Better Safe than Sorry

Grappling with Failures of In-Memory Data Analytics Frameworks

Ghit, Bogdan; Epema, Dick

DOI
10.1145/3078597.3078600

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017

Better Safe than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks

Bogdan Ghiț
Delft University of Technology, the Netherlands
b.i.ghit@tudelft.nl

Dick Epema
Delft University of Technology, the Netherlands
d.h.j.epema@tudelft.nl

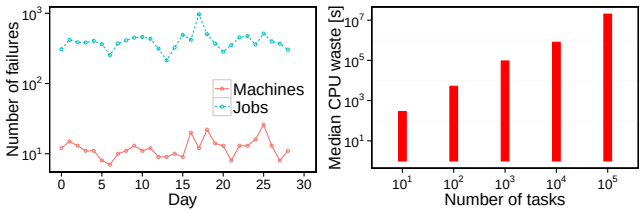
ABSTRACT

Providing fault-tolerance is of major importance for data analytics frameworks such as Hadoop and Spark, which are typically deployed in large clusters that are known to experience high failures rates. Unexpected events such as compute node failures are in particular an important challenge for in-memory data analytics frameworks, as the widely adopted approach to deal with them is to recompute work already done. Recomputing lost work, however, requires allocation of extra resource to re-execute tasks, thus increasing the job runtimes. To address this problem, we design a checkpointing system called PANDA that is tailored to the intrinsic characteristics of data analytics frameworks. In particular, PANDA employs fine-grained checkpointing at the level of task outputs and dynamically identifies tasks that are worthwhile to be checkpointed rather than be recomputed. As has been abundantly shown, tasks of data analytics jobs may have very variable runtimes and output sizes. These properties form the basis of three checkpointing policies which we incorporate into PANDA.

We first empirically evaluate PANDA on a multicluster system with single data analytics applications under space-correlated failures, and find that PANDA is close to the performance of a fail-free execution in unmodified Spark for a large range of concurrent failures. Then we perform simulations of complete workloads, mimicking the size and operation of a Google cluster, and show that PANDA provides significant improvements in the average job runtime for wide ranges of the failure rate and system load.

1 INTRODUCTION

The performance of large-scale data analytics frameworks such as Hadoop and Spark has received major interest [15, 22] from both academia and industry over the past decade. Surprisingly, this research assumes an ideal execution environment, which is in sharp contrast with the resilience-oriented design goals of these systems. In turn, these goals are motivated by the high rates of failures experienced by large-scale systems operating in clusters [12, 17] and datacenters [16, 20]. A key feature influencing the adoption of data analytics frameworks is their *fault-tolerant* execution model, in which a master node keeps track of the tasks that were running on machines that failed and restarts them from scratch on other machines. However, we face a fundamental limitation when the amount of work lost due to failure and re-execution is excessive because we need to allocate extra resources for recomputing work which was previously done. Frameworks such as Spark provide an API for checkpointing, but leave the decision of which data to checkpoint to the user. In this work, we design PANDA, a cluster scheduler that performs automatic checkpointing and so improves the resilience of in-memory data analytics frameworks.



(a) Failure rate (b) CPU waste

Figure 1: The average number of job and machine failures per hour (a) and the median CPU waste per job size range (b) in the Google trace. The vertical axes are in log-scale.

Failures in large-scale clusters are inevitable. The likelihood of having hardware crashes during the first year of a typical 10,000-machine cluster is very high according to several reports from the Google infrastructure team [5]. In particular, the system administrators expect about 1,000 individual machine failures and thousands of disk failures. In order to put into perspective the impact of failures on production workloads, we analyze failure reports from a Google cluster of 12,000 machines running half billion jobs over a month [16]. In Figure 1a we show the rate of machine and job failures in this Google cluster. Despite the relatively small number of machine failures (13 machines every hour), we observe a huge number of jobs (400 jobs every hour) that either fail, get killed by the system, or are simply abandoned by users. We expect this large number of failures to result into large amounts of wasted work. In Figure 1b we show the median job waste, that is the amount of work completed but lost due to failures for the complete range of job sizes (number of tasks). Indeed, the amount of wasted work increases linearly with the job size. The Google infrastructure is only one of a long series of multicluster systems experiencing problems in their infancy and in the long term. For example, the grid computing community has uncovered high failure rates [8], and in particular the flagship project CERN LCG had high failure rates years after going into production, with more than 25% unsuccessful jobs across all sites [3].

As today's clusters have large amounts of free memory [13], frameworks such as Spark advocate in-memory data processing, as opposed to previous on-disk approaches such as Hadoop. Unfortunately, as has been abundantly reported by the community, manipulating large datasets with Spark is challenging, and we have identified three causes of frequent failures in Spark that necessitate jobs to be restarted from scratch. First, the job runtime is very sensitive to the way the framework allocates the available memory during its execution. As a result, it may have variable performance

across different applications depending on how much memory they are allowed to use for storage and for job execution [22]. A second cause is that several built-in operators (e.g., groupBy, join) require that all values for one key fit in the memory. This constraint is in sharp contrast with the design of the framework which only supports coarse-grained memory allocation (per worker). Finally, memory-hungry tasks that produce a large number of persistent objects that stay in memory during the task runtime result in expensive garbage collection [13].

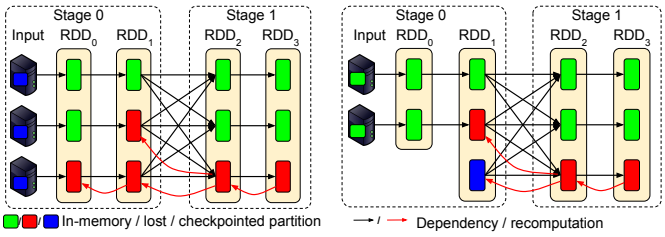
Using checkpointing to improve fault tolerance has a long history in computer systems [25]. In particular, the most commonly used method for checkpointing high-performance computing applications is coordinated checkpointing, where an application *periodically* stops execution and writes its current state to an external stable storage system. As setting the optimal checkpointing interval has been acknowledged as a challenging problem [9], existing solutions require the failure rates and the checkpointing cost to be known upfront, and to be constant over time. These assumptions are unrealistic for data analytics frameworks, which typically run computations in multiple inter-dependent stages each of which generates an *intermediate dataset* that is used as input by other stages. According to several reports from production clusters [4], the sizes of the intermediate datasets may vary significantly across stages of a single job, and as a result they cannot be anticipated.

Checkpointing a task has resource implications which are important to consider. While a task may be quickly recovered from a checkpoint, occupying an extra slot to perform the checkpoint may increase the job runtime due to the high cost of reliably saving the task's output. To remedy this, we propose PANDA, a checkpointing system that carefully balances the opportunity cost of persisting a task's output to an external storage system and the time required to recompute when the task is lost. This opportunity cost is driven by the evidence of *unpredictable intermediate data sizes* and *outlier tasks* of jobs in production traces from Google and Facebook [4, 16], which form the basis of our checkpointing policies. Firstly, we propose the *greedy* policy that greedily selects tasks for checkpointing until a predefined budget is exceeded. Secondly, our *size-based* policy considers the longest tasks of a job because those tasks are more likely to delay the job completion if they are lost. Finally, we design the *resource-aware* policy that checkpoints tasks only if their recomputation cost is likely to exceed the cost of checkpointing it.

In this paper we make the following contributions:

(1) We design PANDA, a fine-grained checkpointing system that checkpoints tasks at stage boundaries by persisting their output data to stable storage (Section 3). We reduce the checkpointing problem to a task selection problem and we incorporate into PANDA three policies designed from first principle analysis of traces from production clusters. These policies take into account the size of task output data, the distribution of task runtimes, or both (Section 4).

(2) With a set of experiments in a multicluster system, we analyze and compare the performance of our policies with single, failing applications under space-correlated failures (Section 5). With a set of large-scale simulations, mimicking the size and the operation of a Google cluster, we analyze the effectiveness of PANDA in reducing the average job runtime of a complete workload (Section 6).



(a) Recomputation (b) Checkpointing

Figure 2: An example of a lineage graph with data dependencies between RDD partitions. The recomputation tree of a missing partition in unmodified Spark (a) and in Spark with checkpointing (b). All lost partitions are located on a single machine and the input dataset is replicated in stable storage.

2 SYSTEM MODEL

In this section we present the main abstractions used by Spark to perform both efficient and fault-tolerant in-memory data processing (Section 2.1). Furthermore, we describe the scheduling mechanism employed by Spark to execute parallel jobs on a cluster with many machines (Section 2.2).

2.1 Lineage Graphs

We explain the RDD data abstraction used by Spark to persist large datasets in the memory of multiple cluster machines and we discuss the notion of *lineage graph*, a fault-tolerant data structure that guards the framework against data loss when machine failures are expected.

Data analytics frameworks such as Spark [26] leverage the distributed memory of the cluster machines with a new abstraction called *resilient distributed datasets* (RDDs), which provides efficient data processing across a broad range of applications (SQL queries, graph processing, machine learning, and streaming). An RDD is a collection of *data partitions* distributed across a set of cluster machines. Users have access to a rich set of transformations (e.g., map, filter, join) to create RDDs from either data in stable storage (e.g., HDFS, S3) or other RDDs. Typically, such transformations are *coarse-grained* because they apply the same operation in parallel to each partition of the RDD.

RDDs may not be materialized in-memory at all times. Instead, Spark maintains the sequence of transformations needed to compute each RDD in a data structure called the *lineage graph*. In other words, the lineage graph is a directed acyclic graph (DAG) where a vertex represents an RDD partition and an incoming edge represents the transformation used to compute the RDD. Furthermore, Spark distinguishes two main types of data dependencies between RDDs: (1) the *narrow* dependency, in which each partition of the parent RDD is used by at most one partition of the child RDD (e.g., map, filter), and (2) the *wide* dependency, in which multiple child partitions may depend on the same parent partition (e.g., join, groupBy).

As RDDs are typically persisted in volatile memory without replicas, a machine failure causes the loss of all partitions that are located on it. Spark automatically recovers a missing partition by identifying in the lineage graph its *recomputation tree*, which is the minimum set of missing ancestor partitions and the dependencies among them needed to recover the partition. Thus, the *critical recomputation path* of a given partition is the sequence of partitions in its recomputation tree that determine the minimum time needed to recover the partition. In the worst case, the critical recomputation path may go back as far as the origin of the input data. Then, Spark applies for each missing partition the sequence of transformations in its recomputation tree according to the precedence constraints among them. As different partitions of the same RDD may have different recomputation trees, the recovery of a complete RDD typically results in recomputing a sub-DAG of the initial lineage graph.

To avoid long critical recomputation paths, Spark allows its users to *cut-off* the lineage graph through a *checkpointing* operation that reliably saves a complete RDD to stable storage. Checkpointing an RDD in Spark is similar to how Hadoop spills shuffle data to disk, thus trading off execution latency with fast recovery from failures. Figure 2 shows an example of a lineage graph for a simple Spark computation, with both narrow and wide dependencies between RDDs. The figure depicts the recovery of a missing partition by recomputing all its ancestors (a) and by reading an existing checkpoint (b).

Spark exposes a basic interface for checkpointing complete RDDs, but it is the user's decision to select which RDDs to checkpoint. As the intermediate RDD sizes are not known upfront, selecting RDDs statically, prior to the execution of an application, is difficult. Spark checkpoints a given RDD by creating a parallel job with tasks that save the RDD partitions from memory to stable storage. However, when the memory is fully utilized, Spark evicts RDD partitions using a *least-recently-used* (LRU) policy. This way of checkpointing RDDs is inefficient because it may trigger recomputations if some RDD partitions are evicted from memory.

2.2 DAG Scheduler

We present an overview of the scheduling architecture used by Spark to (re-)allocate compute slots to jobs that consist of multiple sets of tasks with precedence constraints among them.

To compute an RDD, Spark's scheduler creates a job by translating the RDD dependencies in the lineage graph into a DAG of *processing stages*. Each stage consists of a set of *parallel tasks* that apply the same operation (transformation) to compute independently each RDD partition. In this DAG, tasks pipeline as many transformations with narrow dependencies as possible, and so we identify *stage boundaries* by transformations with wide dependencies. Such transformations typically require a *shuffle* operation, as illustrated in Figure 2. A shuffle operation splits the output partitions of each task in the *parent* stage into multiple shuffle files, one for each task in the *child* stage. Tasks in the child stage may only run once they have obtained all their shuffle files from the parent stage.

In order to compute an RDD, the scheduler executes tasks in successive stages on *worker* machines based on their precedence constraints (data dependencies), data locality preferences (run tasks closer to input data), or fairness considerations (per job quotas). Similarly to Dryad and MapReduce, Spark jobs are *elastic* (or *malleable*) and can run simultaneously, taking any resources (compute slots) they can get when it is their turn. The DAG scheduler in Spark schedules the tasks of a stage only after all its parent stages have generated their output RDDs. Scheduling tasks based on a strict queueing order such as *first-in-first-out* (FIFO) compromises locality, because the next task to schedule may not have its input data on the machines that are currently free. Spark achieves task locality through delay scheduling, in which a task waits for a limited amount of time for a free slot on a machine that has data for it.

Next, we present the main mechanisms that Spark uses to detect and to recover from worker failures. Similarly to other fault-tolerant cluster frameworks, Spark relies on timeouts and connection errors to infer worker failures. The scheduler expects heartbeats from its healthy *workers* every 10 seconds, and marks as lost a worker that has not sent any heartbeat for at least 1 minute. A dead worker not only leads to the failure of its running tasks, but also makes all previously computed work on that worker unavailable. As a consequence, tasks that fail to transfer data from a lost worker trigger fetch errors that may also serve as an early indication of a failure. Spark re-executes failed tasks as long as their stage's parents are still available. Otherwise, the scheduler resubmits tasks recursively in parent stages to compute the missing partitions.

3 DESIGN CONSIDERATIONS

In this section we identify three techniques for checkpointing in-memory data analytics jobs (Section 3.1). Moreover, we investigate the main properties of workloads from Facebook and Google that we use as first principles in the design of our checkpointing policies (Section 3.2). Finally, we propose a scheduling and checkpointing structure for automatic checkpointing of data analytics jobs (Section 3.3).

3.1 Checkpointing Tasks

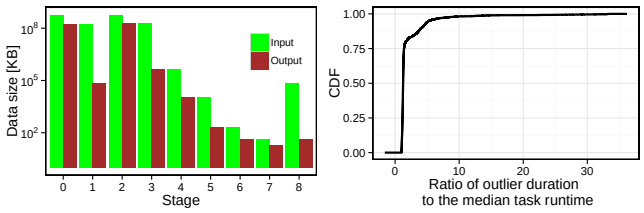
The basic fault-tolerance mechanism used by data analytics frameworks to mitigate the impact of machine failures is to recompute lost data by repeating tasks based on their precedence constraints in the lineage graph. Obviously, this approach may be time-consuming for applications with large lineage graphs. Checkpointing the running tasks of a job to stable storage allows the job to only partially recompute data generated since the last checkpoint. However, checkpointing introduces an overhead proportional to the size of the data persisted to stable storage.

We identify different ways of checkpointing data analytics jobs. One way of doing so is to employ traditional checkpointing mechanisms available in operating systems that suspend the execution of running tasks and store their states for later resumption. In this method, checkpointing jobs is performed *at any point*, as opposed to the later two approaches. However, this process may degrade performance considerably and may trigger frequent machine reboots [11]. Tasks of in-memory data analytics jobs are allocated large heap sizes of multiple GBs, and so checkpointing their states is relatively slow. In addition, recovery from a checkpoint stored on another machine triggers additional network traffic, which may hurt the performance of other jobs in the cluster.

Another approach is to checkpoint tasks at *safe points* from where the remaining work can be executed without requiring any context

Table 1: The workload traces from two large production clusters at Facebook [4] and Google [16].

Trace	Facebook	Google
Dates	October 2010	May 2011
Duration (days)	45	29
Framework	Hadoop	Borg
Cluster size (machines)	600	12,000
Number of jobs	25,000	668,048
Task runtimes	No	**Yes**
Data sizes	**Yes**	No
Failed machines per hour	Unknown	7 to 25



(a) Data sizes **(b) Task runtimes**

Figure 3: The variability of the intermediate data sizes (a, vertical axis in log scale) and the prevalence of outliers (b) in the BTWORLD application.

from the current execution. At a higher level, tasks in data analytics jobs pipeline a sequence of narrow transformations between successive RDD partitions. Tasks split each RDD partition they process into a sequence of *non-overlapping subsets* each of which may have multiple records that share the same *key*. Thus, a natural way to checkpoint tasks for many transformations (e.g., map, reduce, join) is at key boundaries, when all the processing for a key is complete. This approach has been previously proposed in Amoeba [1], a system that aims at achieving true elasticity by trimming the durations of long task through checkpointing. However, because tracking such safe points in data analytics frameworks is notoriously difficult, as they typically require a global view of intermediate data, Amoeba originally supported only a small number of transformations in MapReduce frameworks and has not evolved since.

Finally, we can checkpoint tasks at *stage boundaries* by persisting their *output data* to stable storage. A stage boundary for a task is the point from where the output data is split into multiple shuffle files each of which aggregates input data for a single reducer. Shuffle files are written to the buffer cache, thus allowing the operating system to flush them to disk when the buffer capacity is exceeded. Because checkpointing shuffle files requires complex synchronization between multiple tasks that write sequentially to the same shuffle file, we perform checkpointing on the output data before splitting it into shuffle files. We choose this way of checkpointing because it integrates well with the lineage-based mechanism adopted by current frameworks. We need to recompute a task when either the machine on which it runs fails, or when (part of) the output it produced was located on a machine that fails and is still needed. Checkpointing tasks at stage boundaries helps only in the latter case. After checkpointing the output of a task completely, we no longer need to know how to compute or recover its input, and so we can cut-off its lineage graph.

3.2 Task Properties

Although there is a rich body of work that studies the characteristics of datacenter workloads [4, 16], not many public traces exist. The largest traces available are from the Hadoop production cluster at Facebook [4] and from Google's Borg resource manager [16]. Table 1 shows the relevant details of these traces. An investigation of these traces reveals that datacenter workloads are largely dominated by the presence of outlier tasks, and that the sizes of the intermediate data of jobs may be very variable. Although both traces are relatively old, it is unlikely that these task properties have changed since their collection. In this section we check their validity by analyzing the BTWORLD application [10], which we use to process *monitoring data* from the BitTorrent global network; this application is later described in Section 5.

Unlike a job's input size, which is known upfront, intermediate data sizes cannot be anticipated. Complex applications such as BTWORLD consist of many processing stages, out of which only a few require the complete input, while the others run on intermediate data. Figure 3a shows that there is no strong correlation between the input and output data sizes, and that the output sizes range from a few KB to hundreds of GB. We compute the stage selectivity, defined as the ratio of the output size and the input size, for each job in the Facebook trace. We find that the stage selectivities may span several orders of magnitude: a small fraction of the stages perform data transformations (selectivity of 1), while the large majority are either data compressions (selectivity less than 1) or data expansions (selectivity higher than 1).

In data analytics workloads, tasks may have inflated runtimes due to poor placement decisions (resource contention) or imbalance in the task workload (input data skew). Indeed, Figure 3b shows that 70% of the task outliers in the BTWORLD application have a uniform probability of being delayed between 1.5x and 3x the median task runtime. The distribution is heavy-tailed, with top 5% of the outliers running 10x longer than the median. Similarly, the tasks in the Google cluster are also very variable and fit well a heavy-tailed distribution (Pareto with shape parameter 1.3).

We use the large variability of the intermediate data sizes in the design of a *greedy* checkpointing policy, which employs a specified budget to avoid excessive checkpointing. Similarly, the prevalence of outliers forms the basis of a *size-based* checkpointing policy, which seeks to checkpoint the long running tasks in a job. Finally, we use both properties in a *resource-aware* checkpointing policy, which checkpoints tasks only when the cumulative cost of recomputing them is larger than the cost of checkpointing. In Section 4 we present the design of our policies starting from first principles, with all the features needed to perform well in a datacenter.

3.3 Checkpointing Architecture

We present the main design elements and the operation of PANDA, an adaptive checkpointing system for in-memory data analytics jobs which integrates well with the architecture of current framework schedulers.

Figure 4 shows the architecture of a typical data analytics framework, with a cluster-wide *job scheduler* and a fault-tolerant *distributed filesystem* which coordinate the execution of tasks on a set of cluster machines with co-located processors and storage volumes

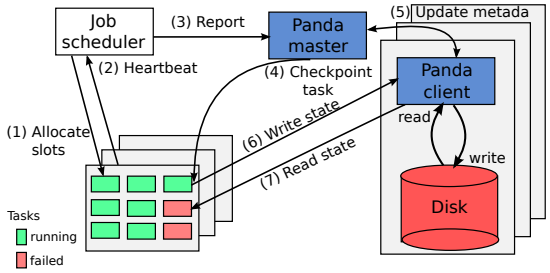


Figure 4: The system architecture for the PANDA checkpointing mechanism in data analytics frameworks.

(illustrated for simplicity as separate entities). The job scheduler handles the allocation of compute slots to numerous parallel tasks of a data analytics job with user-defined constraints (step 1) and waits for periodic heartbeats to keep track of the state of the running tasks (step 2). The distributed filesystem (e.g., HDFS in our deployment) employs a three-way replication policy for fault-tolerance and allows our system to *reliably persist* a data analytics job by saving its input, intermediate, or output datasets.

PANDA's architecture consists of a *checkpoint master* and a set of *clients* located at each cluster machine. The PANDA master is periodically updated by the job scheduler with progress reports of the running tasks (step 3). A progress report incorporates for each task the following properties: the amount of input/output data size read/written so far and the current task runtimes. The master's main role is to decide *when* to checkpoint running tasks and *which* among the running tasks of a job to checkpoint (step 4). To do so, PANDA employs one of the policies presented in Section 4.

The checkpoint master receives updates from clients with the location of each checkpoint in the reliable storage system and maintains a global mapping between every checkpointed partition and the dataset it belongs to (step 5). The PANDA clients access the distributed filesystem for saving and/or fetching partitions on behalf of the job (steps 6 and 7). Thus, before a task starts running, it first uses the PANDA client to retrieve from the checkpoint master the location of its checkpoint. The PANDA client fetches the checkpoint from the distributed filesystem so that the task gracefully resumes its execution from that point onwards. If the task was not previously checkpointed, it executes its work completely.

4 CHECKPOINTING POLICIES

We will now address the question of which subsets of tasks to checkpoint in order to improve the job performance under failures while keeping the overhead of checkpointing low. The policies we propose for this purpose may use the size of the task output data (GREEDY), the distribution of the task runtimes (SIZE), or both (AWARE). Furthermore, we use an adaptation of the widely known periodic checkpointing approach (PERIODIC) to data analytics frameworks that periodically checkpoints all completing tasks. In Table 2 we state the main differences between our policies.

Greedy checkpointing. Our GREEDY policy seeks to limit the checkpointing cost in every stage of a job in terms of the amount of data persisted to disk to a specified *budget*. Intuitively, we want to reduce the number of recomputations after a failure in a *best-effort* way by selecting in each stage as many tasks for checkpointing as the budget allows. The GREEDY policy sets the checkpointing budget of a stage to some fraction of the size of the total input data transferred to it from the tasks of its parent stages. This fraction may depend on the selectivities of the tasks of the stage—if the latter are low, the fraction can be small. For example, for the BTWORLD workflow with a median task selectivity of 0.1, it can be set to 10%.

Table 2: PANDA's policy framework for checkpointing in-memory data analytics jobs in datacenters.

Policy	Data size	Task runtime	Description
GREEDY	yes	no	fraction of the input data
SIZE	no	yes	longest tasks in the job
AWARE	yes	yes	checkpoint vs. recompute
PERIODIC	yes	no	every τ seconds

The GREEDY policy is invoked for every stage of a job once all its parent stages have generated their output RDDs. It will then start checkpointing *any* completing task as long as it does not exceed the stage's budget. Tasks that are in the process of checkpointing when the budget is exceeded are allowed to complete their checkpoints.

Size-based checkpointing. Our SIZE policy aims to reduce the amount of work lost after a failure by checkpointing *straggler* tasks that run (much) slower than other tasks of the job. The main intuition behind the SIZE policy is to avoid recomputing time-consuming tasks that prevent pending tasks of the job from starting.

Straggler tasks in data analytics frameworks may be due to large variations in the *code* executed and the *size of the data* processed by tasks. Across all stages of the BTWORLD workflow, the coefficient of variation in task runtimes is 3.4. Although the code is the same for all tasks in each stage, it differs significantly across stages (e.g., map and reduce). Furthermore, the amount of data processed by tasks in the same stage may vary significantly due to limitations in partitioning the data evenly.

The SIZE policy now works in the following way. In order to differentiate straggler tasks, SIZE builds up from scratch for every running job a history with the durations of its finished tasks. Thus, at any point in time during the execution of a job, SIZE has an estimation of its median task runtime, which becomes more accurate as the job completes a larger fraction of its tasks. SIZE checkpoints only those tasks it considers stragglers, that is, tasks whose durations are at least some number of times (called the *task multiplier*) as high as the current estimation of the median task runtime.

Resource-aware checkpointing. The AWARE policy aims to checkpoint a task only if the estimated benefit of doing so outweighs the cost of checkpointing it. We explain below how the AWARE policy estimates both the recomputation and the checkpointing cost of a task, which is done after it has completed.

Prior to the execution of a job, AWARE sets the probability of failure by dividing the number of machines that experienced failures during a predefined time interval (e.g., a day) by the cluster size. AWARE derives these data from the operation logs of the cluster that contain all machine failing events.

A machine failure may cause data loss, which may require recomputing a task if there are pending stages that need its output in order to run their tasks. However, the recomputation of a task may cascade into its parent stages if its inputs are no longer available and need to be recomputed in turn. We define the DAG level of a task as the length of the longest path in the lineage graph that needs to be recomputed to recover the task from a failure.

AWARE estimates the recomputation cost of a task as the product of the probability that the machine on which it ran fails and its *recovery time*, which is the actual cost of recomputing it, including the recursive recomputations if its recomputation cascades into its parent stages. When the input files of a lost task are still available, either in the memory of other machines or as checkpoints in stable storage, the recovery time is equal to the task runtime. If multiple input files of a task to be recomputed are lost, we assume that they can be recomputed in parallel, and we add the maximum recomputation cost among the lost tasks in its parent stages to its recovery time. We do this recursively as also input files of tasks in parent stages may be lost in turn.

The checkpointing cost of a task is a function of the amount of data that needs to be persisted, the write throughput achieved by the local disks the task is replicated on, and the contention on the stable storage caused by other tasks that are checkpointed at the same time. While the former two may be anticipated, the latter is highly variable and difficult to model accurately. In particular, checkpointing a task along with other tasks that require replicating large amounts of data to stable storage may inflate the checkpointing cost. In order to solve this problem, we propose the following method to approximate the checkpointing cost of a task in a given stage. When a stage starts, we artificially set the cost of checkpointing its tasks 0, thus making AWARE checkpoint the first few waves of tasks in order to build up a partial distribution of task checkpointing times. Then we let AWARE set the checkpointing cost of a task to the 95^{th} percentile of this distribution, which is all the time adapted with the checkpointing times of the checkpointed tasks in the stage.

The AWARE policy now works in the following way. It is invoked whenever a stage becomes eligible for scheduling its tasks, and then, using the job's lineage graph, it estimates the recomputation costs of its tasks. We amortize the checkpointing cost of a task by its DAG level, so that tasks with long recomputation paths are more likely to be checkpointed. AWARE checkpoints only those tasks whose potential resource savings are strictly positive, that is, tasks whose recomputation costs exceed the amortized checkpointing cost.

5 EXPERIMENTAL SETUP

We evaluate the checkpointing policies described in Section 4 through experiments on the DAS multicluster system. In this section we present the cluster configuration and the data analytics benchmarks that we use to assess the performance of PANDA.

5.1 Cluster Setup

We have implemented PANDA in Spark and we evaluate our checkpointing policies on the fifth generation of the Dutch wide-area computer system DAS [7]. In our experiments we use DAS machines that have dual 8-core compute nodes, 64 GB memory, and two 4 TB disks, connected within the cluster through 64 Gbit/s FDR InfiniBand network. We perform experiments with two cluster configurations for long and short jobs with allocations of 20 and 5 machines, respectively.

We co-locate PANDA with an HDFS instance that we use to store the input datasets and the checkpoints performed by our policies. We setup the HDFS instance with a standard three-way replication scheme and a fixed data block size of 128 MB. We assume the HDFS instance runs without failures so that both the input datasets and the checkpoints are always available for PANDA.

Table 3: The cluster configurations for our applications.

Application	Benchmark	Nodes	Dataset	Input [GB]	Runtime [s]
BTWORLD	real-world	20	BitTorrent	600	1587
PPPQ	standard	20	TPC-H	600	1461
NMSQ	standard	20	TPC-H	600	656
PageRank	real-world	5	Random	1	128
KMeans	real-world	5	Random	10	103

We want to analyze the performance of typical data analytics applications under different patterns of *compute node failures*. Therefore, we consider Spark *worker failures*, which may cause loss of work already done that is stored in the local memory of the workers. We assume that new worker machines may be provisioned immediately to replace the lost workers, so that the size of our cluster remains constant during the execution of the application.

We clear the operating system buffer cache on all machines before each experiment, so that the input data is loaded from disk. To emulate a production environment with long-running processes, we warm up the JVM in all our experiments by running a full trial of the complete benchmark. For the experiments we show in Section 6, we report the mean over three executions.

5.2 Applications

In our evaluation we use a diverse set of applications ranging from real-world workflows to standard benchmarks that are representative for data analytics frameworks. Table 3 presents the configuration we use in our experiments for each job. We analyze the performance of PANDA with both long-running jobs that have durations in the order of tens of minutes (e.g., BTWORLD, PPPQ, and NMSQ) and short interactive jobs that take minutes to complete (e.g., PageRank and KMeans). We describe these jobs in turn.

BTWorld. The BTWORLD [21] application has observed since 2009 the evolution of the global-scale peer-to-peer system BitTorrent, where files are broken into hashed pieces and individually shared by users, whether they have completely downloaded the file or not. To help users connect to each other, BitTorrent uses trackers, which are centralized servers that give upon request lists of peers sharing a particular file. BTWORLD sends queries to public trackers of the BitTorrent system and so it collects statistics about the aggregated status of users. These statistics include for each *swarm* in the tracker (users who share the same torrent file) the number of *leechers* (users who own some but not all pieces of the file), the number of *seeders* (users who own all pieces of the file), and the total number of *downloads* since the creation of the torrent.

We have designed a MapReduce-based workflow [10] to answer several questions of interest to peer-to-peer analysts in order to understand the evolution over time of the BitTorrent system. The complete BTWORLD workflow seeks to understand the evolution of each individual tracker we monitor, to determine the most popular trackers (over fixed time intervals and over the entire monitoring period), and to identify the number of new swarms created over time. In our experiments with PANDA, BTWORLD takes an input dataset of 600 GB. As we show in Figure 5a, the lineage graph of BTWORLD consists of a long chain of stages and a single join.

TPC-H. The TPC-H benchmark [2] consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The

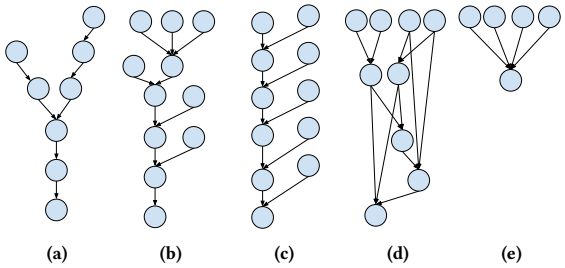


Figure 5: The data flow of BTWORLD (a), PPPQ (b), NMSQ (c), PageRank (d), and KMeans (e) as a DAG of stages: the nodes and the edges represent the stages and the wide dependencies among them.

queries and the dataset have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that analyze large volumes of data, execute queries with high degrees of complexity, and give answers to critical business questions. The benchmark randomly generates eight relational tables with a schema that represents a typical data warehouse dealing with sales, customers, and suppliers. For a detailed description of the benchmark we refer to its standard specification [2].

We use two queries from this benchmark: the Potential Part Promotion Query (PPPQ) and the National Market Share Query (NMSQ). PPPQ seeks candidates for a promotional offer by selecting suppliers in a particular nation that have an excess of a given product – more than 50% of the products shipped in a given year for a given nation. NMSQ determines how the market share of a given nation within a given region has changed over two years for a given product. In our experiments we execute both queries with an input dataset of 600 GB. Figures 5b and 5c show the lineage graphs of both TPC-H queries that combine in almost every stage results from two or three parent stages.

PageRank. PageRank is the original graph-processing application used by the Google search engine to rank documents. PageRank runs multiple iterations over the same dataset and updates the rank of a document by adding the contributions of documents that link to it. On each iteration, a document sends its contribution of r_i/n_i to its neighbors, where r_i and n_i denote its rank and number of neighbors, respectively. Let c_{ij} denote the contribution received by a document i from its neighbor j. After receiving the contributions from its neighbors, a document i updates its rank to $r_i = (\alpha/N) + (1 - \alpha)\sum c_{ij}$, where N is the total number of documents and α a tuning parameter.

We use the optimized PageRank implementation from the `graphx` library of Spark with a 1 GB input dataset. We generate a random input graph with 50,000 vertices that has a log-normal out-degree distribution with parameters μ and σ set to 4 and 1.3, respectively. In Figure 5d we show the lineage graph for a single iteration of PageRank. An interesting property of this application is that its lineage becomes longer with the number of iterations.

KMeans. Clustering aims at grouping subsets of entities with one another based on some notion of similarity. KMeans is one of the most commonly used clustering algorithms that clusters multidimensional data points into a predefined number of clusters. KMeans uses an iterative algorithm that alternates between two main steps. Given an initial set of means, each data point is assigned to the cluster whose mean yields the least *within-cluster sum of squares* (WCSS). In the update step, the new means that become the *centroids* of the data points in the new clusters are computed.

We use the optimized implementation from the `mllib` library of Spark with a 10 GB dataset that consists of 10 millions data points sampled from a 50-dimensional Gaussian distribution. Figure 5e shows that KMeans with four iterations has a relatively simple lineage graph, with a single shuffle operation that combines results from multiple stages that have narrow dependencies to the input dataset.

Table 4: An overview of the experiments performed to evaluate PANDA.

Experiment	Jobs	Policies	Baselines	Failure pattern	Sec.
Parameters	BTWORLD PPPQ	all	none	none	6.1
Overhead	all	all	Spark	none	6.2
Machine failures	BTWORLD PPPQ NMSQ	all	Spark	space-correlated	6.3
Lineage length	PageRank KMeans	AWARE	Spark	single failure	6.4
Simulations	BTWORLD PPPQ PageRank	AWARE	Spark	space-correlated	6.5

6 EXPERIMENTAL EVALUATION

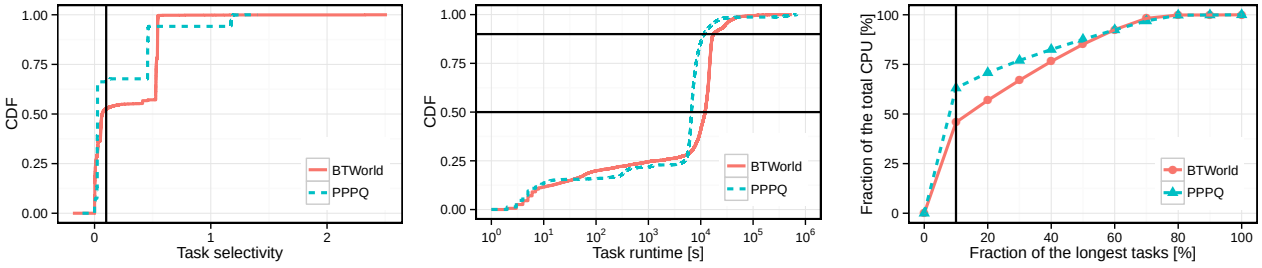
In this section we present the results of five sets of experiments that each address a separate aspect of the performance of PANDA. Table 4 presents an overview of these experiments. We investigate the setting of the parameters in GREEDY, SIZE, and AWARE (Section 6.1). We measure the checkpointing overhead to determine how far we are from the default Spark implementation without checkpointing (Section 6.2). Thereafter, we evaluate the performance of our policies under various patterns of space-correlated failures (Section 6.3). Moreover, we assess the impact of the length of the lineage graph on the performance of PANDA when failures are expected (Section 6.4). Finally, we perform simulations to evaluate the benefit of checkpointing at larger scale (Section 6.5).

6.1 Setting the Parameters

All our policies have parameters needed in order to operate in a real environment. In particular, GREEDY and SIZE use the checkpointing budget and the task multiplier, respectively. In contrast with these policies that both set workload-specific parameters, AWARE sets the probability of failure that quantifies the reliability of the machines, and so it is independent of the workload properties. In this section we seek to find good values of these parameters.

One way of setting the parameters for GREEDY and SIZE is to evaluate the performance of each policy for a range of parameter values with various failure patterns. However, this method is time-consuming, and may in practice have to be repeated often. To remove the burden of performing sensitivity analysis for each policy, we propose two simple rules of thumb based on the history of job executions in production clusters and in our DAS multicluster system. We show in Sections 6.2 through 6.5 that our policies perform well with these rules.

The GREEDY *policy sets the checkpointing budget to the median selectivity of the tasks across all jobs that we use in our experiments.* The



(a) Task selectivities **(b) Task runtimes** **(c) Processing time**

Figure 6: The distributions of the task selectivity (a) and the task runtime (b) for BTWORLD and PPPQ, and the fraction of the total CPU time versus the fraction of the longest tasks in BTWORLD and PPPQ (c). The vertical lines represent the task selectivity of 0.1 (a) and the longest 10% tasks (c), and the horizontal lines represent the median and the 90^{th} percentile of the task runtimes (b).

checkpointing budget limits the amount of data that is replicated to HDFS in each stage. We expect GREEDY to have a large overhead when setting the checkpointing budget to a large value. In Section 3.2 we have shown that in the Facebook production cluster, a large majority of tasks have relatively low selectivities. Figure 6a shows the distributions of the task selectivity in BTWORLD and PPPQ. Because the median task selectivity is below 0.1 for both jobs, we set the checkpointing budget in our experiments with GREEDY to 10%.

The SIZE policy sets the task multiplier to the ratio of the 90^{th} percentile and the median of the runtimes of the tasks across all jobs that we use in our experiments. The task multiplier aims at identifying the longest tasks in a job, and so setting a small value may result in checkpointing a large fraction of tasks. As we have shown in Section 3.2, this is unlikely to happen for data analytics jobs because they typically run tasks that have heavy-tailed durations. Figure 6c shows that only 10% of tasks in BTWORLD and PPPQ account for roughly 50% of the total processing time of the job. As Figure 6b shows, the ratio of the 90^{th} percentile and the median of the distribution of task runtimes for both BTWORLD and PPPQ is 1.5. Thus, we set the task multiplier in our experiments with SIZE to 1.5.

Unlike the previous two policies, which both require an analysis of task properties, the AWARE policy only needs as parameter the likelihood of being hit by a failure. We want AWARE to checkpoint more tasks as it operates on less reliable machines and vice versa. In order to highlight the checkpointing overhead and the performance of AWARE in unfavorable conditions, we assume that all machines allocated to execute our jobs experienced failures. Thus, we set the probability of failure in our experiments with AWARE to 1.

Finally, in order to show the improvements provided by our policies relative to the traditional way of checkpointing, in our experiments with the PERIODIC policy we set the optimal checkpointing interval based on Young's approximation for each application. To do so, our version of the PERIODIC policy requires an estimation of the checkpointing cost and the mean time to failure. Thus, we assume that we know prior to the execution of each job both its checkpointing cost and the failure time.

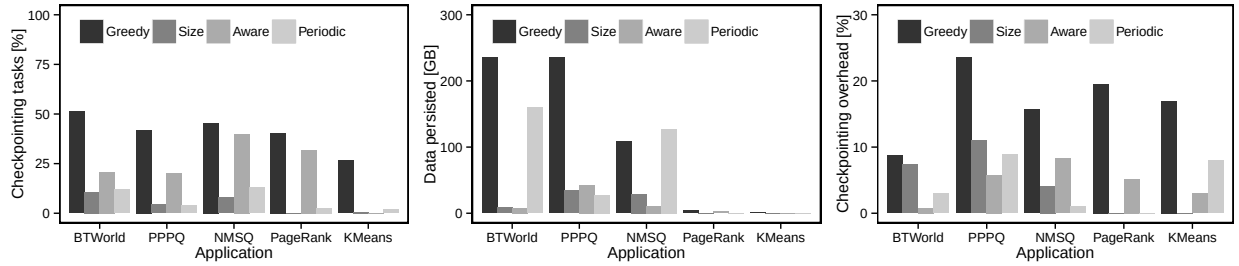
6.2 The Impact of the Checkpointing Overhead

Spark has been widely adopted because it leverages memory-locality, and so it achieves significant speedup relative to Hadoop. Because checkpointing typically trades-off performance for reliability, we want to evaluate how far the performance of PANDA is from the performance of unmodified Spark when it runs on reliable machines. In this section we evaluate the overhead due to checkpointing tasks in PANDA relative to the performance of unmodified Spark without failures and without checkpointing.

Unlike previous approaches to checkpointing that typically save periodically the intermediate state of an application, PANDA reduces the checkpointing problem to a task selection problem. Therefore, we first want to assess how selective our policies are in picking their checkpointing tasks. To this end, in Figure 7a we show the number of tasks that are checkpointed by each policy for all applications. We find that GREEDY is rather aggressive in checkpointing tasks, while SIZE is the most conservative policy. In particular, we observe that with the GREEDY policy, PANDA checkpoints between 26-51% of the running tasks in our applications. Further, because the SIZE policy targets only the outliers, it checkpoints at most 10% of the running tasks for all applications. Similarly to SIZE, our adaptation of the PERIODIC policy checkpoints relatively small fractions of tasks for all applications.

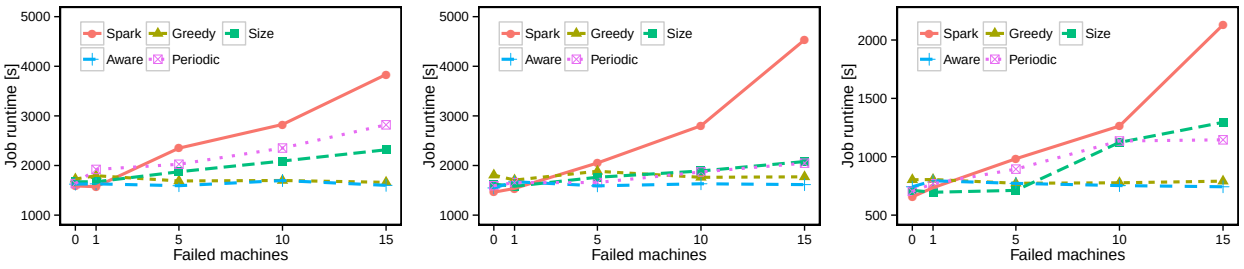
Because AWARE balances the recomputation and the checkpointing costs for each task, the number of checkpointing tasks is variable across different jobs. We observe that for jobs that have relatively small intermediate datasets such as NMSQ and PageRank, AWARE checkpoints roughly 40% of the tasks. However, for BTWORLD and PPPQ, which both generate large amounts of intermediate data, AWARE is more conservative in checkpointing and so it selects roughly 20% of the tasks.

PANDA assumes the presence of an HDFS instance to persist its checkpoints that is permanently available. Running many large applications in a cluster may lead to significant amount of storage space used by PANDA. In Figure 7b we show the amount of data persisted by our policies in each application. We find that GREEDY checkpoints significantly more data than both SIZE and AWARE for all applications. In particular, GREEDY replicates 30 times as much data as SIZE and AWARE with BTWORLD. To perform the checkpoints of all four applications, GREEDY requires a storage space of 1.8 TB (including replicas), whereas SIZE and AWARE require 212 GB and



(a) Checkpointing tasks (b) Checkpointing storage (c) Checkpointing overhead

Figure 7: The fraction of checkpointing tasks (a), the amount of data persisted to HDFS (b), and the checkpointing overhead (c) with all our policies. The baseline is unmodified Spark.



(a) BTWorld (b) PPPQ (c) NMSQ

Figure 8: The job runtime versus the number of failures with all our policies for BTWORLD, PPPQ, and NMSQ. The baseline is Spark with its default lineage-based recomputation.

190 GB, respectively. We also find that despite being rather conservative in selecting its checkpointing tasks, the PERIODIC policy requires a storage space of 1 TB.

Finally, we assess the checkpointing overhead of our policies as a percentage increase in the job runtime relatively to a vanilla version of Apache Spark (see Table 4). Figure 7c shows the checkpointing overhead for all four applications. GREEDY suffers significant performance degradation and its checkpointing overhead may be as high as 20%. However, both SIZE and AWARE incur less than 10% overhead, and so they are very close to the performance of Spark without checkpointing. This result can be explained by what is the main difference between our policies. Whereas GREEDY checkpoints tasks in a best-effort way, both SIZE and AWARE employ more conservative ways of selecting checkpointing tasks based on outliers or cost-benefit analysis. Further, because PERIODIC performs its checkpoints at fixed intervals during the application runtime, the contention on the HDFS is relatively low at all times. As a consequence, although PERIODIC checkpoints significantly more data than both SIZE and AWARE, they all have similar overheads.

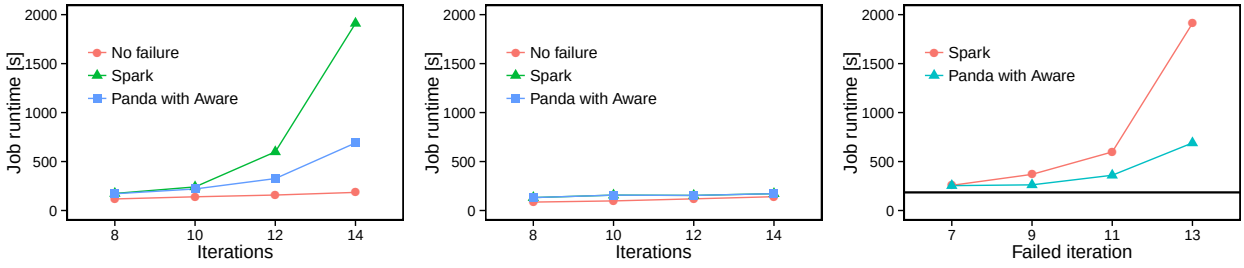
We conclude that both SIZE and AWARE deliver very good performance and they are close to the performance of Spark without checkpointing. These policies are very selective when picking checkpointing tasks, they use relatively small storage space to persist the output data of tasks, and they incur a checkpointing overhead that is usually below 10%.

6.3 The Impact of the Machine Failures

Space-correlated failures, defined as groups of machine failures that occur at the same time across the datacenter, have been frequently reported in large-scale systems such as grids and clusters [17], and more recently in datacenters [16]. Therefore, in this section, we evaluate the performance of PANDA under space-correlated failures. To this end, we report in Figure 8 the job runtime with and without our checkpointing policies for a range of concurrent failures that occur in the last processing stage of our BTWORLD, PPPQ, and NMSQ applications. As a hint of reading this figure, the values at 0 machine failures represent the job runtimes with our policies and with Spark when the job completes without experiencing failures.

Without checkpointing, the recomputation time due to failures causes a significant performance degradation for the entire range of concurrent failures. We observe that the job runtime in unmodified Spark increases linearly with the number of concurrent failures for all applications. For example, when only 25% of the cluster size is lost due to failures, the job runtime increases by 48% for BTWORLD, and by 40% for the two TPC-H queries. For the stress test we consider with 15 out of 20 machines that fail, Spark delivers very poor performance with all applications completing between 2.5 and 3 times as slow as when no failures occur.

Figure 8 also shows that all our checkpointing policies deliver very good performance for the complete range of failures. Both GREEDY and AWARE provide constant runtimes irrespective of the number of failures for all applications. The reason for this result is that they cut-off the lineage graph at key stages, thus avoiding recomputing previously completed work. We also observe that AWARE



(a) PageRank **(b) KMeans** **(c) PageRank**

Figure 9: The job runtime when a single machine fails before the job completes for PageRank and KMeans with different numbers of iterations (a), (b), and the job runtime when a single machine fails in a given iteration for PageRank with 14 iterations (the horizontal line represents the job runtime without failures in unmodified Spark c).

performs slightly better than GREEDY because it introduces a lower checkpointing overhead, as we have shown in Figure 7. SIZE also reduces the impact of failures, but the job runtime still increases linearly with the number of failures. However, SIZE performs at its best and gets very close to the performance of GREEDY and AWARE for jobs such as PPPQ that have outlier tasks with very long durations. In particular, we have shown in Figure 6c that in PPPQ only 10% of the tasks account for more than 60% of the total processing time. Further, we find that PERIODIC has poor performance for BTWORLD, but its performance is very close to the performance of SIZE for the two TPC-H queries.

We conclude that our policies outperform unmodified Spark for any number of space-correlated failures. While both GREEDY and AWARE provide constant job runtimes for the complete range of machine failures, AWARE is our best policy because it introduces a much lower overhead than GREEDY.

6.4 The Impact of the Lineage Length

Although Spark may use the job lineage graph to recover lost RDD partitions after a failure, such recovery may be time-consuming for jobs such as PageRank that have relatively long lineage chains with many wide dependencies (see its DAG in Figure 5d). Conversely, applications such as KMeans, in which narrow dependencies prevail, may be recovered relatively fast from data in stable storage (see its DAG in Figure 5e). In this section we seek to highlight the impact of the lineage graph structure on the job runtime with and without checkpointing in the presence of a single machine failure that occurs at different moments during the job execution.

Figures 9a and 9b show the differences in performance of PANDA with the AWARE policy on the job runtime when a single machine fails before the job completes for different numbers of iterations of PageRank and KMeans. We observe that PageRank completes relatively fast for Spark without failures for the complete range of iterations from 8 to 14. However, the performance of PageRank degrades significantly even when a single failure perturbs the last iteration of the job. In particular, the job runtime is 11 times as large as the fail free execution in Spark for PageRank with 14 iterations. Because PageRank requires many shuffle operations, a machine failure may result in the loss of some fraction of data from each parent RDD, thus requiring a long chain of recomputations. Figure 9a also shows that PANDA with the AWARE policy performs very well and bounds the recomputation time for any number of iterations. Not only does PANDA complete the job four times as fast as the recomputation-based approach in Spark, its performance is also very close to the performance of Spark without failures.

Unlike PageRank, which suffers significantly from failures, we show in Figure 9b that KMeans is rather insensitive to faulty machines and that Spark is less than 5% off the fail-free execution for any number of iterations. The reason for this result can be explained by what is the main difference between the lineage graphs of PageRank and KMeans. As we have shown in Section 5.2, the length of the PageRank lineage graph is proportional to the number of iterations of the job, and so the amount of recomputations triggered by a single failure grows significantly for jobs with many iterations. In contrast with PageRank, KMeans has a much simpler lineage graph, with many narrow dependencies followed by a shuffle, and so its length remains constant irrespective of the number of iterations. As a consequence, checkpointing KMeans is not worthwhile, because these narrow dependencies may be quickly recovered from stable storage. However, because our AWARE policy avoids checkpointing stages that are one hop away from the input dataset, we observe that its operation falls back to default Spark in the case of KMeans.

Finally, Figure 9c shows the results of experiments in which a single machine fails at different moments during the job execution for PageRank with 14 iterations. In general, we observe that Spark performs well when the failure occurs in the early stages of the job. However, it delivers poor performance when the time of failure is closer to the job completion time. We find that PANDA is effective in reducing the recovery time and outperforms Spark irrespective of the time of failure.

We conclude that applications such as PageRank that have many wide dependencies are good candidates for checkpointing, while machine learning applications such as KMeans may be recovered with relatively small recomputation cost. Furthermore, we have shown that PANDA performs very well for lineage graphs in which the recomputation cost is excessive irrespective of the time of failure.

6.5 The Impact of the Failure Pattern

So far, we have evaluated different aspects of the operation of PANDA with single applications that experience space-correlated

Table 5: The distribution of job types in our simulated workload.

Application	Total nodes	Failed nodes	Scale	Runtime [s]	Jobs [%]
BTWORLD	20	5	1.0	1587	16
	20	5	5.0	7935	5
	20	5	10.0	15870	5
PPPQ	20	5	1.0	1461	16
	20	5	5.0	7305	5
	20	5	10.0	14610	5
PageRank (14 iterations)	5	1	1.0	185	16
	5	1	5.0	925	16
	5	1	10.0	1850	16

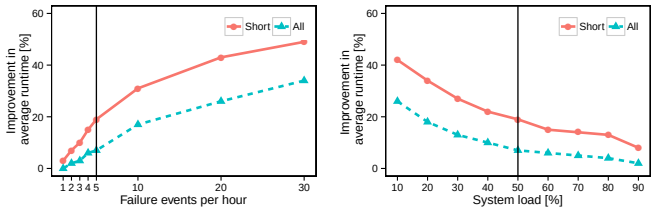
failures at a certain time during their execution. We have shown that our policies deliver very good performance with relatively small checkpointing overhead. However, it is not clear whether the improvements hold for a long-running system when multiple jobs receive service in the cluster only a fraction of which experience failures. Thus, we want to evaluate the improvement in the average job runtime achieved by AWARE for a complete workload relative to unmodified Spark.

We have built our own simulator in order to evaluate the impact of the frequency of failures on the overall improvement of PANDA with the AWARE policy. We simulate the execution of a 3-day workload on a 10,000-machine cluster (similar to the size of the Google cluster discussed in Section 1). We perform simulations at a higher-level than the earlier single-application experiments, and so we use the overall job durations (with or without failures) from experiments rather than simulating the execution of separate tasks.

In our event-based simulator, jobs are submitted according to a Poisson process and they are serviced by a FIFO scheduler. The scheduler allocates to each job a fixed number of machines which are released only after the job completes. Although the Google trace consists of mostly short jobs in the order of minutes, the longest jobs may take hours or even days to complete. To generate a similar realistic workload, we scale up the durations of our jobs by different scaling factors as shown in Table 5. Table 5 also shows the distribution of the job types in our workload. In particular, 80% of the jobs complete within 30 minutes (short jobs), whereas the durations of the remaining jobs exceed 2 h (long jobs).

In order to reuse the results from the experiments, we create the following failure pattern. We assume that failures occur according to a Poisson process that may hit every job at most once. Each failure in our simulation is a space-correlated failure event that triggers the failure of 5 machines. Table 5 shows for every job in our workload the number of failed machines when it is hit by a failure event. In particular, a failure event may hit a single BTWORLD query, a single PPPQ query, or 5 different PageRank jobs. To simulate the execution of a job that is hit by a failure we replace the job runtime given in Table 5 by the job runtime shown in Figure 8 or 9 multiplied by the job's scaling factor. Our assumption here is that both the checkpointing overheads and the improvements in the job runtime achieved by AWARE hold irrespective of the scaling factor of the job. Similarly to the experiments we performed on the DAS, we report averages over three simulations.

Figure 10a shows the results of the simulations for different values of the failure rate under a system load of 50%, which is the average utilization of the Google cluster. PANDA provides significant



(a) Failure rate (b) System load

Figure 10: The improvement achieved by PANDA with the AWARE policy for short jobs and for the complete workload under a system load of 50% versus the failure rate (a), and for a failure rate of 5 versus the system load (b).

gains when machines are more likely to fail but may not be worthwhile when failures occur rarely. Intuitively, jobs are more likely to be hit by failures when the failure rate is high. In particular, the fraction of failed jobs increases from 0.7% to 24% when the failure rate increases from 1 to 30 failure events per hour. We find that PANDA with AWARE reduces the average job runtime with 34% relative to the execution in unmodified Spark when the failure rate is 30 per hour. However, PANDA stops being beneficial when the cluster experiences less than one failure event per hour. Furthermore, Figure 10a shows that the improvement for short jobs is significantly higher than the overall improvement for our workload. For example, for a failure rate of 5, which is equivalent to the maximum failure rate in the Google cluster (see Table 1), short jobs improve by 19% on average, whereas the overall improvement is only 7%.

Finally, in Figure 10b we show the results of the simulations for different values of the system load when 5 failure events are expected every hour. We find that PANDA provides significant improvements over the complete range of system loads, but becomes less beneficial under high loads. The intuition of this result is that failed jobs account for larger fractions of the total number of jobs when the system load is low. In particular, the fraction of failed jobs decreases from 18% to 2.5% when the system load increases from 10% to 90%.

7 RELATED WORK

Checkpointing has traditionally been very important in high-performance computing (HPC) systems, but has lately also received quite some attention for data analytics frameworks.

BlobCR [14] seeks to efficiently capture and roll-back the state of scientific HPC applications in public clouds. Recent work [9] analyzed practical methods for optimizing the checkpointing interval using real-world failure logs. Multi-level checkpointing [6] aims at reducing the overhead of checkpointing in large-scale platforms by setting different levels of checkpoints each of which has its own overhead and recovery capability. An adaptive checkpointing scheme with work migration [24] has been developed to minimize the cost of running applications on resources from spot markets. Similar techniques that aim to reduce the checkpointing overhead of the naive periodic checkpointing policy exploit the temporal locality in failures [19].

Closest to our work, TR-Spark [23] and Flint [18] propose checkpointing policies for data analytics applications that run on transient resources which are typically instable, but not necessarily

due to faults. In particular, TR-Spark employs cycle-scavenging to leverage such transient resources which are kept idle as a resource buffer by cloud providers and may be revoked due to load spikes. TR-Spark takes a statistical approach to prioritize tasks that have a high probability of being completed before the resources where they run are revoked and to checkpoint data blocks that are likely to be lost before they are processed by the next processing stages. To do so, TR-Spark requires both the distribution of the task runtimes and the distribution of the inter-arrival failure time for each resource allocated to the framework. Similarly, Flint provisions instances available on the spot market which have relatively low prices and may be revoked due to price spikes. Flint supports RDD-level checkpointing using an adaptation of the periodic checkpointing policy to data analytics applications. In contrast, PANDA targets a datacenter environment where applications may suffer from outright node failures and proposes a more comprehensive set of task-level checkpointing policies that take into account not only the lineage structure of the applications, but also workload properties such as the task runtimes and the intermediate data sizes.

8 CONCLUSIONS

The wide adoption of in-memory data analytics frameworks is motivated by their ability to process large datasets efficiently while sharing data across computations at memory speed. However, failures in datacenters may cause long recomputations that degrade the performance of jobs executed by such frameworks. In this paper we have presented PANDA, a checkpointing system for improving the resilience of in-memory data analytics frameworks that reduces the checkpointing problem to a task selection problem. We have designed three checkpointing policies starting from first principles, using the size of the task output data (GREEDY), the distribution of the task runtimes (SIZE), or both (AWARE). The GREEDY policy employs a best-effort strategy by selecting as many tasks for checkpointing as a predefined budget allows. The SIZE policy checkpoints only straggler tasks that run much slower than other tasks of the job. The AWARE policy checkpoints a task only if the cost of recomputing it exceeds the time needed to persist its output to stable storage.

With a set of experiments on a multicluster system, we have analyzed and compared these policies when applied to single failing applications. We have found that our policies outperform both unmodified Spark and the standard periodic checkpointing approach. We have also analyzed the performance of PANDA with the AWARE policy by means of simulations using a complete workload and the failure rates from a production cluster at Google. We have found that PANDA is beneficial for a long-running system and can significantly reduce the average job runtime relative to unmodified Spark. In particular, the SIZE policy delivers good performance when the failure rate (fraction of failed machines per day) is relatively low (less than 6%). Although both GREEDY and AWARE turn out to provide significant improvements for a large range of failure rates (more than 6%), AWARE is our best policy because it introduces a much lower overhead than GREEDY. However, when the datacenter is prone to failures of complete racks, the rather aggressive checkpointing strategy of the GREEDY policy may be worthwhile.

9 ACKNOWLEDGMENT

This research was supported by the Dutch national program COMMIT.

REFERENCES

[1] Ganesh Ananthanarayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. 2012. True Elasticity in Multi-tenant Data-intensive Compute Clusters. *ACM SoCC* (2012).

[2] TPC Benchmarks. 2016. http://www.tpc.org. (2016).

[3] CERN. 2016. http://wlcg-public.web.cern.ch/. (2016).

[4] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. 2011. The Case for Evaluating MapReduce Performance using Workload Suites. *IEEE Mascots* (2011).

[5] Jeff Dean. 2009. Designs, Lessons and Advice from Building Large Distributed Systems. *Keynote from LADIS* (2009).

[6] Sheng Di, Yves Robert, Frédéric Vivien, and Franck Cappello. 2017. Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model. *IEEE TPDS* 28, 1 (2017).

[7] Distributed ASCI Supercomputer. 2015. http://www.cs.vu.nl/das5. (2015).

[8] Catalin L Dumitrescu, Ioan Raicu, and Ian Foster. 2005. Experiences in Running Workloads over Grid3. *Grid and Cooperative Computing* (2005).

[9] Nosayba El-Sayed and Bianca Schroeder. 2014. Checkpoint/Restart in Practice: When Simple is Better. *IEEE Cluster* (2014).

[10] Tim Hegeman, Bogdan Ghit, Mihai Capota, Jan Hidders, Dick Epema, and Alexandru Iosup. 2013. The BTWorld Use Case for Big Data Analytics: Description, MapReduce Logical Workflow, and Empirical Evaluation. *IEEE Big Data* (2013).

[11] Michael Isard. 2007. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review* 41, 2 (2007).

[12] Bahman Javadi, Derrick Kondo, Alexandru Iosup, and Dick Epema. 2013. The Failure Trace Archive: Enabling the Comparison of Failure Measurements and Models of Distributed Systems. *Elsevier JPDC* 73, 8 (2013).

[13] Zhaolei Liu and TS Eugene Ng. 2017. Leaky Buffer: A Novel Abstraction for Relieving Memory Pressure from Cluster Data Processing Frameworks. *IEEE TPDS* 28, 1 (2017).

[14] Bogdan Nicolae and Franck Cappello. 2011. BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots. *ACM Supercomputing* (2011).

[15] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and VMware ICSI. 2015. Making Sense of Performance in Data Analytics Frameworks. *USENIX NSDI* (2015).

[16] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. *ACM SoCC* (2012).

[17] Bianca Schroeder and Garth Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE TDSC* 7, 4 (2010).

[18] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. 2016. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. *ACM EuroSys* (2016).

[19] Devesh Tiwari, Saurabh Gupta, and Sudharshan S Vazhkudai. 2014. Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems. *IEEE/IFIP DSN* (2014).

[20] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing Cloud Computing Hardware Reliability. *ACM SoCC* (2010).

[21] Maciej Wojciechowski, Mihai Capotă, Johan Pouwelse, and Alexandru Iosup. 2010. BTWorld: Towards Observing the Global BitTorrent File-Sharing Network. *ACM HPDC* (2010).

[22] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. MEMTUNE: Dynamic Memory Management for In-memory Data Analytic Platforms. *IEEE IPDPS* (2016).

[23] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. *ACM SoCC* (2016).

[24] Sangho Yi, Artur Andrzejak, and Derrick Kondo. 2012. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE TSC* 5, 4 (2012).

[25] John W Young. 1974. A First Order Approximation to the Optimum Checkpoint Interval. *Comm. of the ACM* 17, 9 (1974).

[26] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, and others. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Comm. of the ACM* 59, 11 (2016).