Fault-Tolerance for Distributed Iterative Dataflows in Action

Chen Xu[†*], Rudi Poepsel Lemaitre[§], Juan Soto[§], Volker Markl[§]
[†]East China Normal University, Shanghai 200062, China
[§]Technische Universität Berlin, Berlin 10587, Germany
[†]cxu@dase.ecnu.edu.cn, [§]firstname.lastname@tu-berlin.de, r.poepsellemaitre@campus.tu-berlin.de

ABSTRACT

Distributed dataflow systems (DDS) are widely employed in graph processing and machine learning (ML), where many of these algorithms are iterative in nature. Typically, DDS achieve fault-tolerance using checkpointing mechanisms or they exploit algorithmic properties to enable fault-tolerance without the need for checkpoints. Recently, for graph processing, we proposed utilizing *unblocking checkpointing*, to parallelize the execution pipeline and checkpoint writing, as well as *confined recovery*, to enable fast recovery upon partial node failures. Furthermore, for ML algorithms implemented using broadcast variables, we proposed utilizing *replica recovery*, to leverage broadcast variable replicas and facilitate failure recovery checkpointing-free. In this demonstration, we showcase these fault-tolerance techniques using Apache Flink. Attendees will be able to: (i) run representative iterative algorithms including PageRank, Connected Components, and K-Means, (ii) explore the internal behavior of DDS under the influence of unblocking checkpointing, and (iii) trigger failures, to observe the effects of confined recovery and replica recovery.

PVLDB Reference Format:
Chen Xu, Rudi Poepsel Lemaitre, Juan Soto, Volker Markl. Fault-Tolerance for Distributed Iterative Dataflows in Action. *PVLDB*, 11 (12): 1990-1993, 2018.
DOI: https://doi.org/10.14778/3229863.3236242

1. INTRODUCTION

In recent years, the growing demand for large-scale data analysis, which includes graph processing and machine learning (ML) algorithms has led to the development of various data processing systems. To simplify the entire analytics workflow, we need general-purpose distributed dataflow systems (DDS) (e.g., Flink [2], MapReduce [3], Spark [6]) that support distributed iterative processing with high-level primitives. Typically, both graph and ML computations are known to incur a long runtime since iteration is expensive. However, over execution periods, some of the nodes may fail (e.g., due to operating system crashes, network interruptions, or problems associated with cloud operations, such as virtual machine shutdowns). Hence, it is indispensable that DDS tolerate such failures and continue the iterative process.

Checkpoint-based strategies are commonly adopted to handle failures. They periodically write the status of a dataset into stable storage as a checkpoint during execution and recover from the last checkpoint upon failure. These strategies must reduce both the checkpoint-writing overhead as well as the recovery costs whenever failures occur. In contrast, checkpoint-free strategies exploit algorithmic properties to achieve failure recovery. Recently, for graph processing, we proposed unblocking checkpointing [4], to parallelize the execution pipeline and checkpointing, as well as confined recovery, to employ preserved outgoing messages from live nodes and enable fast recovery upon partial node failures. In addition, for ML algorithms implemented via broadcast variables, we proposed replica recovery [5] to leverage the implicit replicas of broadcast variable and ease failure recovery checkpointing-free.

In this work, we showcase our fault-tolerance techniques using Apache Flink and address three questions: (1) How does *unblocking checkpointing* reduce the checkpoint-writing overhead efficiently? (2) How does *confined recovery* resume computation from a checkpoint upon failure? (3) How does *replica recovery* continue to compute, despite failures and without the need for checkpoints?

2. BACKGROUND

In this section, we briefly revisit the checkpoint-based and checkpoint-free strategies in our recent studies [4, 5]. For our purposes, we primarily focus on graph and ML algorithms.

2.1 Checkpoint-based Strategies

Checkpoint-based strategies enable failure recovery via the use of checkpoints, i.e., by writing checkpoints during normal execution. However, these strategies incur an overhead. Therefore, we reduce the checkpoint-writing cost using *unblocking checkpointing*. Also, we propose *confined recovery* [4] to achieve a fast recovery from failures. Next, we apply unblocking checkpointing and confined recovery to enable fault-tolerance for graph processing.

Unblocking Checkpointing. In general, each iteration in graph processing reads vertex states as well as edges and performs operations. Thereby, generating new vertex states as input for a subsequent iteration. As edges are usually

*Work done when author was working at TU Berlin.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236242

static data in graph processing, failure recovery will solely depend on the vertex states. Hence, saving vertex states as checkpoints is sufficient for fault-tolerance. One way to save checkpoints is to write them independent of the computation. For example, Spark issues a new job to materialize a checkpoint, which typically incurs a high overhead.

In constrast to managing vertex checkpointing independently as in Spark, we incorporate materialization checkpointing into the normal execution [4]. In particular, we employ tail checkpointing to inject checkpoints into the tail of an execution plan, which parallelizes both checkpoint writing and an iteration computation in an unblocking manner. Note that the vertex state at the beginning of an iteration is exactly the same as the one generated by the previous iteration, since the vertex states are updated iteratively during graph processing. Alternatively, we can employ head checkpointing, which writes checkpoints at the head of each iteration. Here, checkpointing in the $(i+1)^{th}$ iteration saves the vertex state result in the i^{th} iteration as a checkpoint. Hence, unlike tail checkpointing, which parallelizes iterative data generation and checkpointing, head checkpointing enables the parallel execution of both checkpoint writing and the entire data computation at each iteration. Consequently, head checkpointing is preferable over tail checkpointing.

Confined Recovery. With the help of checkpoints, DDS are able to replay the computation and recover from failure. That is, all of the nodes must repeat calculations from the last saved checkpoint, which is time-intensive. However, in the case of partial node failures, live nodes still retain the most recent states. Hence, it is unnecessary for the live nodes to recompute the states. Instead, DDS only need to recompute state on those nodes that have failed.

Based on this idea, we introduced confined recovery [4] for iterative graph processing in a join-groupBy-aggregation pattern on DDS. Confined recovery keeps outgoing messages locally during normal execution, and when failure occurs, it explores messages on live nodes, in order to recover failed nodes from the last checkpoint and avoid asking all of the nodes to do a complete recomputation. However, preserving messages locally for all of the operators incurs an additional overhead, which is high. To maintain the fewest number of messages and reduce the overhead, confined recovery stores outgoing messages only for the groupBy operator. For the join operator, we reconstruct those edges that were lost from input datasets, since the edges in a graph are static and apply a join with those vertices that were lost from the last checkpoint upon failure. For the aggregation operator, we require the input data to be partitioned using the same partition function used with the vertex states and then all of the nodes will aggregate the data locally. Hence, upon failure, the aggregated data will be recomputed without any additional information locally.

2.2 Checkpoint-free Strategies

In contrast to checkpoint-based strategies, checkpoint-free strategies never adopt checkpoints explicitly. To disregard the need for checkpoints, we propose replica recovery [5], which leverages implicit replicas to recover. Next, we apply replica recovery to enable fault-tolerance without any requirements on checkpoints for specific ML algorithms.

Replica Recovery. In our investigations, we examine ML algorithms, such as K-Means whose model parameters (i.e., computational state) fit onto a single machine. Even though the overhead costs of checkpointing model parameters in a single instance is negligible, over time the overhead costs will accumulate and become significant. Incidentally, to improve performance, it is common practice to use broadcast variables to implement model parameters, which provides built-in replicas in all of the nodes. These replicas enable us to adopt a checkpoint-free strategy.

Consequently, we propose replica recovery for ML algorithms, whose model parameters fit onto a single machine. Here, we implement model parameters using broadcast variables. Upon failure, replica recovery explores the replicas on healthy nodes to continue the computations without imposing any additional requirements on checkpoints.

3. DEMONSTRATION

This demonstration implements three fault-tolerance techniques, i.e., *unblocking checkpointing*, *confined recovery* and *replica recovery*, in Apache Flink as well as a user-interface (UI), which is deployed on a laptop. Here, we initialize a JobManager and four TaskManagers in Flink to simulate a distributed deployment. In addition, we employ Hadoop Distributed File System (HDFS) version 2.6.1 to save both the input and output data as well as the checkpoint data. For unblocking checkpointing and confined recovery, we employ the PageRank (*PR*) and Connected Components (*CC*) algorithms and use the *cnr-2000*[1] dataset as input. For replica recovery, we employ the K-Means algorithm and use synthetic data generated with Peel [1], a framework designed to help users define, execute, analyze, and share experiments for distributed systems and algorithms.

3.1 Unblocking Checkpointing

Initially, attendees see a UI for "Unblocking Checkpointing" as depicted in Figure 1. Next, they pick either the *PR* or *CC* algorithm as well as a checkpointing strategy, i.e., *no checkpointing*, *tail checkpointing*, or *head checkpointing*. Then, they are able to explore the execution plans expressed in JSON format, visualize the plans and observe the internal behavior of the system for unblocking checkpointing.

For example, Figure 2 shows a partial execution plan for the PageRank algorithm under varying checkpointing strategies. Figure 2a depicts the case when there is no checkpointing. The Map operator (enclosed in one red box, in the tail end of the execution plan) generates an iterative dataset that is fed back to a Partial Solution operator (enclosed in another red box, near the head of the execution plan). Figure 2b depicts the case when tail checkpointing is employed. A DataSink operator (enclosed in a yellow box, in the tail end of the execution plan) saves checkpoints in an external HDFS directory. Figure 2c depicts the case when head checkpointing is adopted. Similarly, a DataSink operator (enclosed in a yellow box, near the head of the execution plan) saves checkpoints in an external HDFS directory. However, in contrast to Figure 2b, the DataSink operator is located at the head of the execution plan in Figure 2c.

3.2 Confined Recovery

Upon clicking on a drop down menu, attendees will see a UI for "Confined Recovery" as depicted in Figure 3. Under "Cluster Status," they observe a JobManager and four TaskManagers. Next, attendees pick either the *PR* or *CC*

[1]http://law.di.unimi.it/datasets.php

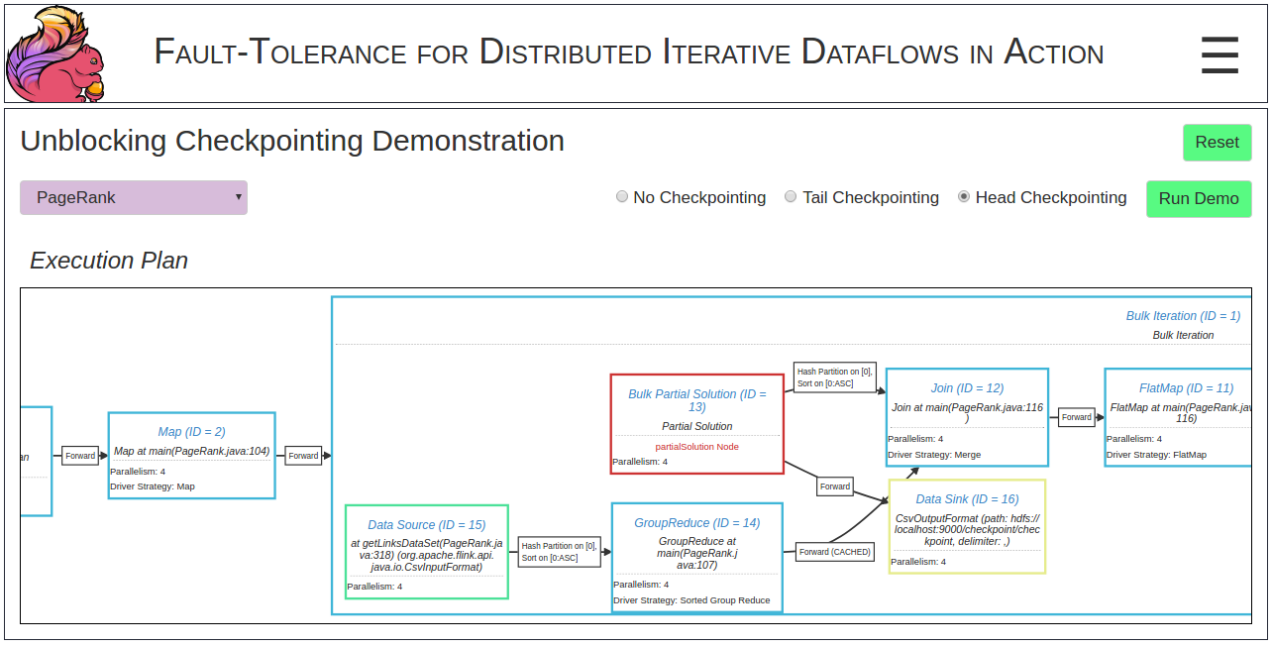


Figure 1: Illustrating Unblocking Checkpointing.

algorithm and experience confined recovery under the single, multiple, or cascading failure scenarios. To create a failure scenario, attendees specify the TaskManagers that should fail. In addition, they specify three parameters, including the: 1) *number of iterations*: the maximum number of iterations for the corresponding algorithm, 2) *checkpoint interval*: the interval between two consecutive checkpoints, and 3) *failed iteration*: the iteration where failure occurs. In particular, we employ the head checkpointing strategy to write checkpoints. Lastly, the job is issued and failure happens, according to the specified parameters. Under the "Job Status," attendees observe whether the execution status is normal, failing, or under recovery.

For example, Figure 3 depicts a single failure scenario, where a TaskManager is selected and highlighted in red. By default, the number of iterations, checkpoint interval, and failed iteration are set to 10, 6, and 8, respectively. This means that the job runs for at most ten iterations, a single checkpoint is saved during the 6^{th} iteration, and a single failure happens during the 8^{th} iteration. By clicking on the "Run Demo" button, confined recovery is illustrated for a single failure. The "Job Status" reflects that the iterative job fails during the 8^{th} iteration and then restarts from the 6^{th} iteration by the live TaskManagers.

3.3 Replica Recovery

Upon clicking on the drop down menu, attendees will see a UI for "Replica Recovery," akin[2] to Figure 3. They can experience replica recovery for the K-Means algorithm under the single, multiple, or cascading failure scenarios. In particular, attendees specify the TaskManagers that should fail to create failure scenarios, as well as two parameters, i.e., the number of iterations and the failed iteration.

[2]Note: Due to space limitations, we do not include a figure for replica recovery. However, it closely resembles Figure 3 with some minor differences.

By default, the failure scenario is a single failure, the number of iterations is 10, and the failed iteration is 8. This means the job runs for at most ten iterations, and a single failure happens during the 8^{th} iteration. Here, no checkpoint is saved. By clicking on the "Run Demo" button, replica recovery for a single failure is illustrated, which reflects that the job failed during the 8^{th} iteration and restarted from the 8^{th} iteration by the live TaskManagers.

4. ACKNOWLEDGMENTS

This work has been supported through grants by the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A).

5. REFERENCES

[1] C. Boden et al. PEEL: A framework for benchmarking distributed systems and algorithms. In *TPCTC*, pages 9–24, 2017.

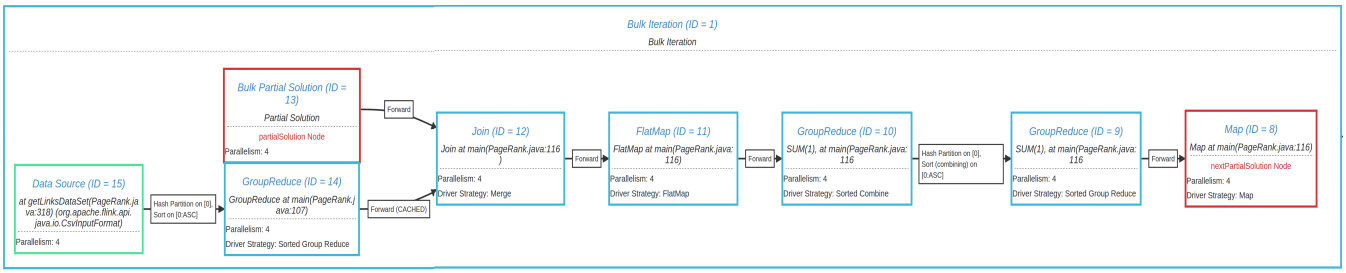
[2] P. Carbone et al. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[3] J. Dean et al. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

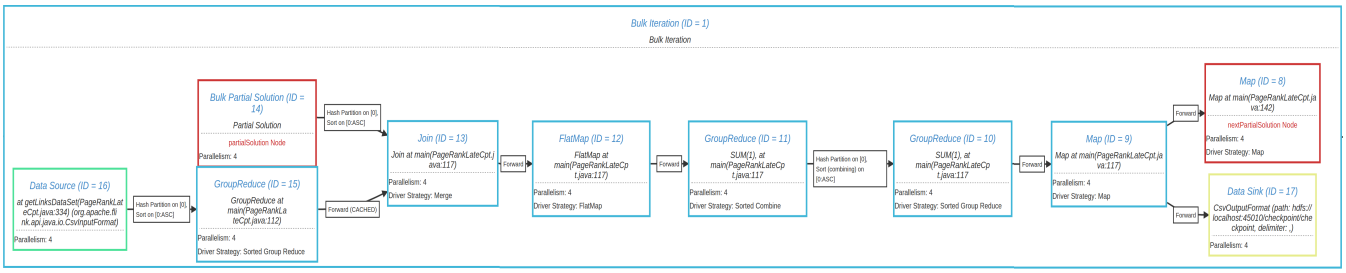
[4] C. Xu et al. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *ICDE*, pages 613–624, 2016.

[5] C. Xu et al. On fault tolerance for distributed iterative dataflow processing. *IEEE Trans. Knowl. Data Eng.*, 29(8):1709–1722, 2017.

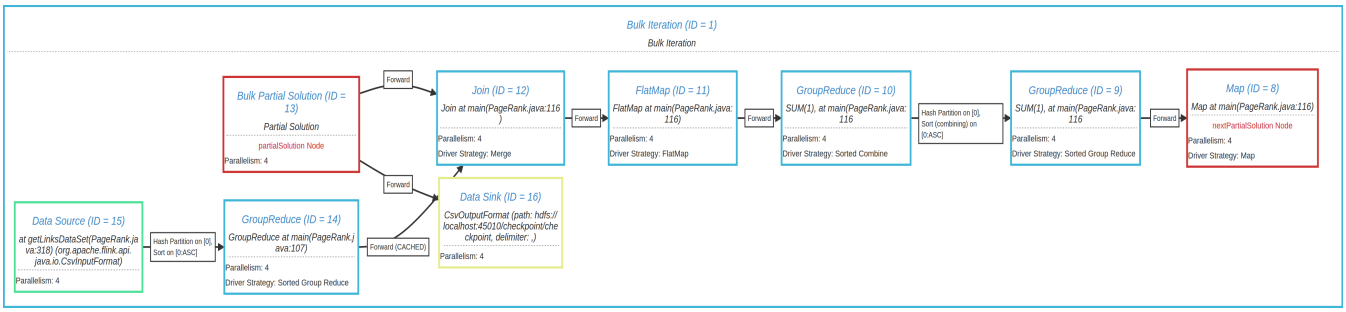
[6] M. Zaharia et al. Spark: Cluster computing with working sets. In *HotCloud*, pages 10:1–6, 2010.



(a) No Checkpointing



(b) Tail Checkpointing



(c) Head Checkpointing

Figure 2: Illustrating an Iteration Execution Plan for PageRank.

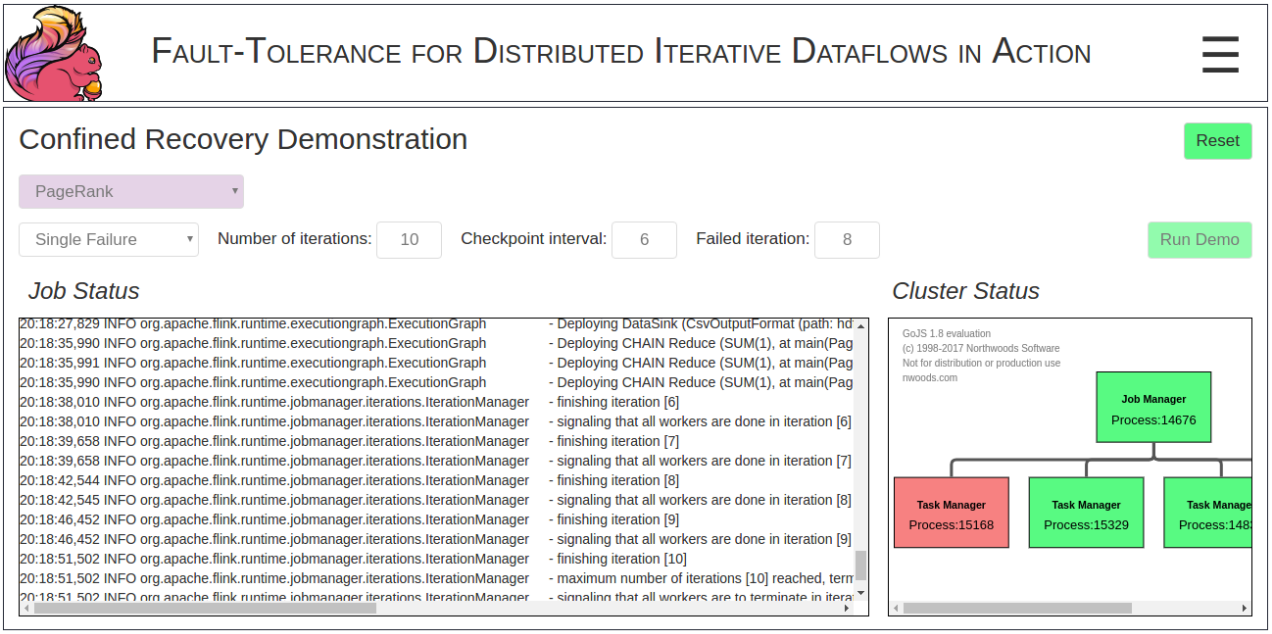


Figure 3: Illustrating Confined Recovery.