Efficient Fault-tolerance for Iterative Graph Processing on Distributed Dataflow Systems

Chen Xu*, Markus Holzemer*, Manohar Kaul†, and Volker Markl*
*Technische Universität Berlin, †IIT Hyderabad
*firstname.lastname@tu-berlin.de, †mkaul@iith.ac.in

***Abstract*—Real-world graph processing applications often require combining the graph data with tabular data. Moreover, graph processing usually is part of a larger analytics workflow consisting of data preparation, analysis and model building, and model application. General-purpose distributed dataflow frameworks execute all steps of such workflows holistically. This holistic view enables these systems to reason about and automatically optimize the processing. Most big graph processing algorithms are iterative and incur a long runtime, as they require multiple passes over the data until convergence. Thus, fault tolerance and quick recovery from any intermittent failure at any step of the workflow are crucial for effective and efficient analysis. In this work, we propose a novel fault-tolerance mechanism for iterative graph processing on distributed data-flow systems with the objective to reduce the checkpointing cost and failure recovery time. Rather than writing checkpoints that block downstream operators, our mechanism writes checkpoints in an unblocking manner, without breaking pipelined tasks. In contrast to the typical unblocking checkpointing approaches (i.e., managing checkpoints independently for immutable datasets), we *inject* the checkpoints of mutable datasets into the iterative dataflow itself. Hence, our mechanism is iteration-aware by design. This simplifies the system architecture and facilitates coordinating the checkpoint creation during iterative graph processing. We achieve speedier recovery, i.e., *confined recovery*, by using the local log files on each node to avoid a complete re-computation from scratch. Our theoretical studies as well as our experimental analysis on Flink give further insight into our fault-tolerance strategies and show that they are more efficient than blocking checkpointing and complete recovery for iterative graph processing on dataflow systems.**

I. Introduction

Graphs as a versatile modeling tool for data analytics are adopted in a multitude of diverse application area. They may represent physical networks (e.g., electrical circuits, roadways, or organic molecules) and less discernible interactions (e.g., ecosystems or sociological relationships). Big data represented as graphs strain the limits of computation of traditional data analytics systems.

To efficiently process big graph data, the information management community has developed a set of novel large-scale distributed graph-parallel frameworks, such as Pregel [1] and GraphLab [2]. These platforms provide a vertex-centric API flexible enough to express arbitrary graph algorithms and graph data that is organized as adjacency lists. These graph-tailored systems enable a wide range of system optimizations due to their restricted focus on graphs.

However, real-world applications often require us to combine graph data with unstructured/tabular data. In many cases, the graph processing algorithm is part of a larger analytics workflow, which includes data preparation, modeling and model application. Such more complex scenarios are cumbersome to handle in the aforementioned graph-parallel systems. For example, to form a web graph it is necessary to first extract the links from crawled web pages. Typically, one uses a scalable dataflow system such as Flink or Spark, which allows for transforming and joining large datasets from multiple sources. Moreover, when confronted with a graph computation problem based on tabular or unstructured data, graph-parallel systems must first transform this representation to adjacency lists before applying the graph algorithm. Dataflow systems, instead, can directly process the tabular data and thus skip the intermediate transformation step entirely. For example, one can compute the transitive closure of a graph directly on the graph's tabular representation using a recursive query [3].

To simplify the entire analytics workflow without installing, managing, and programming against both a graph-tailored system and a dataflow system, general-purpose distributed dataflow frameworks (e.g., MapReduce [4], Spark [5] and Flink [6]) aim to provide high-level graph processing primitives. For graph data analytic tasks, each vertex is associated with a value to represent the current status and the edges link the vertices to illustrate the relationship. Hence, the size of the vertices is proportional to the input dataset size and vertices must therefore be partitioned among the participating machines in order to scale to large datasets [7]. The analysis techniques such as PageRank [8], belief propagation [9] and community detection [10] often require iterative computation with each iteration as a *superstep* [1]. This computation usually incurs a long runtime, since it is executed iteratively until the values of vertices satisfy a convergence or stopping condition. During this prolonged runtime, some nodes in the cluster may undergo halting failures like operating system crashes, network disconnection, or hard-disk crashes. Hence, it is indispensable that the underlying dataflow system tolerates and recovers from such kind of failures in order to continue the iterative computation.

A common approach for handling failures is to periodically write the status of graph vertices into stable storage as a checkpoint during normal execution, and to recover from the latest checkpoint upon failure. Typically, systems write checkpoints in a blocking manner such that each operator produces its complete result before any downstream operator can start consuming the result [11]. For example, the driver programs in Mahout orchestrate the iterative graph computation by manually issuing multiple MapReduce jobs and write the result of each superstep [1], i.e., vertex status, into HDFS as checkpoints, which incurs a high runtime overhead. In an

unblocking manner, Spark writes the checkpoints in the background without requiring program pauses [12]. Stratosphere, which later became Apache Flink, manages the materialization of checkpoints locally by a cache, independently of the pipelined data flow [11]. However, it is oblivious to the iteration control, since checkpointing in the current iteration cannot be assured before starting the next iteration. The iteration-oblivious method is practicable only if all the datasets are immutable. Hence, it is not suitable for checkpointing mutable datasets such as iterative datasets in Flink. Also, this iteration-oblivious method complicates the system design, since an additional component is needed to manage the checkpointing. Upon failure, a straightforward approach for systems to recover is to reload the latest checkpointed state and apply a full re-computation, i.e., complete recovery from that state onwards. HaLoop requires the number of reduce tasks (and the hash function assigning mapper outputs to reducers) to be invariant across iterations [13]. It therefore needs to recompute from the latest checkpointed state, as failures lead may change the number of reduce tasks as well as the hash function. With the help of RDD lineage [12], Spark is able to recover from a failure faster, without a complete re-compuation from the latest checkpointed state. However, not all of the dataflow system maintains the lineage during job execution.

The goal of this work is to provide an efficient fault-tolerance mechanism for iterative graph processing on distributed dataflow systems with the objective to reduce both checkpointing cost and recovery time. Rather than managing the checkpointing independently in an iteration-oblivious way, we incorporate materialization checkpointing in the pipelined execution, making it aware of iterative processing. In particular, we inject checkpointing at the tail of the superstep computation, which parallelizes the checkpoint writing with the production of a vertex dataset at the end of each superstep. Note that at the beginning of superstep $i+1$ the system still holds the vertex state of superstep i, since the vertices are iteratively updated during graph processing. Based on this observation, we propose head checkpointing that writes checkpoints at the head of each superstep to enable parallel execution of checkpoint writing and graph computation during each superstep, which further reduces the checkpointing overhead. This approach not only inherits the advantage of a low execution latency without breaking the pipelined tasks, but simplifies the system design to coordinate the checkpoint writing. Our experiments show that head checkpointing incurs very little overhead in runtime when compared to normal execution (i.e., without checkpointing). Moreover, instead of a simple re-computation completely from the latest checkpointed state, we introduce confined recovery for iterative graph processing in a *join-groupBy-aggregation* pattern on dataflow systems without data transformation lineage. In particular, we employ local logs like Pregel, only during the groupBy stage and adopt the partitioning strategies of the optimizer to avoid logging at the aggregation stage. Considering that the edges in the graph are static, we can reconstruct the lost edges and apply the join with lost vertices upon failure, which avoids a complete recovery. In our experiments, we found that confined recovery reduced the recovery time by 81% in comparison to complete recovery. In other words, confined recovery was nearly 5 times faster than complete recovery. The contributions of this paper are as follows.

TABLE I: Data Schema

Dataset	Schema
Vertex	(v_id: the id of vertex $value$: the solution value of vertex)
Edge	(s_id: the id of the source vertex d_id: the id of the destination vertex $payload$: an optional information of the edge)

- We propose two new checkpointing strategies in unblocking manner, namely *head* and *tail* checkpointing. Our iteration-aware strategies eliminate an additional checkpointing manager for iterative graph processing on dataflow systems.
- We propose *confined recovery* by using local logs during the groupBy stage in iterative graph processing on dataflow systems in order to accelerate failure recovery.
- Our experimental and theoretical studies show head checkpointing and confined recovery outperform blocking checkpointing and complete recovery.

The rest of this paper is organized as follows: Section II introduces the background knowledge about graph computation on dataflow systems. Section III illustrates how to reduce the checkpointing overhead for iterative graph processing during normal execution. Section IV describes how to accelerate recovery for iterative graph processing upon a failure. Section V shows an experimental study on Flink. Section VI highlights related work on fault-tolerance and recovery. Section VII concludes this work.

II. Background

A. Data Model

Without loss of generality, we adopt the schema in Table I to describe the dataset that we use to model the state of a graph processing algorithm. In the `Vertex` dataset, each row identifies a single vertex with its corresponding value. This value represents the solution associated with a vertex. It is set initially and then refined iteratively during the data processing. The input data `Edge` associates a source vertex identifier and a destination vertex identifier as well as an optional payload. The payload is usually the information of the edge. For example, no payload is provided in the *connected components* algorithm.

B. Programming Framework

Generally, for graph processing, a vertex produces messages for other vertices and updates its value based on the messages it receives in each superstep. Employing relational operators, this iterative computation is expressed as:

$$V^{(i+1)} \leftarrow f(V^{(i)} \cup (V^{(i)} \bowtie E)) \tag{1}$$

Here, $V^{(i)}$ is the current value of vertices and E is the edges. First, vertices produce messages for others, i.e., $V^{(i)} \bowtie E$. Then, vertices collect messages as well as the current value, i.e., $V^{(i)} \cup (V^{(i)} \bowtie E)$. Finally, an aggregation function f is applied to produce a new value $V^{(i+1)}$ of vertices for the next superstep.

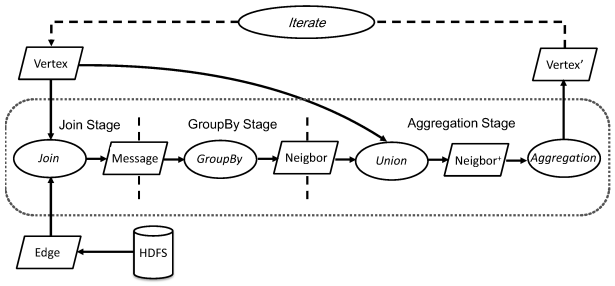


Fig. 1: A Programming Framework for Graph Processing

We can express graph-parallel computation in a distributed dataflow framework as a *join-groupBy-aggregation* pattern, i.e., a sequence of join stages and groupBy stages, similar to the one in GraphX [14]. During the join stage, the `Edge` and `Vertex` datasets join with each other to generate an intermediate dataset, i.e., the messages exchanged between vertices. Here, the join operator is accompanied by a user-defined function to yield the *payload* and/or *value*. For instance, a user-defined join function is specified associated with the join operator in Flink [6], whereas it is achieved by applying a map function following the join operator in Spark [5, 14]. During the groupBy stage, the intermediate dataset is applied by the groupBy operator which groups the data by destination vertex to construct the neighborhood of each vertex. During the aggregation stage, the `Vertex` dataset unions with its neighborhood dataset and then a user-defined aggregation function is applied to compute the new value of the vertex being processed. However, the union operator is optional, since in some applications the value of the vertex solely relies on its neighbors. As an example, a vertex in PageRank computes its own rank by taking into account the rank of its neighbors. Also, union operations are possibly hidden in the join operation. For instance, the current value of the vertex is kept, if the loop of each vertex is added during the initialization of the `Edge` or a variant join such as coGroup [6] is adopted with a user-specified function.

Figure 1 shows a general programming framework for graph processing in dataflow systems. To execute the graph algorithms in dataflow systems, the input data is usually loaded from external storage such as HDFS to construct an `Edge` dataset, whereas the `Vertex` dataset is built by user-specified initial values according to the application. The *iterate operator* indicates that the newly produced vertex set `Vertex'` replaces the previous vertex set `Vertex`, so as to update the value of vertices iteratively. Here, we term `Vertex` and `Vertex'` as the *iterative dataset*.

Generally, the output of each superstep serves as an input to the next superstep during execution of an iterative algorithm. Hence, the `Vertex'` dataset replaces the `Vertex` for the next superstep. This replacement is specified explicitly in an external loop implementation such as Spark, whereas a *built-in back channel* is used for such a kind of replacement implicitly in a native iteration implementation like Flink. As shown in Figure 2, the back channel is a local memory-resident data pool on every node fed by the `Vertex'` dataset and consumed by the `Vertex` dataset.

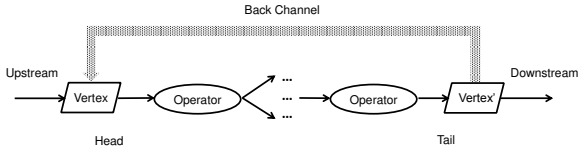


Fig. 2: Native Iterative Dataflow

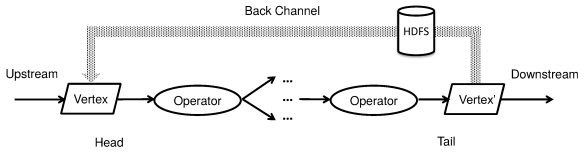


Fig. 3: Writing Checkpoint in Blocking Model

III. Checkpointing

In this section, we first investigate the implementation of blocking checkpointing versus unblocking checkpointing for immutable datasets as well as the cost model. Then, we propose our tail checkpointing and head checkpointing strategies, along with their cost models, for mutable datasets during iterative graph processing on dataflow systems. Both of them incorporate the materialization of checkpoints into the dataflow pipeline execution without an additional manager to coordinate the iteration with the checkpointing. In particular, tail checkpointing incorporates the materialization of checkpoints into pipelined execution by writing a checkpoint at the end of the superstep computation, whereas head checkpointing achieves this by writing a checkpoint at the beginning of the superstep computation. Hence, tail checkpointing parallelizes the checkpoint writing with the production of a vertex dataset at the end of each superstep, whereas head checkpointing parallelizes checkpoint writing with the graph computation of each superstep. Finally, we compare the overhead of three checkpointing strategies based on our proposed cost models and discuss the differences between them.

A. Blocking vs. Unblocking Checkpointing

In a *blocking operator model* [11], each operator produces its complete result before any downstream operator can start consuming the result. This model simplifies the implementation of the checkpointing strategy and is widely adopted by a group of systems including Dryad [15], Mahout and Pregelix [16] to write checkpoints. Following this principle, to write checkpoints, the iterative dataset is saved before starting the next superstep computation. In an iterative dataflow, the checkpoints can be written via the back channel as illustrated in Figure 3. For simplicity, we assume that all the nodes in this cluster work homogeneously and that the workload is perfectly balanced across all the nodes. Then, the overhead O_b of blocking checkpointing is evaluated as follows:

$$O_b = \frac{D^{(i)}}{n\nu} \tag{2}$$

where $D^{(i)}$ is the data size of the checkpoint after the end of superstep i but before the start of superstep $i+1$, n is the number of nodes in the cluster and ν is the upper bound of

the writing rate that each node can realize for a chosen stable external storage system.

The blocking operator model greatly simplifies the task of fault-tolerance as it prevents situations whereby a downstream task consumes a portion of its predecessors output, whereas the remainder becomes unavailable due to a failure [11]. However, this blocking model often increases the execution latency as shown in Example 1. The reason for such a high latency is that checkpoints are written only when the iterative dataset is completely available and the following superstep starts the computation after finishing the checkpointing.

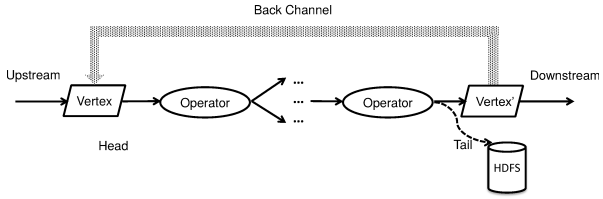
Example 1. *Assume that a graph processing algorithm is running on a cluster consisting of* 10 *nodes. Additionally, the data volume of the iterative dataset* `Vertex'`*, i.e., the checkpoint size, is* 10 *GB and the achieved rate of writing to HDFS per node is* 50 *MB/sec. Then, the additional overhead of writing checkpoints in a blocking operator model is* 20.48 *sec according to Equation 2. Given that the execution time of a superstep without any checkpoint is* 2 *min, then the percentage time spent checkpointing vs. the total execution time is* 14.6% *(*$\approx \frac{20.48\ sec}{2\ min\ +20.48\ sec}$*) approximately.*

Differently, Spark writes the checkpoints in the background without requiring program pauses [12]. It separates the checkpointing from the actual dataflow execution. This seperation is oblivious to the iteration control, since checkpointing in the current iteration cannot be assured before starting the next iteration. The *iteration-oblivious* method is practicable in Spark since all the datasets, i.e., RDDs, are immutable. However, *iteration-oblivious* method is not suitable for checkpointing mutable datasets like *iterative datasets* (e.g., `Vertex` datasets in Figure 1) in Flink as iterative datasets are updated during each iteration. Similarly, Stratosphere [11] proposes to locally materialize checkpoints "on the side" without breaking the pipeline in an *unblocking* manner. It is still oblivious to iteration control and complicates the system design that an additional component is required to coordinate the checkpointing for fault recovery, especially for iterative graph algorithms. Moreover, a disk failure on the node is a disaster for a local materialization strategy since the data on the failed disk would be totally lost and a backward re-computation might be time-consuming.

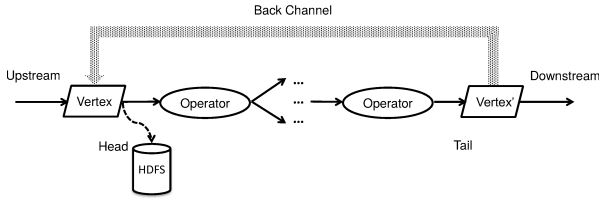
B. Tail vs. Head Checkpointing

Instead of such an iteration-oblivious approach, we adopt an *unblocking checkpointing* by materializing checkpoints into external storage within the pipelined dataflow execution, making it aware of iterative processing. In particular, writing checkpoints to save the mutable iterative dataset into external storage is a special task incorporated implicitly into the pipelined execution. It not only inherits the advantage of a low execution latency without breaking the pipelined tasks, but also enables the iteration coordinator to initiate the next iteration only after checkpointing in the current iteration is completed. Furthermore, external storage like HDFS provides a high availability and reliability for fault-tolerance.

For tail checkpointing, as illustrated in Figure 4a, the checkpoint is written into stable storage along with the production of the `Vertex'` dataset. We employ the pipelined



(a) Tail Checkpointing



(b) Head Checkpointing

Fig. 4: Writing Checkpoint in Unblocking Model

streaming rate ν_t at the tail of the superstep, as `Vertex'` is generated in a pipelined fashion. If the streaming rate to produce `Vertex'` is greater than the highest rate ν to write into the external storage, i.e., $\nu \geq \nu_t$, then the checkpoint can be written without any runtime overhead. Otherwise, the data is accumulated and the process waits to write into external storage. `Vertex'` is completely available in $\frac{D^{(i)}}{n\nu_t}$. During this time, the volume of data which has been written is $\frac{D^{(i)}\nu}{n\nu_t}$. Hence, the volume of remaining data pending for writing is $\frac{1}{n}[D^{(i)} - \frac{D^{(i)}\nu}{\nu_t}]$. The time required to write this remaining data to disk is the additional overhead. Given the writing rate ν, the overhead O^t_{ub} of tail checkpointing is evaluated as follows:

$$O^t_{ub} = \begin{cases} \frac{D^{(i)}(\nu_t - \nu)}{n\nu\nu_t} & \nu < \nu_t \\ 0 & \nu \geq \nu_t \end{cases} \tag{3}$$

Example 2 shows that tail checkpointing parallelizes the checkpoint writing with the production of a vertex dataset at the end of each superstep, so as to reduce the runtime overhead of checkpointing.

Example 2. *Following Example 1, if the rate of pipelined streaming to generate the* `Vertex'` *dataset is* 60 *MB/sec, then the overhead of tail checkpointing is* 3.41 *sec according to Equation 3.*

For head checkpointing, as shown in Figure 4b, the checkpoint is written into stable storage along with the consummation of the `Vertex` dataset. As the `Vertex` dataset is consumed by downstream tasks in a pipeline manner, the checkpoint can be written without any runtime overhead if the writing rate ν into the external storage is greater than the pipelined streaming rate ν_h at the head of the superstep, i.e., $\nu \geq \nu_h$. If $\nu < \nu_h$, there is remaining data after the whole `Vertex` dataset is consumed and the volume is $\frac{1}{n}[D^{(i)} - \frac{D^{(i)}\nu}{\nu_h}]$. The time to write this data is $\frac{D^{(i)}(\nu_h - \nu)}{n\nu\nu_h}$. However, there is still no runtime overhead incurred if this time is less than the normal execution time t_i at superstep i, i.e., $\frac{D^{(i)}(\nu_h - \nu)}{n\nu\nu_h} > t_i$, since writing data into external storage and the downstream processing are done in parallel. Otherwise, the iteration coordinator needs to wait for the writing into

TABLE II: Parameters

Symbol	Parameter
O_b	the overhead of blocking checkpoint
O^t_{ub}	the overhead of tail checkpointing
O^h_{ub}	the overhead of head checkpointing
$D^{(i)}$	data size of the checkpoint at superstep i
n	the number of nodes in the cluster
ν	writing rate to a external storage on each node
ν_t	the rate to generate dataset at the tail of the superstep
ν_h	the rate to consume dataset at the head of the superstep
t_i	the execution time of superstep i without any checkpoint
δ_i	the interference of head checkpoint to runtime at superstep i

external storage to finish, in order to proceed with the next superstep. In such case, the time difference incurs the runtime overhead. However, we need to consider the interference δ_i ($\delta_i < t_i$) of head checkpoint which might delay the runtime of the job, since it still occupies the computation or storage resource. Formally, the overhead O^h_{ub} of head checkpointing is approximated by:

$$O^h_{ub} = \begin{cases} \frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} - t_i + \delta_i & \frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} > t_i - \delta_i \\ 0 & \text{Otherwise} \end{cases} \quad (4)$$

Example 3 shows that head checkpointing parallelizes checkpoint writing with the graph computation of each superstep, so that to reduce the runtime overhead of checkpointing significantly.

Example 3. *Following Example 1, if the rate of pipelined streaming to consume the* `Vertex` *dataset is* 60 *MB/sec, then the overhead of tail checkpointing is* 0 *according to Equation 4.*

C. Comparison

Table II lists the parameters adopted by the cost models in previous sections. Following these cost models, we can conduct the theorems below.

Theorem 1. *The overhead of the unblocking checkpointing is not higher than the blocking checkpointing.*

Proof: We study tail checkpointing vs. blocking checkpointing and head checkpointing vs. blocking checkpointing, respectively.

- Tail checkpointing vs. blocking checkpointing: If $\nu < \nu_t$, then $\frac{\nu_t-\nu}{\nu_t} < 1$, so that $O^t_{ub} = \frac{D^{(i)}(\nu_t-\nu)}{n\nu\nu_t} < \frac{D^{(i)}}{n\nu} = O_b$. Otherwise, $O^t_{ub} = 0$ and $O_b > 0$. Hence, $O^t_{ub} < O_b$. Consider the case $\nu \ll \nu_t$, there is $O^t_{ub} \approx O_b$.
- Head checkpointing vs. blocking checkpointing: If $\frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} > t_i - \delta_i$, then $O^h_{ub} = \frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} - t_i + \delta_i < \frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} < \frac{D^{(i)}}{\nu} = O_b$. Otherwise, $O^h_{ub} = 0$ and $O_b > 0$. Hence, $O^h_{ub} < O_b$.

In summary, we have both $O^t_{ub} \lessapprox O_b$ and $O^h_{ub} < O_b$, which means the overhead of the blocking model is higher than the unblocking model. ■

Theorem 2. *The overhead of head checkpointing is not higher than tail checkpointing, if the superstep execution is simplified as a pipeline at a constant rate, i.e., $\nu_t = \nu_h$.*

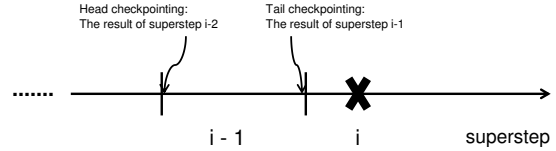


Fig. 5: Difference between Tail and Head Checkpointing

Proof: If $\nu < \nu_h$ and $\frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} > t_i$, then $O^h_{ub} = \frac{D^{(i)}(\nu_h-\nu)}{n\nu\nu_h} - t_i + \delta_i < \frac{D^{(i)}(\nu_t-\nu)}{n\nu\nu_t} = O^t_{ub}$. Otherwise, $O^h_{ub} = 0$ and $O^t_{ub} \geq 0$. Hence, we have $O^h_{ub} \leq O^t_{ub}$. ■

D. Discussion

Theorem 1 shows that the pipelined dataflow system should adopt the unblocking model to save the iterative dataset for graph processing. Theorem 2 indicates that checkpointing the iterative dataset for graph processing at the head of the superstep might incur a low overhead. In some cases such as Examples 2 and 3, head checkpointing has no overhead whereas tail checkpointing incurs a significant cost.

A straightforward question here is the choice of the *checkpointing interval*. An optimal checkpointing interval [17] is derived as $T_{interval} = \sqrt{2T_{MTBF}O_c}$ where O_c is the overhead it incurs to complete the checkpointing and T_{MTBF} is the mean time between failures for the cluster. However, this result is based on the assumption of a Poisson distribution of failures and a long duration of monitoring is required to derive T_{MTBF}. Additionally, our lightweight unblocking checkpointing, especially head checkpointing makes it possible to frequently write checkpoints. We will revisit the checkpointing interval with a special focus on the recovery phase in Section IV.

It deserves to be mentioned that there is a difference in the time of successful checkpointing between head and tail checkpointing. That is, head checkpointing delays writing the result of the current superstep to the next superstep. In particular, after superstep i finishes, tail checkpointing ensures that the result of superstep i is stored whereas head checkpointing ensures that the result of superstep $i-1$ is recorded. As depicted in Figure 5, a checkpoint is set in every superstep and then there is a failure encountered at superstep i'. The system returns to reload the result of superstep $i'-1$ if tail checkpointing is adopted. However, the system can only fetch the result of superstep $i'-2$ if head checkpointing is employed. This difference further motivates our study into the speedy recovery of the iterative dataset value from the latest checkpoint prior to the failure, during graph processing.

IV. Recovery

In this section, we first illustrate the failure recovery problem for iterative graph processing on dataflow systems. Then, we introduce our confined recovery approach for iterative graph processing in a *join-groupBy-aggregation* pattern on dataflow systems without data transformation lineage. In particular, we employ local logs like Pregel only during the groupBy stage and adopt the partition strategies provided by the optimizer to avoid logging at aggregation stage. Consider

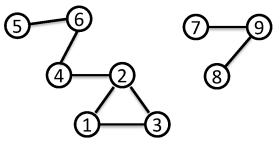


Fig. 6: Graph

that the edges in the graph processing are static, then we reconstruct the lost edges and apply the join with lost vertices upon failure, which avoids a complete recovery. Finally, we analyze the recovery time of our proposed confined recovery approach and discuss how to utilize the remaining memory for local logging together with the checkpoint interval aforementioned in order to reduce the overhead of our logging.

A. Failure Recovery

During the normal execution of the iterative algorithm, data is hosted on a group of compute nodes and some of these nodes can encounter failures. For $j \in [0, n-1]$, let V_j denote the partition of `Vertex` on node N_j.

Definition 1. *(State). The state S is a function recording the latest superstep that has been completed by partition V_j at a certain time.*

If there is no failure, $\forall j \in [0, n-1]$, $S(V_j)$ begins from 0 to i_{max} synchronously. However, data nodes might fail during normal execution so that different partitions complete different supersteps. Hence, different partitions can end up with different states.

Let $F(N_{\mathbb{F}}, i', m)$ denote a failure that occurs on a set $N_{\mathbb{F}}$ ($\mathbb{F} \subseteq [0, n-1]$) of compute nodes when the job performs normal execution in superstep $i' \in ([1, i_{max}])$. Here, m means the latest checkpoint is the result of superstep m. After F is detected, all the partitions residing in the failed nodes are lost and their latest statuses are stored in the latest checkpoint or some other nodes. Hence, we have:

$$S(V_j) = \begin{cases} m & j \in \mathbb{F} \\ i'-1 & j \notin \mathbb{F} \end{cases} \tag{5}$$

In general, the recovery for F is to reset the state of all of the partitions to be $i'-1$, i.e.,

$$S(V_j) = i'-1, \forall j \in [0, n-1] \tag{6}$$

As shown above, the state of the partitions of `Vertex` on healthy nodes is $i'-1$. Hence, the failure recovery task reduces to transforming the state of V_j ($j \in \mathbb{F}$) from m to $i'-1$.

Example 4. *Assume there is a failure $F(N_{\{2\}}, 4, 1)$ when processing the graph in Figure 6. It indicates node N_2 fails when the graph computation is in the 4th superstep, and the latest checkpoint saves the result of the 1st superstep. Therefore, $S(V_0) = S(V_1) = 3$ whereas $S(V_2) = 1$. After the failure recovery, $S(V_2)$ should be 3.*

B. Parallel Confined Recovery

An intuitive failure recovery approach is to recompute since the latest checkpoint. For instance, reloading the checkpoint and applying a re-computation since the 2nd superstep is able to achieve $S(V_2) = 3$ in Example 4. Clearly, this recomputation time is approximately equal to the running time elapsed since the previous checkpoint. However, it is better to avoid a *complete* recomputation, as one or two nodes crashed lead to just a partial loss. Hence, we aim to employ a *confined* reconstruction in order to speed up the process of failure recovery.

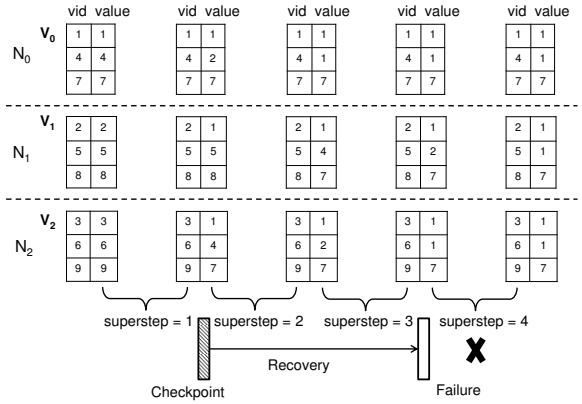


Fig. 7: Vertex in Each Superstep of Connected Components

In order to seek a fast recovery strategy, we first review the dependency between the parent and child datasets which are divided with operators such as *join*, *groupBy*, *union* and *aggregation* in Figure 1. Adopting the dependency between Resilient Distributed Datasets (RDD) in [12], the relationship between the parent and child datasets is classified into two types: *narrow dependency* and *wide dependency*. In a narrow dependency, each partition of the parent dataset is used by at most one partition of the child dataset, whereas each partition of the parent dataset might depend on multiple child partitions in a wide dependency. Thus, the join operator might lead to a wide dependency, as `Vertex` and `Edge` are partitioned independently like GraphX [14]. The groupBy operator leads to a wide dependency as it usually repartitions the `Message` dataset according to the destination id. The union operator leads to a narrow dependency as it merges `Neighbor` and `Vertex`. The type of dependency that aggregation leads to depends on whether `Neighbor` and `Vertex` are co-partitioned. If these two datasets are co-partitioned, aggregation leads to a narrow dependency as aggregation functions are applied locally. Otherwise, it leads to a wide dependency.

As illustrated by Zaharia et al. [5], recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, which is similar to *confined recovery* in Pregel [1], because they only recompute the lost partitions on the failed nodes. With wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of the dataset. Such a failure requires a *complete recovery*, i.e., a re-execution on all the partitions of the ancestor datasets. In our framework of graph processing in Figure 1, a failure might bring the loss of some partitions of `Vertex`, i.e., $V_j, j \in \mathbb{F}$, as well as the `Neighbor` and `Message` datasets, since the join and groupBy lead to wide dependencies between them. Therefore, a complete re-execution from the latest checkpoint is necessary once a failure happens. Clearly, this complete recovery is inefficient,

especially for large graph processing with a relatively long execution time in each superstep.

To avoid an inefficient complete recovery, we investigate the properties of a graph processing framework in dataflow systems to break down the wide dependency so as to apply a confined recovery like narrow dependency. In particular, we describe the recovery strategies according to the reverted direction of *join-groupBy-aggregation* pattern.

- *Aggregation*: The $\texttt{Neighbor}^+$ is *well-partitioned* after the union operation, if `Neighbor` and `Vertex` are co-partitioned. In such a case, aggregation on $\texttt{Neighbor}^+$ does not need a shuffle phase which leads to a narrow dependency. In order to achieve that, the same partition function with `Vertex` is applied to `Message` by the groupBy operator. This optimization is usually achieved by the optimizer.
- *GroupBy*: The lost partition of `Neighbor` depends on the partition of `Message` on the same failed node and the data received from the other partitions on healthy nodes during the shuffle phase. Like the local log in Pregel, if all of the nodes locally store the data routing information within the groupBy shuffle in normal execution, then a confined recovery during the groupBy stage is possible as the data from other nodes is able to be collected without a complete recomputation. Formally, let M_j ($j \in [0, n-1]$) denote the partition of `Message` on node N_j, and $M_{k\to j}$ denote the data items which are sent from node N_k to N_j. For any $k \in [0, n-1]$ and $k \neq j$, $M_{k\to j}$ is required to log on node N_k. This log on N_k is denoted as L_k. Example 5 illustrates the logged adopted in the connected component algorithm.
- *Join*: The runtime optimizer [18] decides whether to co-partition `Edge` and `Vertex` in systems like Flink. If co-partition is adopted, `Edge`, `Vertex` and `Message` become narrow dependent. Otherwise, the join operator leads to a wide dependency `Edge`. However, lost partitions of `Vertex` and `Edge` are able to be reconstructed from checkpoint and data source upon a failure respectively. Hence, the lost partitions of `Message` are recomputed without a complete re-execution to join `Vertex` with `Edge`.

Example 5. *Figure 8 shows the details of the data processing for the second superstep in Figure 7. We assume* `Vertex` *is hash partitioned on three data nodes and* `Vertex` *and* `Edge` *are not co-partitioned without loss of generality. Also, join between* `Vertex` *and* `Edge` *is applied at the* `Vertex` *side for simplicity. After the join operation, the* `Message` *dataset is generated. Then, the outgoing messages are logged locally on each node as depicted in Figure 8. For example, data items* $(3,1),(6,2),(9,7)$ *are sent from node* N_0 *to node* N_2 *and node* N_0 *logs them. Similarly, node* N_1 *also logs* $(3,1),(6,5),(9,8)$ *which are sent to node* N_2.

For all the $j \in \mathbb{F}$, the failure recovery is to transfer $S(V_j)$, i.e., the state of V_j, from m to $i'-1$. In the following, we first discuss single node failure as it is a basic unit for parallel confined recovery. Then, both multiple-node and cascading failures reduce to a variant of single node failures.

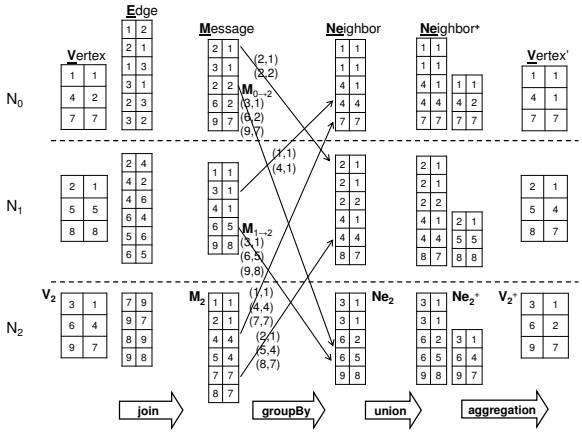


Fig. 8: Data Processing of the 2nd Superstep

Algorithm 1: Confined Recovery for Single Failure

```
1 while S(V_j) < i' − 1 do
2 |   reconstruct V_j and E;
3 |   M_j ← V_j ⋈ E;
4 |   Ne_j ← M_j + L_k(k ≠ j) ;
5 |   Ne_j^+ ← Ne_j ∪ V_j;
6 |   V_j' ← f(Ne_j^+);
7 |   V_j ← V_j';
8 end
```

Single Failure: Instead of utilizing a new available node to substitute the failed node in [1, 19], the healthy nodes are responsible for confined recovery in our approach once a single node failure is detected. Hence, the confined recovery process is applied in parallel on all the remaining healthy nodes. The logical presentation of the recovery process is shown in Algorithm 1. The lost partition of `Vertex` (V_j) and `Edge` (E) are reconstructed from the checkpoint or from external storages (Line 2). Then, V_j joins with E in parallel to rebuild M_j, the lost partition of `Message` on node N_j (Line 3). Making use of $L_k(k \neq j)$, the local log on the healthy nodes, the lost partition of `Neighbor`, i.e., Ne_j, is acquired (Line 4). Lastly, the lost partition V_j' of `Vertex'` is recovered after the aggregation stage (Line 5-6) and V_j is replaced by V_j' which transfers $S(V_j)$ from m to $m+1$ (Line 7). This process is then repeated until $S(V_j)$ reaches $i'-1$ (Line 1). In particular, this process is executed in parallel as illustrated in Example 6.

Example 6. *Consider the same failure* $F(N_{\{2\}}, 4, 1)$ *in Example 4,* $S(V_0) = S(V_1) = 3$ *and* $S(V_2) = 1$. *This failure is recovered since the 2nd superstep and Figure 9 shows the process of parallel confined recovery. Firstly,* V_2 *and* E *are rebuilt. They are assigned to the healthy nodes. Then,* M_2 *is acquired by joining* V_2 *with* E. *Using the logs* L_0 *and* L_1 *on node* N_0 *and* N_1, Ne_2 *is reconstructed and* Ne_2^+ *is the union of* Ne_2 *and* V_2. *Consequently, after applying an aggregation on* Ne_2^+, V_2' *is recovered. By replacing* V_2 *with* V_2', $S(V_2) = 2$. *Similarly,* $S(V_2)$ *would be 3 after a confined recovery for the 3rd superstep.*

Multiple Failures: Multiple node failures reduce to a single node failure if these failed nodes are considered as one node

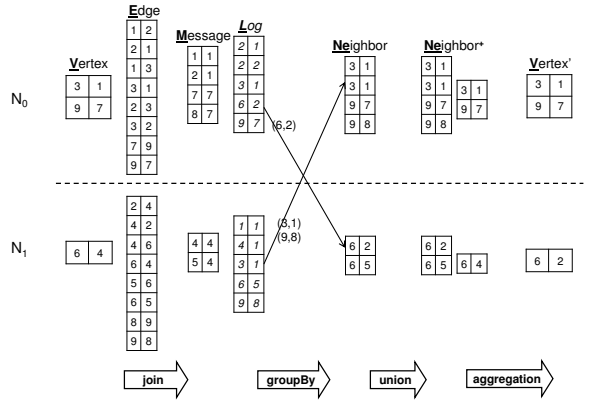


Fig. 9: Parallel Confined Recovery for the 2nd Superstep

logically. Hence, to recover from the multiple node failures, Algorithm 1 is applied similarly where V_j ($j \in \mathbb{F}$) are the lost partitions on the set of nodes $N_{\mathbb{F}}$.

Cascading Failures: Different from multiple node failures where nodes fail at the same time, cascading failures mean new nodes fail when the system is undergoing a recovery. However, cascading failures can be considered as consecutive single node failures. Hence, Algorithm 1 is called consecutively to recover from the cascading failures.

C. Recovery Analysis

We use t_i to denote the execution time of superstep i. For $F(N_{\mathbb{F}}, i', m)$, the complete recovery totally re-computes from the checkpoint at superstep m to the superstep $i'-1$. If the failed nodes are substituted by the new nodes, the time of complete recovery is approximated by

$$\tau_{com} = \eta + \Sigma_{i=m+1}^{i'-1} t_i \tag{7}$$

where η is the constant time of failure detection and data reloading, etc. However, if the failed nodes cannot be substituted immediately, the recomputation is applied on the live nodes and the parallelism is reduced compared to the normal execution. We assume the data assigned on each node is nearly even and adopt $|\mathbb{F}|$ to describe the number of failed nodes, i.e., $N_{\mathbb{F}}$. Then, the time of complete recovery is approximated by:

$$\tau_{com} = \eta + \frac{n}{n - |\mathbb{F}| \bmod n} \Sigma_{i=m+1}^{i'-1} t_i \tag{8}$$

In particular, Equation 7 is considered as a special case $|\mathbb{F}| = n$ in Equation 8.

Different from complete recovery, the confined recovery only re-computes lost partitions on failed nodes. Hence, $\frac{|\mathbb{F}|}{n}$ of the data in normal execution is processed in confined recovery. Consider that this recomputation is executed on $n - |\mathbb{F}|$ live nodes in parallel, then the time of parallel confined recovery is given as follows:

$$\tau_{con} = \eta + \frac{|\mathbb{F}|}{n - |\mathbb{F}| \bmod n} \Sigma_{i=m+1}^{i'-1} t_i \tag{9}$$

For single failure, i.e., $|\mathbb{F}| = 1$, the comparison between Equation 8 with Equation 9 shows that, the time of parallel confined recovery is much less than the time of complete recovery, i.e., $\tau_{con} < \tau_{com}$. For multiple failures, i.e., $|\mathbb{F}| > 1$, the time of confined recovery is less than complete recovery. But as the increasing of failed nodes, τ_{con} is increased until τ_{com}. Accordingly, the recovery time of cascading failures can be derived as more failed supersteps in Equation 8 and 9.

D. Discussion

The analysis above shows that confined recovery outperforms complete recovery during the recovery phase. However, confined recovery relies on local logging during normal execution which incurs additional overhead, since the logs are written to disk. The ideal case of very negligible overhead arises only when the logs can completely fit in main memory.

But the remaining available memory, except the one for normal execution, is usually limited. The accumulation of buffered logs during the computation of supersteps leads to either swapping with disk or preempting the memory for normal execution, which has a negative impact on execution runtime. Here, we employ $\tilde{T}(M)$ to denote the execution time with logging in memory where M is the size of available memory. The logging would not incur significant negative impact if

$$\tilde{T}(M) - T < \varepsilon \tag{10}$$

where T is the execution time without any logging and ε is a threshold. Hence, it is necessary to release the memory after time T. Here, a checkpoint is indispensable to ensure the fault tolerance. In other words, this time T is fit for the checkpoint interval $T_{interval}$ mentioned in Section III-D.

However, it is time consuming to run the workloads repeatedly to identify T. A heuristic strategy is to track the utilization of memory and write the head checkpoint with light overhead when the buffered logs lead to a high utilization of assigned memory. In general, head checkpointing together with confined recovery reduces the overhead at runtime and accelerates the recovery phase upon failure.

V. Experimental Analysis

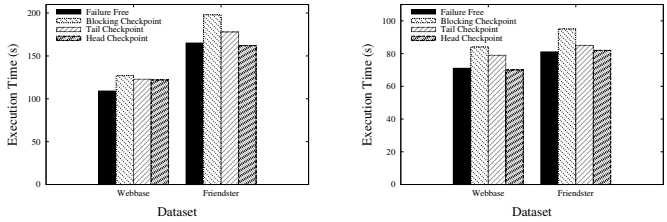
We integrate our proposed head and tail checkpointing strategies as well as confined recovery approaches in Apache Flink [6]. Also, we implement the blocking checkpointing and complete recovery in Flink as baselines. They are widely adopted by other systems such as Mahout, Pregelix and HaLoop. This section shows the results of our experimental study on checkpointing and recovery.

A. Experiment Setting

1) Cluster Setup: The experimental study was conducted on our university cluster. The cluster consists of 24 compute nodes, each of which is equipped with 8 Intel Xeon E5620 CPUs, 32GB of memory, and four 800GB SATA hard disks. On each compute node, we installed Ubuntu 14.04 operating system and Java 1.7. Our customized version of Flink was deployed on this cluster where one node acted as the master running Flink's JobManager while the other nodes acted as slaves with a TaskManager. In order to use HDFS, we deployed Hadoop 2.4.1 on our cluster. Similarly, one node was the master running Hadoop's NameNode and the other 21 nodes acted as DataNodes.

TABLE III: Dataset Description

Dataset	Size	#Vertices	#Edges
Webbase	17.23G	118,142,155	1,019,903,190
Friendster	31.16G	65,608,366	1,806,067,135



(a) Connected Components (b) PageRank

Fig. 10: Checkpoint with 4 Instances per Machine

2) Workload: We restrict our performance evaluation to the *Connected Components* (*CC*) and *PageRank* (*PR*) algorithms. These two representative graph algorithms as well as their variants are widely used in applications such as community detection and web page ranking. They are implemented in most graph processing systems and are simple enough to serve as an effective measure of the system's performance [14]. For the rest of the paper, we refer to the Connected Components and PageRank algorithms as *CC* and *PR*, respectively.

3) DataSet: We evaluate our workload over two datasets: a web-scale URL link graph namely Webbase[1] [20] and a real-life social network graph called Friendster[2]. The dataset details are provided in Table III.

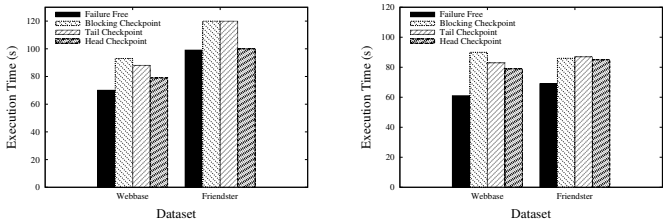
B. Efficiency of our Checkpointing

By default, we setup 4 instances on each physical machine and each instance has 5GB memory. For the execution of the workload, we run 11 supersteps of algorithms with checkpoints placed at superstep 6. In our implementation, *CC* treats the dataset as an undirected graph and generates the reverse edges before starting computation, while *PR* only applies computation based on the original dataset. Hence, *CC* needs to process more data than *PR*, which typically has a longer execution time.

Figure 10 shows the runtime of failure-free execution as well as the one with three different kinds of checkpointing strategies. In Figure 10a, the blocking checkpoint at superstep 6 of *CC* on Friendster incurs an additional overhead of 20% compared to the failure-free execution on Friendster dataset. The overhead of tail checkpoint is increased by 8% compared to failure-free execution, as tail checkpointing parallelizes the checkpoint writing with the production of a vertex dataset at the end of each superstep. The execution time of *CC* on the Friendster dataset with the head checkpoint at superstep 6 is almost the same as the one of failure-free execution, since head checkpointing parallelizes checkpoint writing with the graph computation of each superstep. Therefore, checkpointing in an unblocking manner outperforms its blocking counterpart.

[1]http://law.di.unimi.it/webdata/webbase-2001/

[2]http://snap.stanford.edu/data/com-Friendster.html



(a) Connected Components (b) PageRank

Fig. 11: Checkpoint with 8 Instances per Machine

Additionally, head checkpointing outperforms tail checkpointing. For the Webbase dataset, blocking checkpoint increases the overhead by 17% compared to failure-free execution. Tail checkpoint at superstep 6 increases the overhead of *CC* on the Webbase dataset by 11% compared to failure-free execution. However, head checkpointing incurs the same overhead as tail checkpointing in this case, as the data at the beginning of superstep 6 is consumed very fast by downstream operators, which limits the benefit of the parallelization with the checkpoint. We also evaluate *PR* on these two datasets as shown in Figure 10b. The trend of the execution time follows the one of *CC* on the Friendster dataset in Figure 10a.

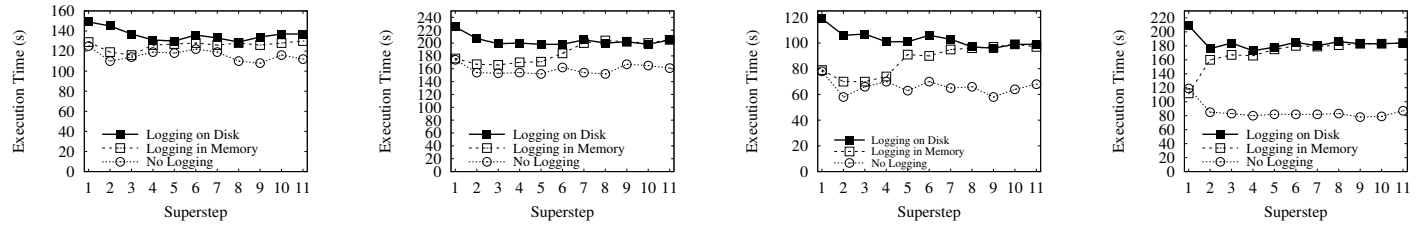
We increase the number of instances on each physical machine to be 8 and each instance has 3.5GB memory. The result in Figure 11 is similar to the one in Figure 10. For instance, in Figure 11a, blocking, tail and head checkpointing increase the overhead of *CC* algorithm on Webbase dataset by 33%, 23% and 13% respectively, compared to failure-free execution. It is interesting that the overhead of tail checkpointing for both *CC* and *PR* on the Friendster dataset is the same as that of blocking checkpointing, since the rate of generating an iterative dataset is much higher than the rate of writing it into HDFS. Also, the head checkpoint does not show benefit compared with the other two checkpointing strategies, due to the interference (δ_i in Equation 4) to the normal execution.

In general, the unblocking checkpointing is more efficient than blocking checkpointing and head checkpointing outperforms tail checkpointing. Comparing Figures 11 and 10, the normal execution is faster because of a higher parallelism with more memory adopted. In such case, the overhead of blocking checkpointing which materializes the data into HDFS becomes significant. Hence, there are more benefits if tail or head checkpointing is adopted.

C. Efficiency of our Recovery

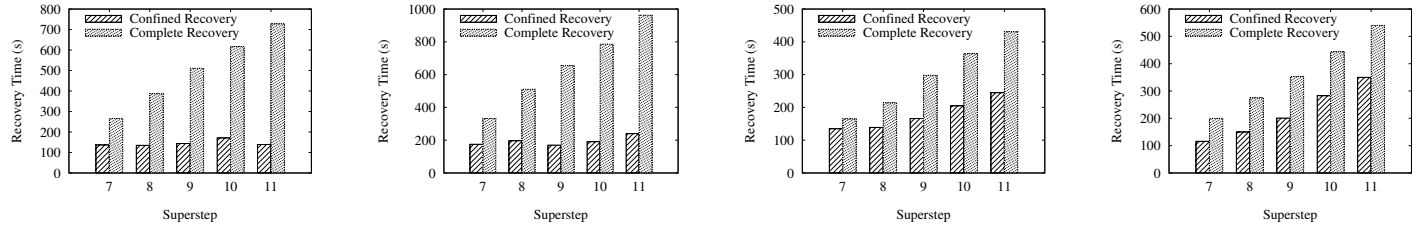
In this group of experiments, we study the logging overhead incurred by confined recovery and the performances of *confined recovery* versus *complete recovery* in terms of single, multiple and cascading failures. Here, we set 4 instances on each physical machine and execute the workloads.

To evaluate the overhead of logging, we run the workloads without and with logging enabled. Figure 12 shows the overhead of logging on disk and in memory respectively. In the worst case, each instance stores all the outgoing messages on disk as local logs which can incur a high overhead. That is, in order to enable confined recovery, it takes longer time during normal execution than complete recovery due to the overhead



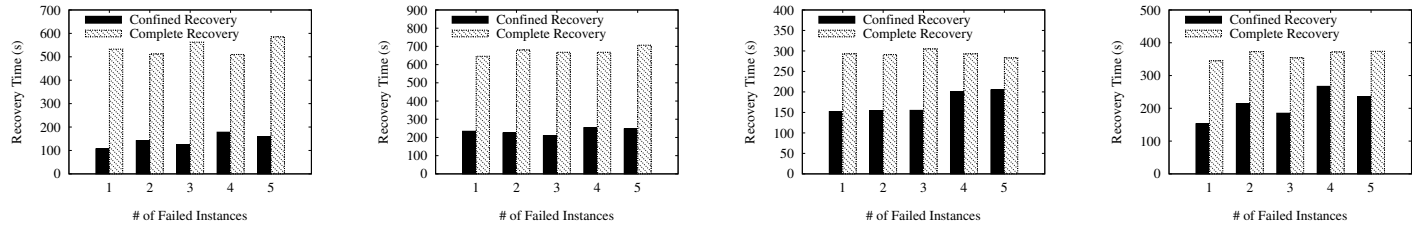
(a) Connected Components + Webbase (b) Connected Components + Friendster (c) PageRank + Webbase (d) PageRank + Friendster

Fig. 12: Overhead of Logging



(a) Connected Components + Webbase (b) Connected Components + Friendster (c) PageRank + Webbase (d) PageRank + Friendster

Fig. 13: Single Failure



(a) Connected Components + Webbase (b) Connected Components + Friendster (c) PageRank + Webbase (d) PageRank + Friendster

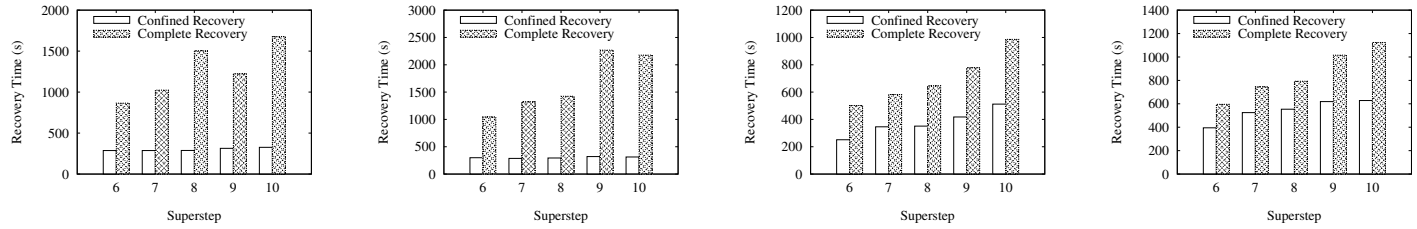
Fig. 14: Multiple Failures

of logging outgoing messages to local disks. Especially, this overhead of disk logging vs. logging-free on *PR* (Figure 12a and 12b) is higher than the one on *CC* (Figure 12c and 12d). Here, the normal execution in *PR* has less disk utilizations than the ones in *CC* as the former processes less data. Thus, the influence of overhead on logging in *PR* significantly slows down the execution. In addition, to keep more logs in memory, we increase the memory for each instance by 2GB and run the workloads again. Generally, logging in memory for the first several supersteps incurs low overhead. Along with the computation of supersteps, the buffered logs are accumulated in memory. It leads to either swapping with disk (see Figure 13b) or preempting the memory for normal execution (see Figure 13d). Both of the cases slow down the execution runtime.

The behaviors in Figure 12 give us a good hint as to when to write a checkpoint. For instance, it is necessary to apply a head checkpoint frequently for *PR* on the Friendster dataset and disable confined recovery, since the logging preempts the memory and slows down the execution runtime. However, we write checkpoints at superstep 6 for all the workloads to make a clear comparison in the following experiments. Also, during the recovery phase, reading the logs from disk or memory does not bring much benefits, since reloading or communication costs dominate the recovery time. Hence, we report the recovery time with logging on disk for the result of confined recovery in the following.

To evaluate the performance of recovery methods for single failure, we write a checkpoint at superstep 6 and kill the same instance in different supersteps after 6 respectively. As illustrated in Figure 13, the time of complete recovery increases along with the number of supersteps, since a failure at a later superstep recomputes more supersteps, which result in longer recovery times. Clearly, the time of confined recovery is less than that of complete recovery. For instance, confined recovery outperforms complete recovery by reducing the recovery time of 48% to 81% for single failures as shown in Figure 13a. In particular, failure detection and reloading data (η in Equation 9) influences the recovery phase significantly, since *PR* has a faster execution time with less data processed. Hence, the time difference of confined recovery vs. complete recovery on *PR* (see Figure 13c and 13d) is less than the one on *CC* (see Figure 13a and 13b).

For mulitple failures, we write a checkpoint at superstep 6 and kill instances when the algorithm is running at superstep 9.



(a) Connected Components + Webbase (b) Connected Components + Friendster (c) PageRank + Webbase (d) PageRank + Friendster

Fig. 15: Cascading Failures

More specifically, the number of killed instances is varied from 1 to 5. Figure 14 illustrates that the recovery time of complete recovery stays almost the same regardless of the number of failed instances, since complete recovery always reloads the full checkpoint and recomputes from this checkpoint onwards and also the number of healthy instances is approaching to the one before failure. Different from complete recovery, the trend of confined recovery's recovery time increases along with the increasing of failed instances, since more failed instances bring more lost data for confined recovery to process. However, the recovery time of confined recovery is still less than that of complete recovery. As an example in Figure 14a, confined recovery reduces the recovery time by 65% compared to complete recovery when 5 instances fail. Similar to the single failure, the relative benefits on *PR* are less significant than the one on *CC*, as failure detection and reloading data dominate the recovery time.

For cascading failures, we run the algorithms with a head checkpoint written at superstep 6 and kill one instance until the superstep reaches 11. Hence, the recovery methods recompute from the beginning of superstep 6. After that, we kill another instance during the recovery process. Figure 15 shows the runtime in terms of the superstep ranging from superstep 6 to superstep 10 in which we kill the second instance. In general, confined recovery outperforms complete recovery in this case of cascading failure. For example, in Figure 15a, confined recovery reduces the recovery time by 72% at superstep 7 for connected component on Webbase dataset compared to complete recovery. Again, more relative benefits on *CC* are acquired than the one on *PR* due to the domination by failure detection and data reloading on *PR*.

VI. Related Work

In this section, we list some works related to fault-tolerance with a focus on *checkpointing strategies* and *failure recovery*.

A. Checkpointing

In general, the purpose of checkpointing is to provide a snapshot of the data for failure recovery. A parallel dataflow framework named MapReduce [4] always materializes the results between map and reduce operators, i.e., checkpointing *operator-level*. Although it provides strong fault-tolerance, it considerably slows down the execution of iterative jobs (e.g. graph processing). Spark [5] writes the checkpoints for RDDs in the background without requiring program pauses [12] and Stratosphere manages the materialization of checkpoints locally by a cache, independent of the pipelined data flow [11]. However, it is oblivious to the iteration control for iterative algorithms such as graph processing, which complicates the system design since an additional component is needed to manage the checkpointing.

Most dataflow systems provide a graph processing interface. The dataflow based graph engines such as Pregelix [16] write checkpoints at the end of certain iterations into external storage such as HDFS. Some general dataflow based iterative computational systems also support some graph processing algorithms. For example, Mahout [21] uses an external driver program to control the loops, and new MapReduce jobs are launched at each iteration. It means Mahout writes the result, i.e., checkpoint, into HDFS at the end of each iteration. Instead of checkpointing at each iteration, HaLoop [13] enables users to specify the frequency of writing checkpoints to disk. However, these *iteration-level* checkpoints are typically written in a blocking fashion.

There also exist graph processing systems that are built based on the message passing paradigm. Pregel [1] uses the Bulk Synchronous Parallel (BSP) model [22] and writes checkpoints by letting the master instruct the workers to save the state of their partitions to persistent storage. However, it writes checkpoints in a blocking manner such that all computations are suspended while the checkpoint is constructed. Asynchronous computational model based systems like GraphLab [2] and PowerGraph [23] introduce a variant of the Chandy-Lamport snapshot checkpointing scheme with a low runtime overhead. However, it is specifically tailored for the asynchronous computational model.

To decide when a checkpoint is needed, a first-order approximation to the optimal checkpoint interval is derived in [17] for operating systems or general data management systems. In parallel data flow system FTOps [24], it showcases an optimizer that finds the best fault-tolerance strategy for each operator in a query plan. Also, a cost-based recovery [25] is proposed to select which intermediates should be materialized. These works are supplements to ours, but they focus on general workloads rather than iterative graph processing.

B. Failure Recovery

Checkpoint-based Recovery: The usual recovery method adopted in current dataflow systems or graph processing systems is checkpoint-based. Some systems provide *checkpoint-based* recovery at *operator-level*. In MapReduce, the results

of every individual program stage (the results of the Map stage in MapReduce) are checkpointed. These results are reprocessed upon operator failures. Mahout also writes checkpoints, i.e., the results of the Reduce stage, into HDFS at each iteration for iterative algorithms. Hence, Mahout provides a strong fault-tolerance at operator-level. Spark as well as GraphX [14, 26] recovers RDDs according to a lineage graph and utilizes checkpoints upon a failure. However, not all of the dataflow system maintains a lineage graph for RDDs. Some systems can support checkpoint-based recovery at *Iteration-level*. HaLoop simply reloads the checkpointed data and reactivates the iterative computation upon a failure. With asynchronous snapshotting, distributed GraphLab as well as PowerGraph repeats the iterations by reloading the snapshot. However, these systems apply a complete recovery with a long recovery time. Pregel employs a confined recovery based on the checkpoint. In particular, it adopts a newly-added node that substitutes the failed node, rolls back, and repeats the computations from the latest checkpoint. However, this new node affords all the recomputation workload. A partition-based reassignment algorithm [19] is proposed to accelerate the recovery process through simultaneous reduction of recovery communication costs and parallelization of the recovery computations. However, it relies on the statistics of the graph partition state kept by additional resources or components.

Application-based Recovery: Different from checkpoint-based recovery, some works aim to provide *application-based* recovery. In optimistic recovery [7, 27], the processing state can reach consistency even after failures using correct algorithmic compensations. However, defining the compensation function is non-trivial and such functions are only provided for very specific graph processing algorithms like link analysis and path enumeration. More recently, Zorro [28] opportunistically exploits vertex replication to quickly rebuild the state of failed servers for graph processing. It reduces the overhead during failure-free execution to zero, but sacrifices the completeness of the result, i.e., generating a slightly inaccurate result.

VII. Conclusion

There is a need for an efficient fault-tolerance mechanism that reduces the checkpointing cost and recovery time for iterative graph processing on distributed dataflow systems. To reduce the overhead of writing checkpoints, we present two checkpointing schemes: head and tail checkpointing. Unlike traditional approaches that manage checkpoints separately from the iteration pipeline, both our strategies inject checkpoints into the execution pipeline, thus eliminating the need for an additional checkpoint manager. Furthermore, in order to accelerate the recovery phase, we introduce confined recovery with logging locally during the groupBy stage in graph processing. The experimental and theoretical studies show that head checkpointing and confined recovery for fault-tolerance on graph processing outperforms blocking checkpointing and complete recovery. Presently, we integrate our proposed checkpointing and recovery strategies in the Flink dataflow engine. Nevertheless, it is possible to integrate our checkpointing strategies into dataflow systems like Spark and Dryad by integrating the checkpointing into the execution plans, while the confined recovery can be achieved in other dataflow system like HaLoop by locally logging the outgoing message during the shuffle phase.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work has been supported through grants by the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

[1] G. Malewicz *et al.*, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.

[2] Y. Low *et al.*, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[3] F. Bancilhon *et al.*, "An amateur's introduction to recursive query processing strategies," in *SIGMOD*, 1986, pp. 16–52.

[4] J. Dean *et al.*, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[5] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *HotCloud*, 2010, pp. 10:1–6.

[6] Apache Flink, http://flink.apache.org.

[7] S. Schelter *et al.*, "'all roads lead to rome': optimistic recovery for distributed iterative data processing," in *CIKM*, 2013, pp. 1919–1928.

[8] P. Lawrence *et al.*, "The pagerank citation ranking: Bringing order to the web," Stanford University, Technical Report, 1998.

[9] J. Pearl, "Reverend bayes on inference engines: A distributed hierarchical approach," in *AAAI*, 1982, pp. 133–136.

[10] A. Lancichinetti *et al.*, "Community detection algorithms: A comparative analysis," *Phys. Rev. E*, vol. 80, no. 5, pp. 056 117:1–11, 2009.

[11] A. Alexandrov *et al.*, "The stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.

[12] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[13] Y. Bu *et al.*, "Haloop: Efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1, pp. 285–296, 2010.

[14] J. E. Gonzalez *et al.*, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI*, 2014, pp. 599–613.

[15] M. Isard *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007, pp. 59–72.

[16] Y. Bu *et al.*, "Pregelix: Big(ger) graph analytics on a dataflow engine," *PVLDB*, vol. 8, no. 2, pp. 161–172, 2014.

[17] J. W. Young, "A first order approximation to the optimal checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[18] F. Hueske *et al.*, "Opening the black boxes in data flow optimization," *PVLDB*, vol. 5, no. 11, pp. 1256–1267, 2012.

[19] Y. Shen *et al.*, "Fast failure recovery in distributed graph processing systems," *PVLDB*, vol. 8, no. 4, pp. 437–448, 2014.

[20] P. Boldi *et al.*, "The webgraph framework I: compression techniques," in *WWW*, 2004, pp. 595–602.

[21] Apache Mahout, http://mahout.apache.org.

[22] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[23] J. E. Gonzalez *et al.*, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012, pp. 17–30.

[24] P. Upadhyaya *et al.*, "A latency and fault-tolerance optimizer for online parallel query plans," in *SIGMOD*, 2011, pp. 241–252.

[25] A. Salama *et al.*, "Cost-based fault-tolerance for parallel data processing," in *SIGMOD*, 2015, pp. 285–297.

[26] R. S. Xin *et al.*, "Graphx: a resilient distributed graph system on spark," in *GRADES*, 2013, pp. 2:1–6.

[27] S. Dudoladov *et al.*, "Optimistic recovery for iterative dataflows in action," in *SIGMOD*, 2015, pp. 1439–1443.

[28] M. Pundir *et al.*, "Zorro: zero-cost reactive failure recovery in distributed graph processing," in *SoCC*, 2015, pp. 195–208.