ATLAS: An Adaptive Failure-Aware Scheduler for Hadoop

Mbarka Soualhia[1] Foutse Khomh[2] Sofiène Tahar[1]
[1]Concordia University, [2]Polytechnique Montréal, Montréal, Québec, Canada
{soualhia, tahar}@ece.concordia.ca, foutse.khomh@polymtl.ca

***Abstract*—Hadoop has become the *de facto* standard for processing large data in today's cloud environment. The performance of Hadoop in the cloud has a direct impact on many important applications ranging from web analytic, web indexing, image and document processing to high-performance scientific computing. However, because of the scale, complexity and dynamic nature of the cloud, failures are common and these failures often impact the performance of jobs running in Hadoop. Although Hadoop possesses built-in failure detection and recovery mechanisms, several scheduled jobs still fail because of unforeseen events in the cloud environment. A single task failure can cause the failure of the whole job and unpredictable job running times. In this paper, we propose ATLAS (AdapTive faiLure-Aware Scheduler), a new scheduler for Hadoop that can adapt its scheduling decisions to events occurring in the cloud environment. Using statistical models, ATLAS predicts task failures and adjusts its scheduling decisions on the fly to reduce task failure occurrences. We implement ATLAS in the Hadoop framework of Amazon Elastic MapReduce (EMR) and perform a case study to compare its performance with those of the FIFO, Fair and Capacity schedulers. Results show that ATLAS can reduce the percentage of failed jobs by up to 28% and the percentage of failed tasks by up to 39%, and the total execution time of jobs by 10 minutes on average. ATLAS also reduces CPU and memory usages.**

***Keywords*-Failure Prediction, Scheduler, Cloud, Hadoop, Amazon Elastic MapReduce**

I. Introduction

MapReduce [1] has emerged as the leading programming model for large-scale distributed data processing. Hadoop [2], the open-source implementation of MapReduce has become the framework of choice on many off-the-shelf clusters in the cloud. It is extensively used in many applications ranging from web analytic, web indexing, image and document processing to high-performance scientific computing and social network analysis. Major large companies like Google, Facebook, Yahoo or Amazon rely daily on Hadoop to perform important data-intensive operations in their data centers. However, because of the scale, complexity and the dynamic nature of cloud environments, failures are common in data centers powering the cloud. Studies [3] show that more than one thousand individual machine failures and thousands of hard-drive failures can occur in a cluster during its first year of service. Several power problems can also happen bringing down between 500 and 1000 machines for up to 6 hours. The recovery time of these failed machines being as high as 2 days. These frequent failures in data centers have a significant impact on the performance of applications running Hadoop [3]. Dinu et al. [3] who examined the performance of Hadoop under failures reported that many task failures occur because of a lack of sharing of failure information between the different components of the Hadoop framework. The Hadoop scheduler is a centrepiece of the Hadoop framework. An effective Hadoop scheduler can avoid submitting tasks on fault-prone machines; which would reduce the impact of machine failures on the performance of the applications running Hadoop. However, basic Hadoop scheduling algorithms like the FIFO algorithm, the Fair-sharing algorithm, and the Capacity algorithm only rely on a small amount of system information to make their scheduling decisions. They are not equipped with pro-active failure handling mechanisms. Yet, a single task failure can cause the failure of a whole job and unpredictable job running times. In our previous work [4] we have shown that it is possible to predict task and job scheduling failures in a cloud environment and that such predictions can reduce the percentage of failed jobs by up to 45%. But, we did not propose an efficient strategy to reschedule tasks predicted as failed. In this paper, we build on that previous work and propose ATLAS (AdapTive faiLure-Aware Scheduler), a new scheduler for Hadoop that adapts its scheduling decisions to events occurring in the cloud environment. Using information about events occurring in the cloud environment (*e.g.*, resource depletion on a node of the cluster or failure of a scheduled task) and statistical models, ATLAS predicts the potential outcome of new tasks and adjusts its scheduling decisions accordingly to prevent them from failing. In addition, ATLAS scheduler introduces novel strategies to reschedule tasks predicted as failed like multiple speculative executions and penalty mechanism. We implement ATLAS in the Hadoop framework of Amazon Elastic MapReduce (EMR). *To the best of our knowledge, ATLAS is the first scheduler for Hadoop that adapts its scheduling decisions based on predicted failures information*. We perform a case study using multiple single and chained Hadoop jobs (these jobs are composed of *WordCount*, *TeraGen* and *TeraSort* job units, to compare the performance of ATLAS with those of the FIFO, Fair and Capacity schedulers. To assess the performance of each scheduler, we compute the total execution times of jobs, the amount of resources used (CPU, memory, HDFS Read/Write), the numbers of finished and failed tasks and the numbers of finished and failed jobs. Using these information, we answer the following research question.

RQ: Does ATLAS outperforms FIFO, Fair, and Capacity schedulers in terms of execution time, number of finished and failed tasks, number of finished and failed jobs, and resource usage? Results show that ATLAS can reduce the percentage of failed jobs by up to 28%, the percentage of failed tasks by up to 39%. Although ATLAS requires training

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

a predictive model, we found that the reduction in the number of failures largely compensates for the model training time. In fact, ATLAS even reduces the total execution time of jobs by 10 minutes on average. ATLAS also reduces CPU and memory usages, as well as the number of HDFS Reads and writes.

The remainder of this paper is organized as follows: Section II presents background information about Hadoop. Section III presents the motivation for this work. Section IV describes our proposed scheduler (*i.e.*, ATLAS). Section V describes our case study and discusses the obtained results. Section VI discusses threats to the validity of our work. Section VII summarizes the related literature, and Section VIII concludes the paper and outlines some avenues for future works.

II. Background

A. *MapReduce-Hadoop*

MapReduce is a programming model designed to perform parallel processing of large datasets using a large number of computers (nodes) [1]. It splits jobs into parallel sub-jobs to be executed on different processing nodes where the data are located instead of sending the data to where the jobs will be executed. A MapReduce job is composed of map and reduce functions and the input data. The map function subdivides the input record into a set of intermediate <key, value> pairs. The input data which are called splits represent a set of distributed files that will be assigned to the mappers. The reduce function takes a set of values to process for a same key and generates the output for this key. MapReduce requires a master (known as "JobTracker") that controls the execution procedure across the mappers (*i.e.*, worker running a map function) and the reducers (*i.e.*, worker running a reduce function), using "TaskTrackers", to ensure that all the functions are executed and have their input data.

Hadoop [2] is a Java-based open source implementation of MapReduce proposed by Cutting and Cafarella in 2005. It has become the de facto standard for processing large data in today's cloud environment. Hadoop is composed of two main units: a storage unit (Hadoop Distributed File System (HDFS)) and a processing unit (MapReduce). It has a Master-Slave architecture: the master node consists of a JobTracker and NameNode. A slave (or worker) node can act as both a DataNode and TaskTracker. Hadoop hides all system-level details related to the processing of parallel jobs (such as the distribution to HDFS file store or error handling), allowing developers to write and enhance their parallel programs while focusing only on the computations issues rather than the parallelism ones.

B. *Hadoop Schedulers*

The default scheduling algorithm of Hadoop is based on the First In First Out (FIFO) principle. Facebook and Yahoo! have developed two new schedulers for Hadoop: Fair Scheduler and Capacity scheduler, respectively. The default scheduler of Hadoop uses a FIFO queue [5]. The received jobs are partitioned into sub-tasks which will be loaded into the queue and executed in the order in which they are submitted, regardless of the type and size of the jobs. Although, the FIFO algorithm is easy to implement and grants a full access to resources to the scheduled jobs, starvation is possible as a scheduled job can use the entire resource of the cluster for a long time, while others wait in the queue until they time out. The Fair Scheduler [5] was developed by Facebook to ensure that resources are assigned fairly among different jobs so that all users get on average the required resources available in the cluster over the time. It supports multi-user execution in one cluster unlike the default FIFO scheduler of Hadoop. Moreover, it can use the priorities assigned to users applications as a factor to determine the required resources. Therefore, it can guarantee that long jobs will not be starving, by optimizing their waiting time in the queue. The Capacity scheduler [5] was originally implemented by Yahoo!. It supports multi-user execution within one cluster and allows large number of users to execute their jobs fairly over time. In fact, it divides the cluster into multiple queues with configurable capacity (*i.e.*, CPU, memory, disk, etc.). The queues support jobs priorities and guarantee that there is a limit on the allocated resources to all users in order to prevent some jobs from using all available resources in the queue where they are assigned.

III. Motivation

A. *Limitation of Current Hadoop's Implementation*

Dinu *et al.* analysed the behavior of the Hadoop framework under different types of failure and found that TaskTracker and DataNode failures are very important since they affect the availability of jobs input and output data [3]. In addition, these failures can cause important delays during the execution of HDFS read and write procedures. Their experiments showed that a single failure can lead to unpredictable execution time; for example the average execution time of a job, which is 220s, can reach 1000s under a TaskTracker failure and 700s under a DataNode failure. Moreover, they claimed that the recovery time of the failed components (such as TaskTracker or DataNode) in Hadoop can be long and can cause jobs delays which may affect the overall performance of a cluster. As an illustration, let's consider a TaskTracker that sends heartbeats to the JobTracker every 10 minutes (this is the default value), if a failure occurs in the TaskTarcker within the first minute after a communication with the JobTracker, all the tasks assigned to this TaskTracker will fail and the JobTracker will notice these failures only after about 9 minutes, resulting in a delay in the rescheduling of the failed tasks and an increase of the execution time of the job. Even if tasks are speculatively executed to prevent their full rescheduling in the event of a failure, there is still a cost associated with this replication (the resource spent on the speculative executions) [3].

In addition, DataNode failures have a large impact on the start up time of speculative task executions. This is because of the statistical nature of the speculative execution algorithm which is based on data about task progress. In fact, if a task was making good progress and suddenly fails because of a DataNode failure, its speculative execution will start with a delay (*i.e.*, later than speculative executions of straggler tasks), since Hadoop expected a normal behavior from that task. Also, many map and reduce tasks may fail because they exceeded the number of failed attempts allowed by the TaskTracker. For simplicity and scalability, each computing node in Hadoop manages failure detection and recovery on its own and hence the launched tasks can not share failure information between them. Therefore, multiple tasks, including the speculative tasks, may fail because of an error already encountered by a previous task [3].

B. *Problem Formulation*

Let's consider *N* jobs submitted to Hadoop, where each job is composed of *X* map tasks and *Y* reduce tasks. Let's assume that each job is using *R(CPU, Memory, HDFS Read/Write)* resources from *M* machines in an Hadoop cluster. Each map/reduce task is allowed a maximum number of scheduling attempts: each new task is assigned to a node and if it fails it can be rescheduled multiple times either on the same node or on another available node. When a task exceeds its maximum number of scheduling attempts allowed by the TaskTracker, the task is considered to be failed, otherwise it is finished successfully. Because of the dependency between map and reduce tasks, if either a map or a reduce task fails, the whole job to which the task belongs will fail as well, even if all the other tasks in the job were completed successfully. For example, in Figure 1, *Job3* failed because one of its map tasks failed (because it exceeded its maximum number of scheduling attempts). As a consequence of the failure of this map task, all reduce tasks were failed automatically.

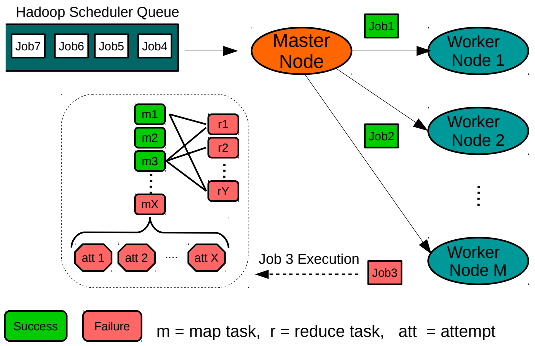

Fig. 1: Example of Hadoop Job Failure

More formally, if $S(job)$ is the outcome of an executed job; $S(MapAtt_{ip})$ the status of a map_i after the p^{th} attempt (we give a value of 1 when an attempt is successful and 0 otherwise) and $S(ReduceAtt_{jq})$ the status of $reduce_j$ after the q^{th} attempt. If *K* and *L* are the maximum numbers of scheduling attempts allowed for map and reduce tasks respectively:

$$S(job) = [\prod_{i=1}^{X}(\sum_{p=1}^{K} S(MapAtt_{ip}))] * [\prod_{j=1}^{Y}(\sum_{q=1}^{L} S(ReduceAtt_{jq}))] \qquad (1)$$

Given that the execution time of a task is the sum of execution times of all its launched attempts (both the finished and the failed attempts), the more a task experience failed attempts, the longer its execution time will be. This delay in the execution of the tasks will also translate into longer execution times for the job (to which the tasks belong) and larger resources usages. More specifically, if *T(job)* is the total execution time of a job composed of a set of $A = \{map_i\}_{i \in X}$ map tasks and a set of $B = \{reduce_j\}_{j \in Y}$ reduce tasks. If $T(MapAtt_{ip})$ and $T(ReduceAtt_{jq})$ are respectively the execution times of the map_i and $reduce_j$ tasks during the attempts p, q respectively:

$$T(job) = Max_A(\sum_{p=1}^{K} T(MapAtt_{ip})) + Max_B(\sum_{q=1}^{L} T(ReduceAtt_{jq})) \qquad (2)$$

Therefore, we believe that it is very important to reduce the number of tasks failed attempts if we want to improve the overall performance of an Hadoop cluster. By reducing the number of tasks failed attempts, one will reduce the turnaround time of jobs running in the cluster. If one can predict task failure occurrences and adjust scheduling decisions accordingly to prevent failures from occurring, one may be able to reduce the number of tasks failed attempts. In our previous work [4] we have shown that it is possible to achieve such predictions. In the following Section IV, we build on our previous work and propose ATLAS, a scheduling algorithm that adapts its scheduling decisions based on predicted failures information.

IV. ATLAS: An Adaptive Failure-aware Scheduler

A. *Proposed Methodology*

In this section, we present our scheduler ATLAS, which can reduce the number of tasks failed attempts by predicting task scheduling outcomes and adjusting scheduling decisions to prevent failure occurrences. We describe the approach followed to analyse Hadoop's log files and build task failure predictive models. Figure 2 presents an overview of this approach. First, we run different jobs on Hadoop cluster in order to get trace of data about previously executed tasks and jobs. Next, we analyse log files obtained from Amazon EMR Hadoop Clusters and extract jobs and tasks main attributes. Next, we analyse correlations between tasks attributes and tasks scheduling outcomes. Finally we apply statistical predictive learning techniques to build task failures prediction models. The remainder of this section elaborates more on each of these steps.

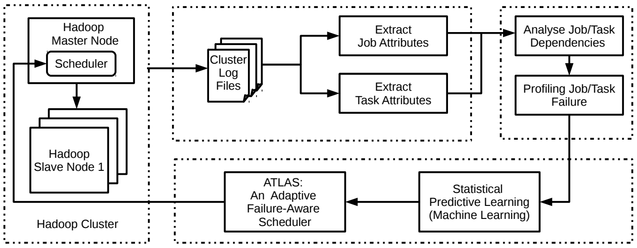
Hadoop Master Node
Scheduler
Hadoop Slave Node 1
Hadoop Cluster
Cluster Log Files
Extract Job Attributes
Extract Task Attributes
Analyse Job/Task Dependencies
Profiling Job/Task Failure
ATLAS: An Adaptive Failure-Aware Scheduler
Statistical Predictive Learning (Machine Learning)

Fig. 2: Overview of Our Proposed Methodology

1) Extraction of Tasks/Jobs Attributes: First, we run different workload in parallel including single and chained jobs on Amazon EMR Hadoop clusters. We run different single jobs in parallel such as *WordCount*, *TeraGen*, *TeraSort* to get different workload on several machines. In addition, we run chained jobs (sequential, parallel and mix chains) composed of *WordCount*, *TeraGen*, *TeraSort* jobs to get different types of job running on the cluster. Also, we vary the size of the used jobs (number of map and reduce tasks, number of jobs in a chained job). These jobs represent different job pattern similar to the ones running in real world applications. We implemented a bash script to extract job/task attributes. So, for each job we extracted: *job ID, priority, execution time, number of map/reduce, number of local map/reduce tasks, number of finished/failed map/reduce tasks and the final status of the job*. For each task we extracted the following information: *job ID, task ID, priority, type, execution time, locality, execution type, number of previous finished/failed attempts of the task, number of reschedule events, number of previous finished/failed tasks, number of running/finished/failed tasks running on the TaskTracker, the amount of used resources (CPU, Memory and HDFS Read/Write) and the final status of the task*. More details about the job and task attributes can be found in [6].

2) Profiling of Tasks/Jobs Failure: To identify the correlation between job/task scheduling outcome and their attributes, we analyse the dependencies between the jobs and tasks and perform a mapping between the failed ones and their attributes. Second, we checked the obtained data to identify the most relevant attributes that impact the final scheduling outcome of task/job by removing the ones having unchanged value or null value. This step is preliminary to the following one.

3) Statistical Predictive Learning: We aim to explore the possibility to predict a potential task failure in advance based on its collected attributes and machine learning techniques. We believe that if we can share the failure information between tasks in advance, we can prevent the occurrence of the predicted failure and reschedule them on appropriate clusters to ensure their timely and successful completion. To do so, we choose several regression and classification algorithms in *R* [7] to build models: GLM (General Linear Model), Random Forest, Neural Network, Boost, Tree and CTree (Conditional Tree). More details about these algorithms can be found in [6]. We use different training and testing data set for both jobs and tasks. We collected data related to 70,000 jobs and 180,000 tasks from the Hadoop cluster we used in our experiments as described in Section IV-A1. The log data were collected over a fixed period of time of 10 minutes. The training time is related to the steps of training process and not to the complexity of the running jobs. We apply 10-fold random cross validation to measure the accuracy, the precision, the recall and the error of the prediction models [7]. In the cross validation, each data set is randomly split into ten folds. Nine folds are used as the training set, and the remaining fold is used as the testing set.

B. The ATLAS Scheduling Algorithm

ATLAS aims to provide better scheduling decisions for predicted failed tasks, in order to ensure their successful execution. A scheduling decision may require either assigning the tasks to other TaskTrackers with enough resources or waiting for some other tasks to be finished. We designed ATLAS using the predictive model that provided the best results in terms of precision and accuracy when predicting tasks scheduling outcomes. ATLAS integrates with any Hadoop's base scheduler (like FIFO, Fair, Capacity, etc). In fact, when tasks are predicted to succeed, ATLAS relies on Hadoop's base scheduler to make its scheduling decision. Algorithm 1 presents ATLAS in details. The main algorithm is composed of 3 main parts: (1) a task failure prediction algorithm, (2) an algorithm to check the availability of resources, and (3) a task rescheduling algorithm (for potential failed tasks). We implemented a procedure to collect the attributes of the tasks (map/reduce) as described in [6]. The attributes of the tasks represent the predictors of our models. Using the values of these attributes, the failure prediction algorithm predicts whether a task will be finished or failed if executed. The response of the trained models is a binary variable taking *"True"* if a scheduled task succeeds and *"False"* if it fails. We implemented two different prediction algorithm for the mappers and the reducers since they have different input parameters. Next, if a task is predicted to succeed, our algorithm checks the availability of the TaskTracker and DataNode to verify if they are activated or not, since we noticed that the scheduler may assign tasks to a dead TaskTracker, because of the predetermined frequency of heartbeats between the JobTracker and TaskTracker (between two heartbeats a JobTracker has no means to know that a TaskTracker is dead).

Moreover, we implemented a procedure to modify the time spent between two successive heartbeats based on collected information about TaskTracker failures. This procedure is running in parallel along with ATLAS. So, if there are very frequent TaskTracker failures (*i.e.*, more than 1/3 of TaskTrackers were failed between two heartbeats), the time between two heartbeats will be decreased in order to detect faster node failures and reschedule tasks early on, on other alive nodes. This time is decreased each time by half of the previous time elapsed between two heartbeats (*i.e.*, the value was 10 min, then it will be decreased to 5 min) until reaching a minimum value. The minimum value in our experiment is 2 min. If there are less TaskTracker failures (*i.e.*, less than 1/3 of the workers), this time will be increased in order to reduce the cost associated with communication between the JobTracker and TaskTracker. The value of heartbeat is adjusted on the fly according to events related to TaskTracker failures.
After checking the TaskTracker, ATLAS checks if there are enough slots on the selected TaskTracker or not since some tasks may fail because of a high number of concurrent tasks on a TaskTracker. If an assigned task is predicted to fail but there are enough available resources in the cluster, ATLAS will launch the task speculatively on many nodes (specifically the ones that are not very distant) that have enough resources, in order to speed up the execution of the task and increase the chances of success of the task. All the decisions made by the ATLAS scheduler are controlled by a time-out metric

from Hadoop's base scheduler. Hence, if a task reaches its time-out, its associated attempt will be considered as failed and the task will be rescheduled again but with a low priority. We rely on a penalty mechanism to manage the priority of the tasks. We assign a penalty to tasks causing delays to other tasks and tasks that are predicted to fail multiple times. This penalty reduces their execution priority, causing them to wait in the queue until enough resources are available to enable their speculative execution on multiple nodes. In Algorithm 1, we denote *JobTracker* as *JT*, *TaskTracker* as *TT* and *DataNode* as *DN*.

Algorithm 1 : The ATLAS Scheduling Algorithm

```
if (TypeofTask(Task) == "Map") then
    /* Collect attributes of Task as described in [6] */
    Attributes = Collect-Attributes-Map(Task)
    /* Learning Algorithm will predict if Task will be finished/failed */
    Predicted-Status = Predict-Map(Task, Attributes)
else
    Attributes = Collect-Attributes-Reduce(Task)
    Predicted-Status = Predict-Reduce(Task, Attributes)
end if
if (Predicted-Status == "SUCCESS") then
    /* Check whether the TT and DN are dead or alive */
    Check-Availability(TT,DN)
    if (TT and DN are available) then
        /* Test if the TT have enough slots to serve Task */
        Check-Availability-Slots(Task,TT)
        if (Slots are available in TT) then
            Execute(Task,TT)
        else
            Wait Until Free Slots in TT and Time-Out Not Reached
            if (Time-Out is Reached ) then
                /* Resubmit Task since it will fail in such conditions */
                Send to Queue + Penalty
            else
                /* Execute Task in the TaskTracker TT */
                Execute(Task,TT)
            end if
        end if
    else
        while (TT/DN not activated and Time-Out Not Reached ) do
            /* Send a HeartBeat to the JT to activate TT/DN */
            Notify JT to Activate TT/DN
        end while
        if (Time-Out is reached) then
            Send to Queue + Penalty
        else
            Execute(Task,TT)
        end if
    end if
else
    if (There are Enough Resources on Nodes) then
        /* Launch Many Speculative Instance of Task to increase the probability of
its success/
        Execute-Speculatively(Task,N)
    end if
end if
```

V. Evaluation

In this section we presents the design of our case study aimed at assessing the effectiveness of the ATLAS scheduler.

A. Setup of the Case Study

We instantiated 15 Hadoop machines in Amazon EMR. We set one machine to the role of master, another one to the role of secondary master (to replace the master in case of failure) and 13 machines to the role of slaves. We selected 3 different types of machines to have heterogeneous environment which represents a real world environment hosting real and different machines. The three types of Amazon EMR machines are *m3.large*, *m4.xlarge* and *c4.xlarge* [8]. Details about their characteristics are listed in Table II. We choose these types of machine since they can support different workloads and they were widely used in the literature to test many systems in cloud environment. In addition, different types of Amazon EMR instance allow to have a real world cluster where different types of machines are used. We used the AnarchyApe tool described in [9] to create different failure scenarios in Hadoop nodes such as TaskTarcker and DataNode failures, slowdown or drop in the network, tasks/jobs failure etc. as described in [6]. To specify the amount of failures to be injected in the Hadoop clusters, we performed a quantitative analysis of failures in the public Google Traces [4]. We found that more than 40% of the tasks and jobs can be failed [4]. Therefore, in our case study, we performed different simulations of varying the injected failure rates, with a maximum failure rate of 40%.

Machine Type	vCPU*	Memory (GiB)	Storage (GB)	Network Performance
m3.large	1	3.75	4	Moderate
m4.xlarge	2	8	EBS-Only+	High
c4.xlarge	4	7.5	EBS-Only+	High

* Each vCPU is a hyperthread of an Intel Xeon core [8].

+ Amazon EBS is a block-level storage volume for an EC2 instance [8].

TABLE II: Amazon EC2 Instance Specifications

To generate the jobs, we used the *WordCount*, *TeraGen* and *TeraSort* examples provided by Apache, as job unit to create single or chained jobs as described in Section IV-A1. The generated jobs represent different job pattern from real world applications. These jobs have different input files that we downloaded to have large set of data to process. For each single job, we decided on the number of map and reduce tasks using information about the number of HDFS blocks in the input files. For example, we had jobs with 10 map tasks and 15 reduce tasks. For each chained job, we decided on the number of job unit within each chain (*e.g.*, 3 jobs, 20 jobs), the type of used job (*e.g.*, *WordCount*, *TeraSort*) and the structure of the jobs (*e.g.*, sequential, parallel and mix chains). For each type of Hadoop scheduler (*i.e.*, FIFO, Fair, Capacity), we generated several single and chained jobs and collected their execution logs to build the task failure prediction model required by ATLAS. We built prediction models using all the classification algorithms described in Section IV-A3 and assessed their performances following the 10-fold cross-validation approach described in Section IV-A3. Also, we retrained the prediction models in the instantiated Amazon EMR machines which represent a cloud environment where drastic changes may occur because of unreliable machines. This step was performed each 10 min to make the proposed system more robust. We implemented ATLAS using the prediction model that achieved the best performance, and compared the performance of Hadoop equipped respectively with the FIFO scheduler, the Fair scheduler, the capacity scheduler, and our proposed ATLAS scheduler integrated with these 3 existing schedulers. All the comparisons were done using the exact same jobs and data. We measured the performance each Hadoop's scheduler

Task	Scheduler	FIFO					Fair					Capacity				
Map Task	**Algorithm**	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**
	Tree	68.6	85.3	74.3	31.4	12.34	92.6	85.2	62.1	7.4	9.14	68.7	85.7	73.4	31.3	58.18
	Boost	75.9	86.7	78.5	24.1	180.51	65.9	85.6	73.5	34.1	199.80	72.5	85.3	71.3	27.5	280.56
	Glm	63.5	87.2	72.4	36.5	9.43	67.4	88.6	65.8	32.6	13.34	62.1	83.6	61.8	37.9	16.01
	CTree	65.9	87.5	65.2	34.1	15.61	62.6	85.2	69.3	37.4	16.38	61.8	79.5	61.2	38.2	17.53
	Random Forest	**83.7**	**86.4**	**94.3**	**16.3**	**23.53**	**79.8**	**83.9**	**94.0**	**20.2**	**25.91**	**78.3**	**85.2**	**89.2**	**21.7**	**25.97**
	Neural Network	65.8	85.8	79.4	34.2	61.81	68.7	86.3	74.1	31.3	63.61	72.1	83.6	72.3	27.9	59.71
Reduce Task	**Algorithm**	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**	**Acc.**	**Pre.**	**Rec.**	**Err.**	**Time**
	Tree	72.4	95.3	69.3	27.6	13.51	72.8	92.2	73.0	27.8	10.23	63.2	83.4	68.7	36.8	13.25
	Boost	83.5	93.4	85.1	16.5	297.29	93.1	98.7	92.3	6.9	269.37	66.8	91.7	54.7	33.2	198.37
	Glm	61.9	92.6	65.1	38.1	17.32	77.2	91.3	75.3	22.8	17.39	68.7	91.9	71.9	31.3	19.83
	CTree	79.3	92.6	81.4	20.7	16.85	81.4	91.1	81.3	18.6	16.52	61.7	91.5	65.1	38.3	17.13
	Random Forest	**95.3**	**98.1**	**95.9**	**4.7**	**35.65**	**92.5**	**97.6**	**92.4**	**7.5**	**28.54**	**83.4**	**91.5**	**95.6**	**16.6**	**27.93**
	Neural Network	74.5	91.5	75.3	25.5	89.74	81.5	97.3	71.6	18.5	78.61	74.9	94.1	83.7	25.1	85.37

TABLE I: Accuracy, Precision, Recall, Error (%) and Time(ms) for different Algorithms: (10-fold Cross-validation)

using the total execution times of jobs, the amount of resources used (CPU, memory, HDFS Read/Write), the numbers of finished and failed tasks, and the numbers of finished and failed jobs.

B. Case Study Results

1) Prediction Algorithms: Table I summarises the performance of the six prediction models applied on data collected from the schedulers' logs. This result shows that the Random Forest algorithm achieves the best precision, recall, accuracy and error when predicting the scheduling outcome of the Map and Reduce tasks for the three studied schedulers (FIFO, Fair and Capacity). This is because Random-Forest algorithm uses the majority voting on decision trees to generate results which makes it robust to noise, resulting usually in highly accurate predictions. For map tasks, a Random Forest model can achieve an accuracy up to 83.7%, a precision up to 86.4%, a recall up to 94.3% and an error up to 21.7%. The total execution time of the 10-fold cross-validation was 25.97 ms. For reduce tasks, the Random Forest model achieved an accuracy up to 95.3%, a precision up to 98.1%, a recall up to 95.9% and an error up to 16.6%. The total execution time of the evaluation of Random Forest for reduce tasks was 35.65 ms. Also, we noticed that Random Forest is achieving these results in an acceptable time compared to the other algorithms (for example, the Boost algorithm can take up to 297.29 ms which can affect the performance of the scheduler). In addition, we also found a strong correlation between the number of running/finished/failed tasks on a TaskTracker, the locality of the tasks, the number of previous finished/failed attempts of a task, and the scheduling outcome of the task. More specifically, tasks characterized by multiple past failed attempts, many concurrent tasks (running on the same TaskTracker) that experienced multiple failed attempts, have a high probability to fail in the future.

2) Performance Evaluation of the Schedulers: As stated in Section V-B1, we used the Random Forest algorithm (to predict the scheduling outcome of tasks) when implementing the proposed ATLAS scheduler. Figure 3, Figure 5 and Figure 7 present respectively the number of finished jobs, map and reduce tasks for the three schedulers. Overall, we observe that the number of finished jobs, map and reduce tasks in ATLAS are higher in comparison to the results obtained for the FIFO, Fair, and Capacity schedulers. This was expected since the prediction model enables the quick rescheduling of tasks that are predicted to fail. In addition, the improvement is larger for FIFO and Fair schedulers compared to the capacity scheduler. This happens because the capacity scheduler forces the killing of any tasks consuming more memory than configured [10].

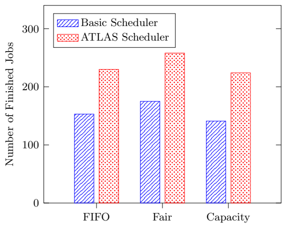

Fig. 3: Finished Jobs

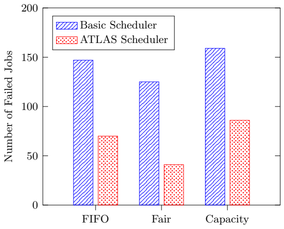

Fig. 4: Failed Jobs

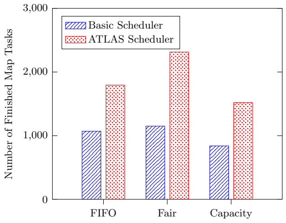

Fig. 5: Finished Map Tasks

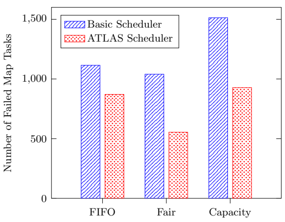

Fig. 6: Failed Map Tasks

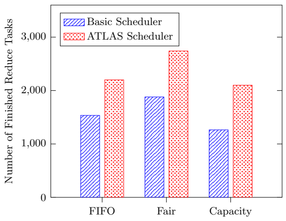

Fig. 7: Finished Reduce Tasks

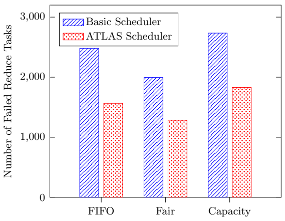

Fig. 8: Failed Reduce Tasks

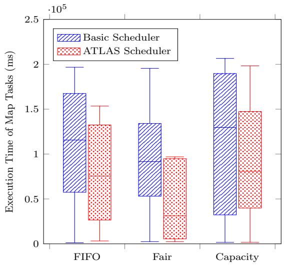

Fig. 9: Map Exec. Time

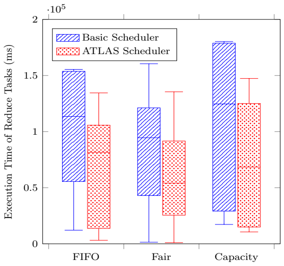

Fig. 10: Reduce Exec. Time

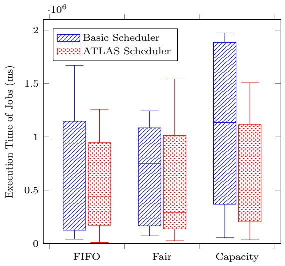

Fig. 11: Total Exec. Time of Jobs

Job/Task	Scheduler	FIFO		Fair		Capacity	
		Basic	ATLAS	Basic	ATLAS	Basic	ATLAS
	Resource	Avg.	Avg.	Avg.	Avg.	Avg.	Avg.
Job	CPU (ms)	11495	8415	12647	9538	14475	10784
	Memory (10^5 bytes	7479	4530	7741	3647	9463	5486
	HDFS Read (10^3 bytes)	9930	7431	10968	8762	12463	8360
	HDFS Write (10^3 bytes)	8583	5985	9784	6202	10285	7420
Task	CPU (ms)	3855	2520	4033	2184	4170	2851
	Memory (10^5 bytes	1412	1058	2496	1741	2638	2115
	HDFS Read (10^5 bytes)	1638	1215	1894	1428	7426	4541
	HDFS Write (10^5 bytes)	1774	1385	3643	2429	5052	3715

TABLE III: Resources Utilisation of the Different Hadoop Schedulers

The number of finished tasks is improved by up to 46% when using ATLAS instead of the Fair scheduler (see *ATLAS-Fair* in Figure 5), and the number of finished jobs increased by 27% when using ATLAS instead of the Fair scheduler (see *ATLAS-Fair* in Figure 3). The improvement of the number of finished jobs is lower than the improvement of the number of finished tasks since a single task failure causes the whole job to fail. We also noticed that the number of failed jobs and tasks was decreased by up to 28% for the jobs (see *ATLAS-Fair* in Figure 4) and up to 39% for the tasks (see *ATLAS-Capacity* in Figure 6) respectively. Moreover, we also observed that when the failure of one map task causes the failure of the dependent reduce tasks belonging to the same job, ATLAS is unable to propose a better scheduling decision (because some data are lost, as explained in Figure 1). Moreover, we noticed that the number of finished single and chained jobs was improved. This was expected because ATLAS enables the successful processing of the tasks composing these jobs and because of the dependency between the jobs within the chained jobs. In addition, the number of finished single jobs was higher than the number of finished chained jobs. This is due to the dependency between the jobs composing the chained one (*i.e.*, sequential ones). Also, a single job failure in the composed chain can cause the failure of the whole chained job. **In general, we conclude that our proposed ATLAS scheduler can reduce up to 39% of tasks failures and up to 28% of jobs failures that are experienced by the 3 other schedulers (*i.e.*, FIFO, Fair, and Capacity).**

Overall, the execution time of tasks and jobs is lower for ATLAS. We attribute this outcome to the fact that *ATLAS* reduces the number of launched attempts and the time spent to execute the attempts. The total execution time of jobs was decreased on average by 10 minutes (from 20 minute to 10 minute), representing a 30% of reduction of the total execution time of these jobs (see the *ATLAS-Capacity* in Figure 11) and the total execution time of tasks by about 1.33 minute (from 2.33 minute to 1 minute) (see *ATLAS-Capacity*: reduce task in Figure 10). For long running jobs (running for 40-50 minutes), the reduction was up to 25 minutes (representing a 54% reduction) over the Capacity scheduler. In this context, we should mention that there was an overhead associated with the training phase of the predictive algorithm and the communication between the JobTracker and TaskTrackers. However, this overhead was very small and is included in the execution time of ATLAS presented on Figure 11, Figure 9, and Figure 10. In fact, the reduction in the number of failures largely compensated for this overhead.
ALTLAS successfully reduces the overall execution times of tasks and jobs in Hadoop.

By enabling the rescheduling of potential failed tasks in advance, ATLAS reduces the total execution time of the tasks and the number of tasks and jobs failure events. Consequently, it is expected that it will reduce resource utilisations in the cluster, since it should save the amount of resources that would have been consumed by tasks failed attempts. The results presented in Table III confirms this anticipated outcome. In fact, by quickly rescheduling tasks predicted to be failed, ATLAS can save the resources that would have been consumed by these tasks. ATLAS speculatively executes the tasks predicted to be failed on multiple nodes to increase their chance of success. Whenever one of these tasks achieves a satisfactory progress, the other speculative executions are stopped.
Overall, the jobs and tasks executed using ATLAS scheduling policies consumed less resources than those executed using the FIFO, Fair, or Capacity schedulers (in terms of CPU, Memory, HDFS Read and Write).

VI. Threats To Validity

This section discusses the threats to validity of our study:

Construct validity threats concern the relation between theory and observation. When building the predictive models used by ATLAS, we did not include information about the requested resources by tasks, since this information was not available in the collected logs. However, it is possible that some tasks were failed because they did not receive their requested resources. Our predictive models would hardly predict such failures. Nevertheless, we used the number of available slots in the machines to predict task failure in case of shortage of resources. Hence, ATLAS can reschedule the task on a different machine (with enough resources).

Internal validity threats concern the tools used to implement ATLAS. In addition, we used AnarchyApe [9] to inject different types of failures in Hadoop machines. We relied on rate of failures observed in Google clusters [4]. It is possible that the majority of Hadoop clusters do not experience such high rate of failures. It is also very possible that our simulations missed some types of failures occurring in Hadoop clusters. Future works should be performed on a more diverse set of Hadoop clusters and different failure rates.

Conclusion validity threats concern the relation between the treatment and the outcome. We implemented a procedure to check the time spent by ATLAS in a way to not exceed the time-out value specified by the scheduler. In addition, we carefully checked the time spent by the algorithm to check the availability of the TaskTarckers and the DataNodes and to activate them through the JobTracker in a way that do not generate extra overhead times. We also verified that the scheduling decisions generated by ATLAS did not violate any property of the system.

Reliability validity threats concern the possibility to replicate our study. We believe that our proposed approach can be reused on other cloud platforms such as Microsoft Azure. To do that, a developer only needs to record the log files of processing nodes, build the predictive models, and implement the ATLAS algorithm on top of a Hadoop scheduler (like FIFO, Fair or Capacity) to adjust scheduling decisions according to task failure predictions.

External validity threats concern the generalization of our obtained results. Further validation on larger clusters using diverse sets of tasks and jobs is desirable.

VII. Related Work

Given the dynamic nature of Hadoop environment, its scheduler can use information about the factors affecting their behavior to make better scheduling decisions and improve cluster performance. One scheduler that was proposed to do that is LATE [11]. LATE collects data about running tasks and assigns weights to tasks based on their progress. Using historical information about the weights assigned to tasks in the past, LATE prioritizes new tasks waiting to be executed. LATE was able to improve the execution time of jobs by a factor of 2 in large Hadoop cluster. Quan *et al.* proposed SAMR (Self-Adaptive MapReduce scheduling), a scheduler that uses hardware system information over time to estimate the progress of tasks and adjust the weights of map and reduce tasks, to minimize the total completion time of a job [12]. SAMR does not consider job characteristics such as size, execution time, or weights. To improve on this limitation of SAMR, Xiaoyu *et al.* proposed ESAMR(Enhanced Self-Adaptive MapReduce scheduling) [13] which considers system information about straggling tasks, jobs length, etc. ESAMR uses the K-means clustering algorithm to estimate task execution times. In [14], Tang *et al.* proposed a scheduling algorithm named SARS (Self-Adaptive Reduce Start time) which uses job completion time, reduce completion time and the total completion time information as well as system information to decide on the starting time of reduce tasks. By improving the decisions about when to start reduce tasks, SARS could reduce the average response time of the tasks by 11%.

VIII. Conclusion and Future Work

In this paper, we proposed ATLAS (AdapTive faiLure-Aware Scheduler), a new scheduler for Hadoop. The primary goal of ATLAS is to reduce the failure rates of jobs and tasks and their running times in Hadoop clusters. Based on information about events occurring in the cloud environment and statistical models, ATLAS can adjust its scheduling decisions accordingly, in order to avoid failure occurrences. We implemented ATLAS in Hadoop deployed on Amazon Elastic MapReduce (EMR) and performed a case study to compare its performance with those of the FIFO, Fair and Capacity schedulers. Results show that ATLAS can reduce the percentage of failed jobs by up to 28% and the percentage of failed tasks by up to 39%. Although ATLAS requires training a predictive model, we found that the reduction in the number of failures largely compensates for the model training time. ATLAS could reduce the total execution time of jobs by 10 minutes on average, and by up to 25 minutes for long running jobs. ATLAS also reduces CPU and memory usages, as well as the number of HDFS Reads and writes. In the future, we plan to extend ATLAS using unsupervised learning algorithms and assess the performance of ATLAS when the prediction model is retrained at fixed time intervals.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *ACM Communications*, 51(1):107–113, 2008.

[2] A. Rasooli and D. G. Down, "A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems," in *International Conference on SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1284–1291, 2012.

[3] F. Dinu and N. Eugene, "Understanding the Effects and Implications of Compute Node Related Failures in Hadoop," in *Symposium on High-Performance Parallel and Distributed Computing*, pp. 187–198, 2012.

[4] M. Soualhia, F. Khomh, and S. Tahar, "Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR," in *International Conference on High Performance Computing and Communications*, pp. 58–65, 2015.

[5] Y. Ji, L. Tong, T. He, J. Tan, K. won Lee, and L. Zhang, "Improving Multi-job MapReduce Scheduling in an Opportunistic Environment," in *International Conference on Cloud Computing*, pp. 9–16, 2013.

[6] M. Soualhia, F. Khomh, and S. Tahar, "ATLAS: An Adaptive Failure-Aware Scheduler for Hadoop," Department of Electrical and Computer Engineering, Concordia University, Montreal, QC, Canada, Tech. Rep., 2015. [Online]. Available: http://arxiv.org/abs/1511.01446

[7] The R Project for Statistical Computing. [Online]. Available: http://www.r-project.org/, Last Access October, 2015

[8] Amazon EC2 Instances. [Online]. Available: http://aws.amazon.com/ec2/instance-types/, Last Access October, 2015

[9] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. Campbell, and W. H. Sanders, "Failure Scenario As a Service (FSaaS) for Hadoop Clusters," in *International Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, pp. 5:1–5:6, 2012.

[10] Hadoop capacity scheduler. [Online]. Available: http://hadoop.apache.org/docs/current1/capacity_scheduler.html, Last Access October, 2015

[11] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *International Conference on Operating Systems Design and Implementation*, pp. 29–42, 2008.

[12] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment," in *International Conference on Computer and Information Technology*, pp. 2736–2743, 2010.

[13] X. Sun, C. He, and Y. Lu, "ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm," in *International Conference on Parallel and Distributed Systems*, pp. 148–155, 2012.

[14] Z. Tang, L. Jiang, J. Zhou, K. Li, and K. Li, "A Self-Adaptive Scheduling Algorithm for Reduce Start Time," *Future Generation Computer Systems*, 34-44(0):51-60, 2015.