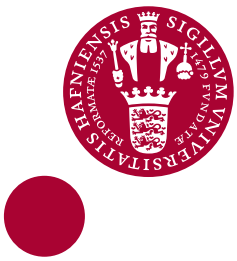


Progressive recovery of correlated failures in distributed stream processing engines

Su, Li; Zhou, Yongluan

Published in:
Advances in Database Technology

DOI:
10.5441/002/edbt.2017.59

Publication date:
2017

Poster Paper

Progressive Recovery of Correlated Failures in Distributed Stream Processing Engines

Li Su
University of Southern Denmark
lsu@imada.sdu.dk

Yongluan Zhou
University of Southern Denmark
zhou@imada.sdu.dk

ABSTRACT

Correlated failures in large-scale clusters have significant effects on systems' availability, especially for streaming data applications that run continuously and require low processing latency. Most state-of-the-art distributed stream processing engines (DSPEs) adopt a blocking recovery paradigm, which, upon correlated failure, would block the progress of recovery until sufficient new resources for recovery are available. As the arrival of new resources is usually progressive, a blocking paradigm fails to minimize the recovery latency. To address this problem, we propose a progressive and query-centric recovery paradigm where the recovery of the failed operators would be carefully scheduled to progressively recover the outputs of queries as early as possible based on the current availability of resources. In this work, we propose and implement a fault-tolerance framework which supports progressive recovery after correlated failures with minimum overhead during the system's normal execution. We also formulate the new problem of recovery scheduling under correlated failures and design effective algorithms to optimize the recovery latency. The proposed methods are implemented on Apache Storm and preliminary experiments are conducted to verify their validity.

1. INTRODUCTION

Fault tolerance is critical to Distributed Stream Processing Engines (DSPEs), such as Apache Storm [14] and Spark Streaming [3], mainly due to the long running time and the low latency requirement of streaming data applications. Previous researches in this area are mainly focused on individual and independent node failures and ignore correlated failures [7, 8], where a number of nodes fail within a short interval. Correlated failures can be caused by failures of shared hardwares, such as switches, routers, and power facilities, or by software problems, such as bad software patches applied across a number of nodes. Although large-scale correlated failures occur less frequently than independent ones, they have significant effects on a system's availability [7].

Correlated failures exhibit characteristics that are very different from independent failures. First of all, correlated failures would incur the unavailability of a large amount of resources. One cannot assume an instant availability of sufficient resources to recover the continuous queries from such failures. Repairing the failed nodes or acquiring additional resources would take a significant amount of time. For example, it may involve solving the software or hardware problems, restarting the failed nodes, and adding them back to the DSPE. Even if the DSPE is running on a cloud environment and virtual resources can be easily allocated to replace the failed nodes, negotiating and acquiring a large amount of new resources would still incur a latency non-negligible for streaming data applications. More importantly, the recovered or newly allocated nodes would probably not become available simultaneously, but rather one after another with noticeable time gaps between them. In other words, the current assumption that all the resources needed for recovery are available at the same time cannot be held.

Most existing DSPEs, such as Flink [1] and Storm [14], adopt a blocking recovery approach in the sense that the recovery of all the parallel operator partitions would be blocked until sufficient new resources are acquired. However, due to the gradual availability of resources in the recovery of correlated failures, such a blocking approach fails to minimize the recovery latency. It is much more desirable to adopt a progressive recovery approach, where the operator partitions can be recovered progressively upon the availability of new resources. Furthermore, the existing systems also adopt an operator-centric paradigm in the scheduling of the recovery, where the operator partitions are scheduled for recovery individually in a topological order. Note that the accurate outputs of a query can only be generated if and only if all the operator partitions of this query are executing normally, this operator-centric paradigm fails to minimize the latency of recovering the producing of query outputs. To address the insufficiency of the existing approaches, we propose a progressive and query-centric recovery paradigm where the recovery of the failed operator partitions would be progressively scheduled to recover the outputs of queries as early as possible based on the current availability of resources. More specifically, if correlated failure happens, we gradually increase the number of recovered queries following the arrival pace of the restarted or the newly acquired nodes. Furthermore, unlike the operator-centric paradigm, our query-centric paradigm attempts to schedule the recovery of the failed partitions to produce the output of a query as soon as possible. This new paradigm would provide not only a shorter recovery latency and earlier query results, but also a more responsive and smoother transition from a failed state to a fully recovered one.

In summary, we propose a fault-tolerance framework that can support progressive recovery during a correlated failure, which imposes minimum overhead during the system's normal execution. We also formulate the new problem of query-centric recovery scheduling under correlated failures, which is an NP-hard problem. To provide a solution for a large-scale job topology, we propose an ef-

ficient and effective approximate algorithm. We implement the recovery framework and the scheduling algorithm on top of Apache Storm, a popular and mature open-source DSPE and conduct multiple sets of experiments on Amazon EC2 to validate the effects of progressive recovery.

2. RELATED WORK

Fault tolerance for DSPEs can be generally categorized into two types [9]: passive approaches and active approaches. Passive techniques include checkpoint [1], upstream buffer [6, 12] and source replay [14, 1, 2]. Active approaches [4, 5, 13, 12] employ hot-standby replicas to achieve faster failure recovery with higher resource consumption. The mainstream DSPEs, such as Samza [2], Flink [1] and Storm [14] adopt source replay and checkpointing techniques. Our checkpointing scheme is similar to the one used in [1]. Both the works in [6, 12] combine checkpointing and upstream buffer to achieve fault tolerance as we do in this work, while missing the optimization for recovery scheduling makes them not suitable for progressively recovering large-scale correlated failures. [13] presents a framework to combine both active and passive techniques to maximize the accuracy of the fast tentative query outputs in correlated failure. Different from [13], which mainly focuses on optimizing resource assignment to improve the quality of tentative outputs, our approach focuses on progressive recovery that minimizes the latency of completely recovering correlated failures, which is orthogonal to the problem studied in [13].

3. PRELIMINARIES

As in most of the mainstream DSPEs, such as Storm [14] and Samza [2], 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. The execution plan of a query consists of multiple operators, each of which contains a user-defined function and can subscribe the output streams of other operators. An operator can be parallelized into multiple operator partitions that have 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 would form the complete input of an operator partition, which is also referred to as partition for simplicity throughout this work.

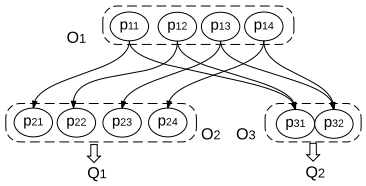


Figure 1: An example topology which consists of two queries Q_1 and Q_2, whose operator sets are $\{O_1, O_2\}$ and $\{O_1, O_3\}$, respectively.

By denoting the operator partitions as vertex and data streams between the operator partitions as directed edges, the execution plan of a query can be abstracted as a directed acyclic graph (DAG). Figure 1 depicts an example DAG. The computation states, input and output buffers for each partition are maintained separately from each other. The output stream of an operator can be shared by the execution plans of multiple queries. Therefore, the DAGs of queries are connected by the shared vertex. We refer to the topology that is composed by all the queries which are concurrently running within the DSPE as the global topology. A user-specified priority, which is denoted as a numerical value (set as 1 by default), is assigned to each query within the topology

4. FAULT TOLERANCE

In this section, we present the fault-tolerance framework that supports progressive recovery and some implementation details.

Checkpointing We use punctuations to trigger checkpointing in partitions and synchronize the progress of checkpoints. Punctuations are generated periodically and inserted into the source streams in a broadcasting fashion. On receiving the punctuations with the same sequence number from all the input streams, a partition starts the process of checkpointing and then broadcasts this punctuation to its downstream neighboring partitions. As the punctuations are not arriving simultaneously, data items arrive after the punctuations must be buffered before the checkpoint is done. Assuming that the last checkpoint of a partition is triggered by punctuation P_{k-1}, tuples from S_i which are received after P_k will be stored in the input buffer. After receiving P_k from all the input streams, the partition generates a checkpoint that stores its computation state and then acknowledges the coordinator. The coordinator tracks the checkpointing progress of the whole topology. Once the coordinator is acknowledged that all the partitions have completed checkpointing for punctuation P_k, it knows that a global synchronized checkpoint of the entire topology for P_k, denoted by $cp(P_k)$, is generated.

Adaptive Buffering. Source buffering is a widely adopted fault-tolerance technique in DSPEs. With source buffering, the system buffers the source data of which the processing state have not been included in the latest global checkpoint. In other words, when a global checkpoint of the entire topology is completely made, we can trim the source buffers by removing those source data whose processing are already reflected in the global checkpoint. It is important to note that the buffers for failure recovery differ from the buffers used in data transfer. The latter can be easily trimmed whenever the data are transferred to the downstream nodes. Due to its simplicity and low overhead, the source buffering approach is widely adopted in most existing operational DSPEs, including Storm [14], Flink [1] and Samza [2]. Upstream buffering is another bufffering technique that requires each partition to buffer its own output until a global checkpoint is made. Due to its high overhead during normal execution, this approach is not used in most mainstream DSPEs.

However, source buffering cannot support progressive recovery, because whenever we need to recover the state of a partition, we have to replay the buffered data from the sources till the current partition. Only recovering a part of the failed partitions makes little sense because the recovery of any remaining one would require to redo the whole recovery again. This means the recovery progress should be blocked until there are sufficient resources to recover all the failed partitions. To solve the above problem, we adopt an approach, called adaptive buffering, which would only incur overhead during failure recovery. With adaptive buffering, we only buffer at the sources during normal execution. Once a burst of multiple node failures is detected within a time window, all the partitions except for the sinks would buffer their outputs to support progressive recovery. These output buffers are turned off when a new global checkpoint is completely created, which indicates the correlated failure is completely recovered.

Figure 2 presents an example of adaptive buffering. Before failure is detected, only the partition in the source operator (i.e., p_1), has output buffer. When partitions p_3, p_4, and p_5 are detected to be failed, at timestamp ts_1, p_3 and p_4 are restarted and the output buffer is turned on in partition p_2 and p_3. At ts_2, after p_5 is restarted, it will first process the output buffer of partition p_3. After all the partitions are recovered, output buffer is turned off in the

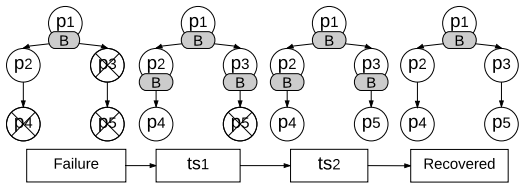


Figure 2: An example of adaptive buffering.

non-source partitions, only the output buffer in p_1 is preserved.

Progressive Recovery. Once failure is detected, assuming that P_k is the punctuation of the latest successful global checkpoint, the failed partitions are restarted and the states of the whole topology are restored or rollbacked with checkpoint $cp(P_k)$. The system would switch to the progressive recovery mode if the total number of failed nodes is higher than a threshold within a specific time window, otherwise it would simply use the blocking recovery method. With adaptive buffering, the output buffers in all the partitions are now turned on and the input data with a greater sequence number than P_k will be replayed from the sources. These output buffers could be used to resume the progress of the failed partitions that are recovered when new recovery resources arrive.

Note that node failures may not occur simultaneously during a correlated failure. In other words, it is possible that additional failures could be detected before the current recovery is completed. With the adaptive upstream buffers, instead of rolling back the states of the whole topology to $cp(P_k)$ again, we only restore the states of the newly failed partitions with $cp(P_k)$ and replay the data buffered in their upstream neighbors. However, as the progress of the newly restored partitions fall behind their downstream neighbors that have been recovered, the downstream may receive duplicated tuples and therefore have to perform duplicate elimination to guarantee exactly-once processing.

However, for a partition p_i with multiple input streams, as the tuples from different upstream partitions may arrive in different orders, p_i may produce outputs in different orders across different replays. To solve this problem, we enforce Order-Preserved processing during recovery to ensure that p_i processes its input in an identical order across different replays. The order-preserved processing is turned on in the beginning of the recovery. The source-buffered data would be divided into mini-batches and each partition attaches a local sequence number that increases monotonically to each of its output tuples. For a partition p_i, tuples within the same batch are stored in its input buffer. When it receives all the data from a batch from all its inputs, p_i starts processing these data from each input stream in a predefined round-robin order. In this way, the order of the output data are guaranteed to be identical across multiple replays. With order-preserved processing, the downstream of p_i can skip duplicate tuples by checking the sequence numbers of tuples from p_i. After the recovery is completed, the order-preserved processing will be turned off together with adaptive buffering.

Implementation. We implement our system on Enorm [11], which is a distributed stream processing system built on Apache Storm [14]. In our system, a special bolt, called control bolt, is automatically generated and appended to the user-submitted job topology. The responsibilities of the control bolt include collecting workload statistics and handling node failure. The fault tolerance coordinator in the control bolt detects node failures by checking their heartbeats in ZooKeeper. Upon a failure is detected, the coordinator calls the optimization algorithm presented in Section 5 to schedule failure recovery following the pace of acquiring new resources. The control bolt is stateless, if failed, it will be restarted by Nimbus in Storm on another node and the interrupted recovery scheduling will be resumed.

5. OPTIMIZING RECOVERY PLAN

In this section, we define the problem of optimizing the recovery scheduling and present an outline of our optimization algorithm. Given a global topology T, we denote the resource consumption of operator O_i in T as C_i, the parallelization degree of O_i as m_i and the resource consumption of p_{ij}, the jth partition of O_i, as c_{ij}. We have $C_i = \sum_{j=1}^{m_i} c_{ij}$. Queries can be assigned with priorities according to their importance and Q_i's priority is denoted by prt_i.

If the amount of available resources is not enough to recover all the failed partitions of a correlated failure, we have to select a subset of the failed partitions for recovery. Whenever a set of new nodes are available, a set of failed partitions will be scheduled for recovery, which is referred to as a partial recovery plan. A failed query is called recovered if and only if all of its failed partitions are recovered. We present a formal definition for the problem of optimizing recovery plan as follows:

RECOVERY PLAN OPTIMIZATION*: For a global topology T, a set of failed queries QS, and the amount of computation resources R available for failure recovery, choose a subset of the failed operator partitions for recovery such that the sum of the priorities of the recovered queries is maximized.*

The RECOVERY PLAN OPTIMIZATION problem is NP-hard, as it can be reduced from the Set Union Knapsack problem, which has been proved to be NP-hard [10]

Considering that operators can be shared by multiple queries, it is natural to prioritize recovering the queries whose operators are shared by more queries. Furthermore, as the failed queries have various recovery costs and priorities, we should consider the profit that can be achieved by using per unit of resource while generating the recovery plan. Taking the above two factors into consideration, we define *Profit Density*, referred to as PD_i, of query Q_i and use it to rank the recovery priorities of the failed queries. PD_i is calculated as follows:

$$PD_i = \frac{prt_i}{\sum^{O_k \in Q_j} \frac{C_k}{f_k}}$$

In the above equation, C_k is the cost of recovering the failed partitions in operator O_k, f_k is the frequency that O_k is shared by the other failed queries. The approximate optimization algorithm starts by calculating the profit density of each failed query. The failed queries are put into a list and sorted in descending order according to their profit density. Next, the list is traversed from the beginning to find the query, Q_i, whose recovery cost is smaller than the amount of currently available resources. The failed partitions belonging to Q_i will be put into the recovery plan. The profit density of the other failed queries will be updated and the list of failed queries are re-sorted. The above loop continues until the resource constraint is reached. The time complexity of this algorithm is $O\left(M^2 \cdot logM\right)$, where M is the number of the failed queries.

6. EVALUATION

All the experiments are conducted on Amazon EC2 using the m3.large instance. We use a real data set consisting of 569,382 tweets crawled from Twitter, which are repeatedly emitted in order into the source operator to emulate a long-standing application.

To explore the time of attaching new nodes to a cluster on a cloud platform, e.g., Amazon EC2, we conduct experiments to record the time interval between when the instance acquiring is started and when the newly attached node is ready to host processing task. We collect in total 180 samples and present their distribution in Fig-

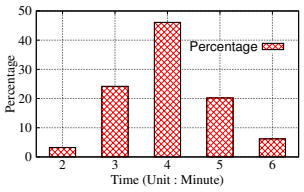


Figure 3: Distribution of time cost of attaching new nodes to a deployed cluster.

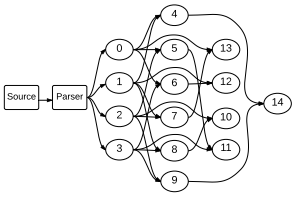
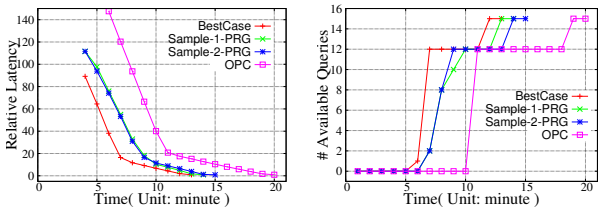


Figure 4: Topology used in the experiment of progressive recovery.

ure 3. One can see that, even on the cloud platform, the newly attached nodes are not arriving simultaneously. The time to attach a new node varies from 2 minutes to 6 minute. This result consolidates our motivation for progressive recovery.



(a) Relative Latency (b) Available Queries

Figure 5: Average relative latency of recovered queries and the number of available queries after correlated failure.

Figure 4 shows the structure of the job topology used in the recovery experiments. There are 15 queries in this topology. The sink operator of query Q_i is denoted as O_i. The parallelization degree is set as 1 for the *Source* and 5 for the other operators. The *Source* operator emits tweets in the rate of 1000 tuples per second. On receiving a tweet, the *Parser* emits a tuple for each hashtag within the tweet. Operator O_1, O_2, O_3, and O_4 conduct sliding-window aggregates, which count the hashtag frequency with various window settings and output the updates of the window instances. Operator O_i, $4 \leq i \leq 14$, maintains the states of the sliding-window aggregates it subscribes.

End-to-end processing latency is a critical performance metric for most streaming data applications. As recovering a large-scale correlated failure would inevitably incur significant increment on processing latency, we propose two metrics that are relevant with processing latency to measure the effectiveness of the compared recovery schemes. Assuming that the latencies of queries before the failure are stable, we propose **Relative Latency** that measures the difference of a query's latency before and after failure. Denoting l_s as a query's latency before failure and l_r as that after failure, its relative latency, RL, is calculated as $\frac{l_r}{l_s}$. Therefore, 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, e.g., $\Theta = 1.2$ in this set of experiments, Q_i is considered as an **Available Query**, which means it has recovered to a normal state. The cluster initially consists of 10 nodes, and we manually kill the 8 nodes where the sink operators of the 15 queries are deployed to inject a correlated failure, and then 8 new nodes are acquired and attached to the cluster to perform recovery.

Figure 5(a) and Figure 5(b) present the relative latency of the recovered queries and the number of available queries using different recovery paradigms. In both figures, *BestCase* denotes the case where all the new nodes become available simultaneously after 3 minutes and the recovery of all the failed partitions are started immediately after that. *Sample-1-PRG* and *Sample-2-PRG* are two different runs using progressive recovery and *OPC* represents the blocking operator-centric recovery.

As one can see in Figure 5, *BestCase* outperforms the others in both the relative recovery latency and the number of available queries, this is because all the failed partitions are recovered only 3 minutes after the failure. On the contrary, *OPC* has the worst recovery performance as its recovery is started after all the new nodes are ready, which results in that *OPC* has more input tuples buffered than the others before the recovery is started. The relative latency of *OPC* is nearly 50% higher than that of *BestCase* at the beginning of the recovery, and it also takes more time for the average relative latency of *OPC* to return to the stable level than *BestCase*. The relative recovery latency and the number of available queries with progressive recovery are between those of BestCase and *OPC*, as the failed partitions are gradually recovered following the pace of resource acquiring. This experiment shows that, compared to the blocking and operator-centric recovery, adopting progressive recovery brings better latency and less time for the failed queries to become available.

7. CONCLUSION

In this work, we present a query-centric progressive recovery framework to improve the efficiency of recovering correlated failure in DSPEs. Following the arriving pace of the newly acquired resources after a correlated failure, failed partitions are scheduled to be progressively recovered such that the outputs of failed queries can be generated as early as possible. We present an effective approximate algorithm to optimize the recovery plan. Experimental results show that, compared to the paradigm of blocking operator-centric recovery, our approach exhibits significant advantages while recovering correlated failures.

8. REFERENCES

[1] http://flink.apache.org/.
[2] http://samza.apache.org/.
[3] http://spark.apache.org/streaming/.
[4] M. Balazinska et al. Fault-tolerance in the borealis distributed stream processing system. SIGMOD '2005.
[5] P. Bellavista et al. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. EDBT'2014.
[6] C. Fernandez et al. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD '2013.
[7] D. Ford et al. Availability in globally distributed storage systems. OSDI'2010.
[8] T. Heath et al. Improving cluster availability using workstation validation. SIGMETRICS '2002.
[9] J.-H. Hwang et al. High-availability algorithms for distributed stream processing. ICDE '2005.
[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg, 2004.
[11] K. G. S. Madsen and Y. Zhou. Dynamic resource management in a massively parallel stream processing engine. CIKM '2015.
[12] A. Martin, A. Brito, and C. Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. DEBS '2014.
[13] L. Su and Y. Zhou. Tolerating correlated failures in massively parallel stream processing engines. ICDE '2016.
[14] A. Toshniwal, S. Taneja, et al. Storm@twitter. SIGMOD '2014.