. Considering that the DP algorithm is too expensive for large-scale topologies, we present an efficient approximate algorithm (Algorithm 3), which is referred to as BestDensity.

BestDensity prunes the tremendously huge search space by constructing an initial set of candidate plans denoted as SC, which consists of all the combinations of 2 failed queries whose recovery costs are smaller than the amount of available resources. As a partition can be shared by multiple queries, it is natural to prioritize recovering the queries whose failed partitions are shared by more queries. Furthermore, as the failed queries have various recovery costs and priorities, the profit that can be achieved by using per unit of resource should be maximized while expanding the partial recovery plan. With the above two considerations, we define the *Profit Density* of query Q_i, denoted by PD_i, as follows:

Definition 3. *Profit Density:*

$$PD_i = \frac{prt_i}{\sum_{p_k \in Q_j} \frac{c_k}{f_k}}$$

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

Algorithm 3: BestDensity(R, FP)

```
Input: R: amount of available resources; FP : set of failed partitions;
Output: RP : recovery plan;
1  n ← 0; CP_n ← ∅; SC ← {CP_n};
   /* CP_0 is the initial empty candidate plan; SC is the set of
      candidate plans.                                              */
2  foreach partition p_i ∈ FP do
3      c_i ← the resource consumptions of p_i;
4  M ← 0;
   /* M denotes the number of failed queries;                       */
5  foreach query Q_i do
6      if Q_i contains failed operator then
7          M++; C_i ← the resource consumption of recovering all the failed
             partitions in Q_i;
8          PD_i ← Profit_Density(Q_i, FP);
9  Find Q_k such that PD_k ≥ max {PD_i, where C_i ≤ R};
10 CP_n ← CP_n ∪ {Q_k}; n++;
11 for i ← 1 to M − 1 do
12     Q_m ← ith failed query;
13     for j ← i + 1 to M do
14         Q_n ← jth failed query;
15         C_{m,n} ← the resource consumption of recovering Q_m and Q_n
             concurrently;
16         if C_{m,n} ≤ R then
17             CP_n ← CP_n ∪ {Q_m, Q_n};
18             SC ← SC ∪ {CP_n};
19             n++;
   /* Enumerate all the combinations of 2 failed queries where the
      sum of their recovery costs is smaller than R;                */
20 for i ← 1 to |SC| do
21     CP_i ← the ith plan in SC;
22     found ← false; maxDens ← 0; index = 0;
23     while Cost(CP_i) ≤ R do
24         foreach Failed query Q_m ∉ CP_i do
25             if Cost(CP_i) + C^i_m ≤ R & PD^i_m ≥ maxDens then
26                 index ← m; found ← true; maxDens ← PD^i_m;
               /* C^i_m and PD^i_m denote the cost and profit desity
                  of Q_m calculated under the condition that only
                  the queries in CP_i are recoverd;                 */
27         if found then
28             CP_i ← CP_i ∪ Q_index;
29             maxDens ← 0;
30         else
31             break;
32 return the recovery plan RP, RP ∈ SC, where the sum of priorities of the
     recovered queries in RP is the maximum among SC;
```

and a recovery cost smaller than $R - Cost(CP_i)$. 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. The details of this algorithm are presented in Algorithm 3.

BestDensity starts by calculating the recovery cost and profit density of each failed query ($line\ 2 - line\ 8$). 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 ($line\ 9 - line\ 10$). 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 ($line\ 11 - line\ 19$). Denoting the number of failed queries as M, the number of initial candidate plans is smaller than $\frac{M\cdot(M-1)}{2} + 1$. Next, each candidate plan in SC is gradually extended by adding the failed query with the largest profit density under the resource constraint ($line\ 21 - line\ 31$). 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 $O\left(M^3 \cdot logM\right)$, where M is the number of failed queries. The approximate ratio of Algorithm 3 is given in Theorem 5.1.

Theorem 5.1. *Algorithm 3 achieves an approximation ratio of* $\left(1 - e^{-\frac{1}{d}}\right)$*, 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:

$$\sum_{Q_i \in RQ} prt_i \geq \left(1 - e^{-\frac{1}{d}}\right) \cdot \sum_{Q_j \in RQ^*} prt_j$$

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. The proof of Theorem 5.1 can be found in Appendix D.2.

During the incremental recovery, an upper bound (e.g., 80%) is set as the maximally 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 RECOVERY OF COMPLETE FAILURE

Complete failure of a processing cluster could be incurred by correlated failures (e.g., rack failures) within a data center or the outage of entire data center. In this section, we present how to extend our approach to facilitate recovery from a complete failure of the processing cluster.

To guarantee exactly-once processing, one must ensure that there is no loss of input data during the recovery process. Furthermore, the persisted checkpoints of the topology state should also be available in the case of complete failure. Kafka is often used as fault-tolerant input cache in streaming applications. Here we use a multi-cluster Kafka solution [5] to replicate input data in a standby Kafka cluster, which could be deployed in a different data center. In addition, Kafka is also used to persist checkpoints across clusters. Alternative approaches to replicate checkpoints include cache service (e.g., Redis) and distributed file system (e.g., HDFS), which could also be deployed across data centers. ZooKeeper is used to store the topology configuration, checkponting progress

and statistics of resource consumptions. Note that deploying a standby Kafka cluster also requires to set up at least one local ZooKeeper server. For the sake of fault tolerance, the information that is supposed to be stored in ZooKeeper should be replicated in the ZooKeeper servers deployed for the standby Kafka cluster.

Complete failures are detected utilizing the heartbeat mechanism. All the nodes in the primary processing cluster periodically send heartbeats to ZooKeeper servers in the standby cluster. A monitoring daemon reads these heartbeats and starts the recovery if there is no heartbeat from any node after a configurable timeout. On detecting a complete failure, the first step of recovery is to acquire computation resources. The processing cluster should be recovered in a data center where the standby Kafka cluster is located, which could also be the data center where the failed cluster was originally deployed. This acquisition of recovery resources is specified by pre-written script. The Nimbus daemon of the processing cluster is started on the first available node. After reading topology configuration from ZooKeeper, Nimbus restarts the control bolt on the same node. Whenever a new node joins the processing cluster, the elastic scheduler in Nimbus starts a task for each operator in the topology on this new node. The control bolt broadcasts the appearance of the new node in the entire processing cluster and only treats it as available until all the nodes in the cluster acknowledge their awareness of this new node. With the information on checkpointing progress in ZooKeeper, the control bolt selects the latest global checkpoint to restore the states of partitions when they are restarted. The control bolt also reads the resource consumption of partitions stored in ZooKeeper, which are necessary input parameters when optimizing the incremental recovery plan. Following the arriving pace of new nodes, failed partitions are scheduled to be incrementally recovered according to the approach described in Section 4.4.

Upon a complete failure, an intuitive recovery approach is to block the recovery until all the newly acquired nodes are available. Incremental recovery can speed up the production of final query outputs. Compared with the solution of maintaining a standby processing cluster, incremental recovery is more resource efficient, especially when the scale of the processing cluster is large. Recovery from complete failure has not been fully implemented in our prototype system. We list it as one of our future work.

7 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. A real dataset consisting of 569, 382 tweets crawled from Twitter is used as the source. There are in total 894, 098 hashtags in this dataset, of which 309, 555 are identical. The tweets 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 replays 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 buffering. 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 Programing 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. Furthermore, we also examine the influence of fault-tolerance operations on the system throughput and latency. The results are presented in Appendix C due to limited space.

7.1 Necessity of Incremental Recovery

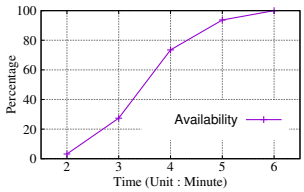


Figure 7: Cumulative distribution of time to acquire new nodes.

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 7. 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.

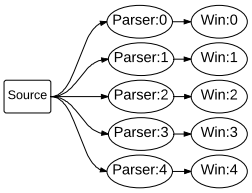


Figure 8: Topology 1

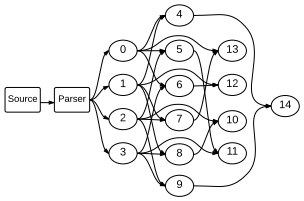


Figure 9: Topology 2

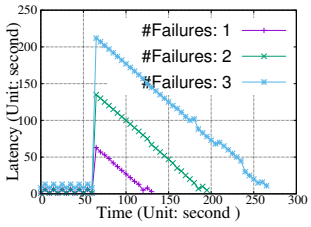


Figure 10: Storm: Repeated Source Replay

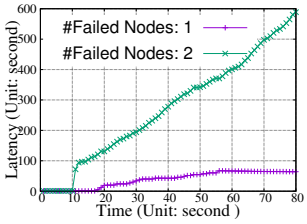


Figure 11: Spark Streaming: Insufficient Recovery Resources

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 8, 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.

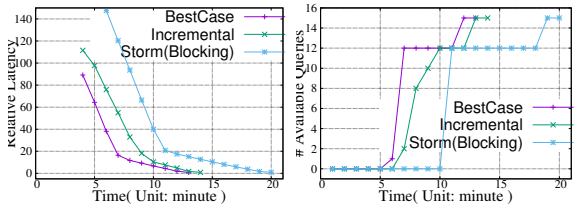
For the experiment with Storm, the checkpointing interval in Storm is set as 10 seconds, and the input rate is set as 200 tuples per second, such that we can study the influence of repeated source replay on processing latency even when the system is not overloaded. The message timeout interval is set 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 10, 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. As one will see in our later experiments, such as Figure 15, a desirable recovery strategy is to only replay input of the failed partitions on each occurrence of node failure, hence each node failure only brings a small increase to the latency of the affected tuples.

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 11, 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 is not helpful for failure recovery, especially in the cases of correlated failures.

7.2 Incremental Recovery

In this part of experiments, we compare the recovery performance of incremental recovery and blocking operator-centric recovery upon correlated failures. Figure 9 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. The *Source* operator emits tweets in the rate of 1,000 tweets per second. 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 count hashtag frequency with various window settings and output the updates. Operator O_i, $4 \leq i \leq 14$, maintains the states of sliding-window aggregates that it subscribes to. Initially, we deploy 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.

We use two metrics to measure the effectiveness of failure recovery. **Relative Latency** measures the difference between a query's latency before and after failures. A query's latency is calculated as the average end-to-end latency of the input tuples that contribute to the output of this query. Denoting l_s as a query's latency before failures and l_r as that after failures, its relative latency, RL, is calculated as $\frac{l_r}{l_s}$. After query Q_i is recovered, RL_i would gradually approximate 1. Within a time interval Δ_T, if the average RL_i of Q_i is smaller than Θ ($\Theta = 1.2$), Q_i is considered as an **Available Query**, which means it has been recovered to a normal state.



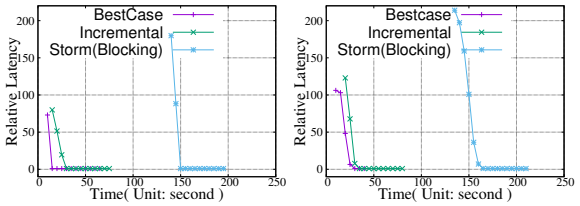
(a) Average relative latency (b) #Available queries

Figure 12: Recovery of correlated failure where all the partitions in the 15 aggregate operators (O_0, O_1, ..., O_{14}) are failed.

In Figure 12, *Incremental* denotes our approach where the *BestDensity* algorithm is used to optimize the recovery plans. Furthermore, we use *BestCase* to denote the case that all the new nodes arrive soon 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.

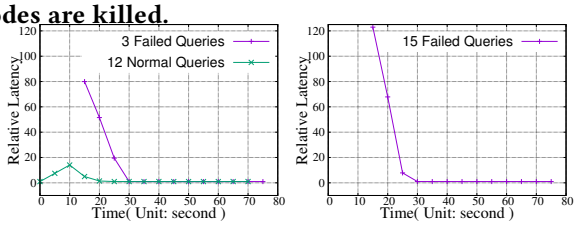
As one can see in Figure 12, *BestCase* always outperforms the other two approaches. This is because *BestDensity* starts recovering all the failed partitions earlier than the other approaches. On the other hand, *Storm (Blocking)* has the worst recovery performance as it blocks recovery after all the 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 for the failed queries to recover to a normal state.

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 9. 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.



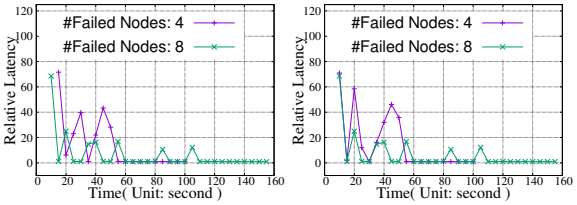
(a) Failure in Downstream (b) Failure in Upstream

Figure 13: Relative latency of failed queries after 2 nodes are killed.



(a) Failure in Downstream (b) Failure in Upstream

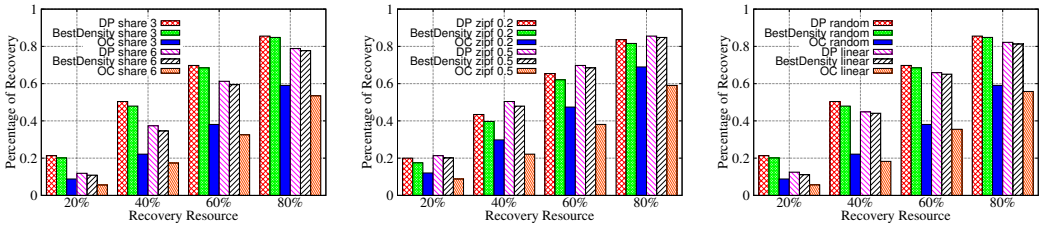
Figure 14: Incremental Recovery: relative latency of normal and failed queries after 2 nodes are killed.



(a) Reverse-Topological (b) Topological

Figure 15: Incremental recovery: relative latency of failed queries after 4 and 8 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: $\{O_0, O_1, ..., O_{14}\}$.

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_0, O_1, O_2 and O_3, are failed; (2) partitions in 3 down stream 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 13 presents the relative latency of the failed queries. As one can see, *Incremental* outperforms *Storm (Blocking)* in both scenarios, as the latter blocks recovery until all the newly deployed nodes arrive. Figure 14 presents the relative latency of the failed queries and normal queries after incremental recovery is started. In Figure 14(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



(a) Share Frequency (b) Skewness (c) Query Priority

Figure 16: The x-axis indicates the percentage of the amount of recovery resources to the sum of the resource consumption of failed partitions. The y-axis represents the ratio of the sum of recovered queries' priorities to that of all the failed queries. (a): The maximal share frequency of a partition is set as 3 or 6. (b): The share frequencies of partitions follow Zipfian distribution with the skewness parameter, s, set as 0.2 or 0.5. (c): Query priority is either set as a random value between 1 and 10 or linearly increases from 1 to 10 according to the workloads of its operators.

are located in the upstream of the topology, all the queries in the topology are failed. As shown in Figure 14(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 15 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 15(b)) and a reverse-topological order (Figure 15(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.

7.3 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 16, *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 always tries to recover the operator with the minimum recovery cost in topological order.

Figure 16(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 16(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 16(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 16(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.

8 RELATED WORK

Fault tolerance techniques in DSPSs can be generally categorized into two types [17]: passive and active approaches. Passive techniques include checkpointing [6, 14], upstream backup [1, 7, 23, 26, 27] and source buffering [6, 33]. Active approaches [3, 4, 31] employ hot-standby replicas to achieve faster failure recovery with higher resource consumption.

The authors of [18] propose the technique of delta checkpoints to compress the size of checkpoints. The work in [22] integrates fault tolerance with scaling operations in DSPSs such that the computation state stored in a checkpoint can

be utilized to speedup workload migration. [4] 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. [41] proposes a hybrid approach that switches between active and passive standby modes to optimize recovery from transient failures.

Parallel recovery [7, 39] partitions the workloads of a failed computation component into multiple pieces and restarts them in parallel to achieve fast recovery. Parallel recovery is not proposed to solve the challenges posed by correlated failures in DSPSs. The scheduling of recovery order is still necessary for DSPSs even with parallel recovery, as recovery resources may not arrive simultaneously after correlated failure. The techniques of incremental recovery proposed in this work are orthogonal to parallel recovery, though combining them together could improve recovery efficiency.

DSPSs could suffer from poor performance when the workload distribution is skewed. Load balancing techniques [9, 11, 28] 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 [39] 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. Drizzle [36] is a low-latency batch-based DSPS implemented on Spark Streaming. By grouping the scheduling of multiple micro-batches together, Drizzle minimizes the overhead of centralized batch scheduling in Spark Streaming. As Spark Streaming adopts a lazy strategy on RDD transformation, it has no control over the order on how the failed RDDs are reconstructed. Our query-centric scheduling approach can be used on Spark Streaming and Drizzle to optimize the recovery order of failed RDDs, which can speed up the production of final query outputs upon correlated failures.

Storm [33] uses source buffering and checkpoint for fault tolerance. However, it does not guarantee exactly-once processing. Samza [26] 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 [34] is a high-level abstraction built on top of Storm, which utilizes batching and anchoring techniques to guarantee exactly-once processing. Both Flink [6] and Naiad [24] persist the topology states by generating and storing consistent checkpoints and utilize the technique of 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 [6]. MillWheel [1] also adopts upstream backup. It performs duplicate elimination by assigning 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 [15, 25, 30] 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 the appearance of correlated failures is less frequent than the single node failures, it could greatly harm the system availability [10] and incur significant latency to the streaming data applications running on DSPSs. The work in [16] shows that, instead of simultaneous failing of multiple nodes, the failures of nodes involved within a correlated failure could span a short interval. Borg [37] 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. [31] 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 latency of queries while recovering correlated failures, which is orthogonal to the problem studied in [31].

9 CONCLUSION

In this paper, we present a fault-tolerance framework for DSPSs that incrementally schedules the recovery of correlated failures with a query-centric paradigm. With incremental recovery, failed partitions are gradually recovered according to the arriving pace of recovery resources, such that the outputs of the affected 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. Experimental results indicate 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

[1] Tyler Akidau, Alex Balikov, et al. MillWheel: Fault-tolerant Stream Processing at Internet Scale. In *VLDB '2013*.

[2] Jason Baker, Chris Bond, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR'2011*.

[3] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD '2005*.
[4] Paolo Bellavista, Antonio Corradi, Spyros Kotoulas, and Andrea Reale. Adaptive Fault-Tolerance for Dynamic Resource Provisioning in Distributed Stream Processing Systems. In *EDBT '2014*.
[5] Yeva Byzek. Disaster Recovery for Multi-Datacenter Apache Kafka Deployments. In *Technical Report '2017*.
[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. In *VLDB '2017*.
[7] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *SIGMOD '2013*.
[8] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive Stream Processing Using Dynamic Batch Sizing. In *SOCC '2014*.
[9] Junhua Fang, Rong Zhang, et al. Parallel Stream Processing Against Workload Skewness and Variance. In *HPDC'2017*.
[10] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI '2010*.
[11] Tom Z.J. Fu, Jianbing Ding, et al. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *ICDCS'2015*.
[12] Matthieu Gallet, Nezih Yigitbasi, Bahman Javadi, Derrick Kondo, Alexandru Iosup, and Dick H. J. Epema. A Model for Space-Correlated Failures in Large-Scale Distributed Systems. In *Euro-Par '2010*.
[13] Olivier Goldschmidt, David Nehme, and Gang Yu. 1994. Note: On the set-union knapsack problem. *Naval Research Logistics (NRL)* 41, 6 (1994), 833–842.
[14] Yu Gu, Zhe Zhang, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. An Empirical Study of High Availability in Stream Processing Systems. In *Middleware '2009*.
[15] Taliver Heath, Richard P. Martin, and Thu D. Nguyen. Improving Cluster Availability Using Workstation Validation. In *SIGMETRICS '2002*.
[16] E. Heien, D. LaPine, D. Kondo, B. Kramer, A. Gainaru, and F. Cappello. 2011. Modeling and tolerating heterogeneous failures in large parallel systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[17] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE '2005*.
[18] J. H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE '2007*.
[19] Kafka. 2018. (2018). Retrieved June, 2018 from http://kafka.apache.org/
[20] M. Kalyanakrishnam, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Failure Data Analysis of a LAN of Windows NT based Computers. In *SRDS '1999*.
[21] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* (2010).
[22] Kasper Grud Skat Madsen and Yongluan Zhou. Dynamic Resource Management In a Massively Parallel Stream Processing Engine. In *CIKM '2015*.
[23] André Martin, Andrey Brito, and Christof Fetzer. Scalable and Elastic Realtime Click Stream Analysis Using StreamMine3G. In *DEBS '2014*.
[24] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *SOSP '2013*.
[25] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *NSDI '2006*.
[26] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. In *VLDB '2017*.
[27] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys '2013*.
[28] Nicoló Rivetti, Leonardo Querzoni, et al. Efficient Key Grouping for Near-optimal Load Balancing in Stream Processing Systems. In *DEBS '2015*.
[29] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. Failure Data Analysis of a Large-Scale Heterogeneous Server Environment. In *DSN '2004)*.
[30] Bianca Schroeder and Garth A. Gibson. A Large-scale Study of Failures in High-performance Computing Systems. In *DSN '2006*.
[31] Li Su and Yongluan Zhou. Tolerating Correlated Failures in Massively Parallel Stream Processing Engines. In *ICDE '2016*.
[32] Dong Tang and Ravishankar K. Iyer. 1992. Analysis and Modeling of Correlated Failures in Multicomputer Systems. *IEEE Trans. Computers* 41, 5 (1992), 567–577.
[33] Ankit Toshniwal, Siddarth Taneja, et al. Storm@Twitter. In *SIGMOD '2014*.
[34] Storm Trident. 2018. (2018). Retrieved June, 2018 from http://storm.apache.org/
[35] Kaushik Veeraraghavan, Justin Meza, et al. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *OSDI'2018*.
[36] Shivaram Venkataraman, Aurojit Panda, et al. Drizzle: Fast and Adaptable Stream Processing at Scale. In *SOSP '2017*.
[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *EuroSys '2015*.
[38] Nezih Yigitbasi, Matthieu Gallet, et al. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *ICDCS '2010*.
[39] Matei Zaharia, Tathagata Das, et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP '2013*.
[40] Yanyong Zhang, Mark S. Squillante, et al. 2004. Performance Implications of Failures in Large-Scale Cluster Scheduling. In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop*.
[41] Zhe Zhang, Yu Gu, et al. A Hybrid Approach to High Availability in Stream Processing Systems. In *ICDCS '2010*.

A RECOVERY OF COMPLETE FAILURE

B DYNAMIC PROGRAMMING

In this section, we present the details of the Dynamic Programming algorithm to solve the Partial Recovery problem, which is described in Algorithm 4. This algorithm starts by calculating the recovery cost of each failed query (lines 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 (lines 19–22). 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 SC is returned.

In this algorithm, the size of the candidate plan set SC increases exponentially with the number of queries in the topology. To reduce the scanning cost, we remove CP_i from SC if all of the potential expansions based on CP_i have been considered ($line$ 24). The time complexity of this algorithm is bounded by $O\left(2^M\right)$, where M is the number of failed queries.

Algorithm 4: Dynamic Programming: DP(R, FP)

Input: **R**:Amount of available resources; **FP** :Set of failed partitions;
Output: **RP** :Recovery plan;

1 $usage \leftarrow 0$;
2 $n \leftarrow 0$; $CP_n \leftarrow \emptyset$; $SC \leftarrow \{P_n\}$;
/* $usage$ is the amount of currently used resources; n is the index of the next candidate recovery plan; CP_0 is the initial empty candidate plan; SC is the set of candidate plans. */
3 **foreach** *Query* Q_i **do**
4 $QT_{ni} \leftarrow \emptyset$;
5 $C_{ni} \leftarrow 0$;
6 **foreach** *Partition* $p_j \in FP$ **do**
7 **if** p_j *belongs to* Q_i **then**
8 $QT_{ni} \leftarrow QT_{ni} \cap \{p_j\}$;
9 $C_{ni} \leftarrow C_{ni} + Cost(p_j)$;
10 **while** $uasge < R$ **do**
11 $usage \leftarrow usage + 1$;
12 **foreach** *candidate plan* CP_i *in* SC **do**
13 $dif \leftarrow usage - Cost(CP_i)$;
14 $U_i \leftarrow \max\{C_{ij} \mid C_{ij} \neq 0 \ \& \ QT_{ij} \nsubseteq CP_i\}$;
15 **if** $dif \leq U_i$ **then**
16 **foreach** *Query* $Q_k \in \{Q_k \mid C_{ik} == dif\}$ **do**
17 $n++$; $CP_n \leftarrow CP_i \cup \{QT_{ik}\}$;
18 $SC \leftarrow SC \cup \{CP_n\}$;
19 **foreach** *Query* Q_m **do**
20 update QT_{nm} and C_{nm};
21 **else**
22 remove CP_i from SC;
23 return the recovery plan RP, $RP \in SC$, where the sum of priorities of the recovered queries in RP is the maximum among SC;

C OVERHEAD OF FAULT TOLERANCE

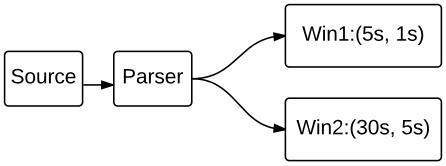
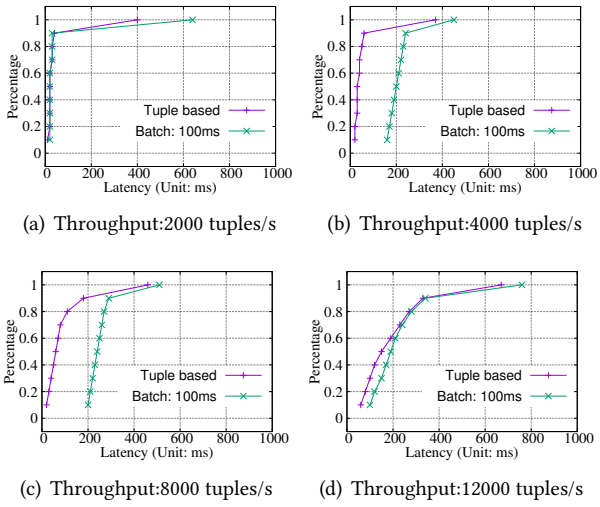


Figure 17: Topology used in the performance experiments. Win(T, S) indicates the period and slide of window are T and S, respectively.

We conduct experiments to study the overhead of checkpointing during normal processing and order-preserving processing during the failure recovery. We use the tweet data set described in Section 7. The topology used in this set of experiments is depicted in Figure 17. The *Source* operator emits tweets into the system in a user-specified rate. The *Parser* operator extracts and emits hashtags of each input tweet to the downstream operators. *Win1* and *Win2* are stateful sliding window operators that count the frequency of hashtags. The *Source* operator has 1 partition and the parallelization degree of other operators are 4. The latency of a tuple is recorded as the interval between the moment when it is emitted by the source operator and the time when it has been completed processed by both window operators. We deploy a cluster with 10 m3.large nodes in this set of experiments.

C.1 Order-Preserving Processing & Adaptive Buffering



(a) Throughput:2000 tuples/s (b) Throughput:4000 tuples/s

(c) Throughput:8000 tuples/s (d) Throughput:12000 tuples/s

Figure 18: Throughput and latency of our system with/without order-preserving processing and adaptive buffering.

Order-preserving processing and adaptive buffering are activated simultaneously during incremental recovery. To evaluate their overhead, we namually switch on order-preserving processing and adaptive buffering and compare the system performance with the configuration of tuple-based processing without output buffer. The checkpointing interval is set as 20 seconds in this set of experiments. The batch interval is set as 100 ms. We run the topology with different data input rates until the system reaches its maximal throughput. The results are presented in Figure 18, where *Batch* : 100ms denotes the case that order-preserving processing and adaptive buffering are activated and *Tuplebased* represents the case of tuple-based processing without output buffer.

Results show that the both cases show that both can achieve a maximal throughput of 12, 000 tuples per second. As expected, the tuple-based processing has advantage in latency performance, especially when the workload of

the system is under pressure. This is because, with order-preserving processing, tuples are split into batches and a batch is processed only after all the tuples belonging to this batch have arrived, which would increment the latency but only brings little influence on system throughput. On the other hand, maintaining output buffer also brings negative effect on latency. However, as order-preserving processing and adaptive buffering are only activated during incremental recovery, where throughput is the main factor determining the recovery efficiency, the latency increment caused by order-preserving processing and adaptive buffering is almost negligible comparing with the latency incurred by failure detection and acquiring new recovery resources.

C.2 Checkpointing

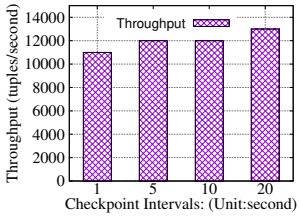


Figure 19: Throughput

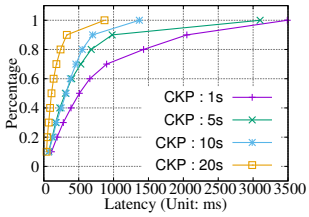


Figure 20: Latency

We conduct experiments with different checkpointing intervals to study the overhead of checkpointing operations during normal processing. The results of maximal throughput that the system can achieve with different checkpointing intervals are presented Figure 19. As one can see, the system has a lower throughout with a smaller the checkpointing interval. This is mainly because of the serialization and deserialization expenses of checkpointing. We also recorded the end-to-end latency of the system with different checkpointing intervals while the throughout is set as 11,000 tuples per second, which the system can handle with all the tested checkpointing intervals. The results presented in Figure 20 show that the best latency performance is achieved when the checkpointing interval is set as 20 seconds. The 99-percentile latency of the case with a 20-second checkpointing interval is three times shorter than the case with a 1-second checkpointing interval. Although setting a short checkpointing interval could reduce the number of replayed tuples during recovery, system performance could be harmed due to frequent checkpointing. For recovering correlated failure, the arriving pace of available resources dominates the recovery efficiency, hence the impact of checkpointing intervals on recovery efficiency is limited.

D PROOF

D.1 NP-hardness of Recovery Scheduling

Proof. A query is recovered if and only if all of its failed partitions are recovered. Given a recovery plan RP, the set of recovered queries is denoted as RQ. A partition in FP could be shared by multiple queries in RQ. The priority of query Q_j is represented as prt_j. For a failed partition $p_k \in FP$, its resource consumption is represented as c_k. Denoting the amount of available resources as R, the problem of recovery scheduling can be formalized as:

$$\begin{cases} \sum_{p_i \in RP} c_i \ \leq R \\ maximize \ \sum_{Q_j \in RQ} prt_j \end{cases} \tag{1}$$

The above problem is NP-hard, as it can be reduced from the NP-hard knapsack problem in polynomial time [13]. □

D.2 Approximation Ratio of BestDensity

In this section, we present the proof for Theorem 5.1. Let RP^* be the optimal recovery plan generated under resource constraint B. Sort the queries in RP^* in the descending order according to their profit density $PD_i = \frac{prt_i}{\sum^{p_k \in Q_j} \frac{C_k}{f_k}}$, where C_k is the recovery cost of p_k and f_k is the number of failed queries that share p_k. Let f be the maximum number of queries that a partition could be shared by. Let RP be the first plan that consists of the first 2 queries in RP^*. Let RP' be the set of queries generated by the *BestDensity* algorithm to RP, which is denoted by $Q_1, Q_2, \ \dots \ Q_r$. Let Q_{r+1} be the first query from RP^* RP that is not added into RP' due to resource constraint.

We present the following lemmas and their proofs before giving the proof of Theorem 5.1.

Lemma 1. *For each* $i = 1, \ \dots, r+1$, *the following holds*

$$PD_i \geq \frac{W_i}{B} \cdot \left(|RP^* \backslash RP| - \sum_{j=1}^{i-1} PD_j \right)$$

Proof. For all $Q_j \in RP^* \backslash \{RP \cup \{Q_1, \ \dots \ Q_{i-1}\}\}$, we have the following:

$$\frac{PD_j}{W_j} \leq \frac{PD_i}{W_i}$$

$$W\left(RP^* \backslash \left\{RP \cup RP'_{i-1}\right\}\right) \leq B$$

Therefore,

$$\begin{aligned} |RP^*\backslash RP| - \sum_{j=1}^{i-1} PD_j &= \sum PD_j, where \\ Q_j \in RP^* \backslash \{RP \cup \{Q_1, \ ..., \ Q_{j-1}\}\} & \\ &\leq \sum \frac{PD_i \cdot W_j}{W_i} \\ &\leq \frac{B \cdot PD_i}{W_i} \end{aligned}$$

□

Lemma 2. *For each $i = 1, \ ... \ , r+1$, the following holds*

$$\sum_{j=1}^{i} PD_i \geq (1 - \prod_{j=1}^{i}(1 - \frac{W_j}{B})) \cdot (|RP^*|\backslash RP)$$

Proof. We present an inductive proof for Lemma 2. When $i = 1$, Lemma 2 holds. Assuming that Lemma 2 holds for the case $i-1$, we need to show that it holds for the case i.

$$\begin{aligned} \sum_{j=1}^{i} PD_j &= \sum_{j=1}^{i-1} PD_j + PD_i \\ &\geq \sum_{j=1}^{i-1} PD_j + \frac{W_i}{B} \cdot \left(|RP^*\backslash RP| - \sum_{j=1}^{i-1} PD_j \right) \\ &= \sum_{j=1}^{i-1} PD_j \cdot (1 - \frac{W_i}{B}) + \frac{W_i}{B} \cdot (RP^*\backslash RP) \\ &\geq (1 - \prod_{j=1}^{i-1}(1 - \frac{W_{j-1}}{B}) + \frac{W_i}{B}) \cdot (|RP^*|\backslash RP) \\ &\geq (1 - \prod_{j=1}^{i}(1 - \frac{W_j}{B})) \cdot (|RP^*|\backslash RP) \end{aligned}$$

□

Because $\mathcal{W} = \sum_{i=1}^{r+1} W_i = \sum_{i=1}^{r+1} \sum_{O_k \in Q_i} \frac{C_k}{f_k}$, Therefore,

$$\begin{aligned} d \cdot \mathcal{W} &= \sum_{i=1}^{r+1} \sum_{O_k \in Q_i} \frac{C_k}{f_k} \cdot d \\ &\geq \sum_{i=1}^{r+1} \sum_{O_k \in Q_i} C_k \\ &\geq B \end{aligned}$$

We know that, if $a_1, \ ... \ a_m \in R^+$ and $\sum_{i=1}^{m} = \alpha \cdot A$, then the function $(1 - \prod_{j=1}^{m}(1 - \frac{a_j}{A}))$ achieves the minimum value when $a_1 = a_2 = ... = a_m = \frac{\alpha \cdot A}{m}$.

Now we present the proof of Theorem 5.1.

Proof. From Lemma 2, we have:

$$\begin{aligned} \sum_{j=1}^{r+1} PD_j &\geq (1 - \prod_{j=1}^{r+1}(1 - \frac{W_j}{B}))(|RP^*\backslash RP|) \\ &\geq (1 - \prod_{j=1}^{r+1}(1 - \frac{W_j}{\mathcal{W} \cdot d}))(|RP^*\backslash RP|) \\ &\geq (1 - (1 - \frac{1}{d \cdot (r+1)})^{r+1})(|RP^*\backslash RP|) \\ &\geq (1 - e^{-\frac{1}{d}})(|RP^*\backslash RP|) \end{aligned}$$

Because $PD_{r+1} \leq \frac{1}{2}|RP|$, denoting the plan generated by the BestDensity algorithm as $\mathcal{RP}$, we have

$$\begin{aligned} |\mathcal{RP}| &\geq |RP| + |RP'| \\ &= |RP| + \sum_{i=1}^{r+1} PD_i - PD_{r+1} \\ &\geq |RP| + (1 - e^{-\frac{1}{d}})(|RP^*\backslash RP|) - PD_{r+1} \\ &\geq \frac{1}{2}|RP| + (1 - e^{-\frac{1}{d}})(|RP^*\backslash RP|) \\ &\geq (1 - e^{-\frac{1}{d}})(|RP| + |RP^*\backslash RP|) \\ &\geq (1 - e^{-\frac{1}{d}}) \cdot |RP^*| \end{aligned}$$

□

D.3 Proof of Exactly Once Processing

Proof. Firstly, we reason that if all the input streams of a partition p_i are fault tolerant, p_i could achieve exactly-once processing during the incremental recovery. If all the input sources of p_i are fault-tolerant, as the unprocessed inputs are buffered in the upstream, p_i would receive all the unprocessed with a sequence number larger than punctuation P_k, where P_K is the ID of the last completed global checkpoint. Therefore, p_i can correctly eliminate duplicate inputs by comparing the sequence numbers with P_k to achieve exactly-once processing.

If p_i only subscribes input sources, as the source streams are fault-tolerant, following the above discussion, p_i can enforce exactly-once processing. If p_i subscribes output streams of other operators, because p^i's inputs are buffered in its upstream neighbors during incremental recovery, p_i would process every unprocessed input tuple once and only once. In the case that incremental recovery is not adopted, the last completed global checkpoint is used to rollback (restore) the states of all partitions in the topology. Input data are replayed in the source from the progress of the selected global checkpoint, which also guarantees exactly-once processing. In conclusion, our approach guarantees exactly-once processing with the appearance of failures.

□