ISSN No. 0976-5697

Volume 5, No. 6, July-August 2014

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Hadoop MapReduce Multi-Job Workloads using Resource Aware scheduler

Shivakumar.N[1], Rashmi S[2], and Anirban Basu [3]
[1,2,3]Department of Computer Science and Engineering,
East Point College of Engineering and Technology, Bangalore, India

Abstract-Cloud computing features a flexible computing infrastructure for large-scale data processing. MapReduce is a typical model providing an logical framework for cloud computing and Hadoop, an open-source implementation of MapReduce, is a common platform to realize such kind of parallel computing model. We present a resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization across machines while observing completion time goals. Existing MapReduce schedulers define a static number of slots to represent the capacity of a cluster, creating a fixed number of execution slots per machine. This abstraction works for homogeneous workloads, but fails to capture the different resource requirements of individual jobs in multi-user environments. Our technique leverages job profiling information to dynamically adjust the number of slots on each machine, as well as workload placement across them, to maximize the resource utilization of the cluster.

Key Words- Map Reduce, scheduling, resource-awareness, performance Management, Large-Scale Data Processing, Hadoop.

I. INTRODUCTION

With Increase in the development of web applications, peta bytes of data generated by various kinds of network applications are processed in the Internet. Cloud computing is developed to process massive data, such as distributed data sorting, log analyzing, machine learning and so on. Analyzing these huge volumes of data requires a scalable solution, MapReduce[2] , is one well-known cloud computing model, features an efficient framework to analyze data in parallel with flexible job decomposition and sub-tasks allocation. MapReduce can be deployed on a large number of suitable machines and can automatically handle node failures.

Hadoop, a project maintained by Apache Software Foundation and an open-source implementation of MapReduce, is primarily used by Yahoo and also Facebook, Amazon and Baidu etc. Hadoop is a suitable platform to deal with variety of applications such as data mining and extraction on large-scale of data. In Hadoop, there are multiple Map and Reduce tasks in a MapReduce job. Each task is a single unit of work that can be performed together with other tasks in parallel.

Assigning tasks to node assign is performed by a master node, which distributes tasks to slave nodes[1]. Each slave node has a fixed number of Map and Reduce slots for executing Map and Reduce tasks. At any time, each slot can run only one task. Slot offers a simple abstraction of the available resources on a physical machine. The primary advantage of slots is the ease of implementation of the MapReduce programming model in Hadoop.

The industry and research community have witnessed an remarkably good growth in research and development of data-analytic technologies. Pivotal to this phenomenon is the adoption of the MapReduce programming paradigm and its open-source implementation Hadoop. Pioneer implementations of MapReduce have been designed to provide overall system goals (e.g., job throughput). Thus, support for user-defined goals and resource utilization management have been left as secondary considerations at best.

We believe that both capabilities are crucial for the further development and adoption of large-scale data processing. On one hand, more users wish for ad-hoc processing in order to perform short-term tasks[9]. Furthermore, in a Cloud environment users pay for resources used. Therefore, providing consistency between price and the quality of service obtained is key to the business model of the Cloud. Resource management, on the other hand, is also important as Cloud providers are motivated by profit and hence require both high levels of automation and resource utilization while avoiding bottlenecks.

II. BACKGROUND KNOWLEDGE

A. MapReduce:

MapReduce, introduced by Google in 2004, is one of the famous software frameworks to support distributed computing on Large amount of data sets on clusters of computers. It is widely used in various kinds of applications like distributed data sorting, log file analyzing and machine learning and so on. The main aim of MapReduce is to distribute the processing across many nodes to take advantage of parallel processing power. This is generally done by dividing the dataset into several chunks, and then processing those chunks in separate nodes.

ISSN No. 0976-5697

Volume 5, No. 6, July-August 2014

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

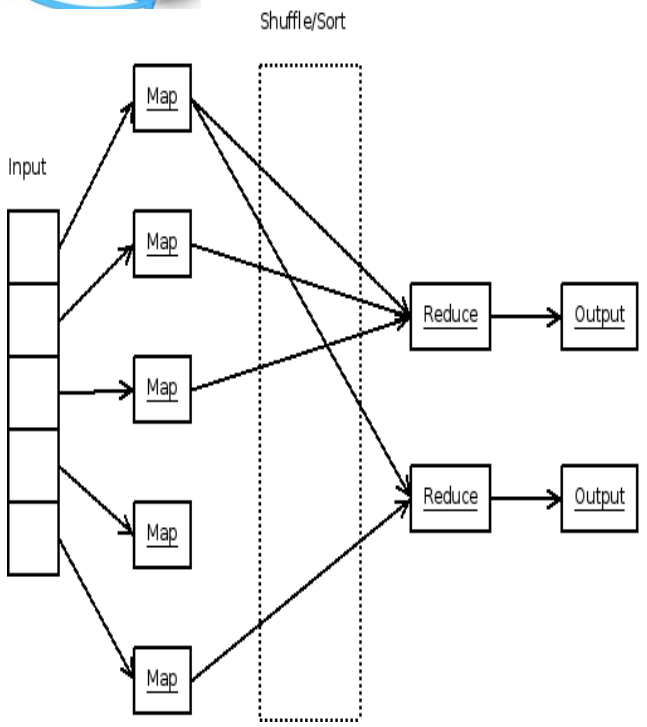


Figure 1. MapReduce

A MapReduce job mainly consists of two phases, Map and Reduce. The input data-sets are split into independent pieces of default size. The Map task turns the chunk into a set of key-value pairs. One Map process is invoked to process one chunk of input data, and each Mapping operation is independent to each others. Then, the intermediate key-value pairs from the output of each Map task are collected and sorted by key, then transferred to the location where a Reduce process would operate on the intermediate data. Reduce tasks merge all intermediate values associated with the same intermediate key to form a possibly smaller data set. In other words, all key-value pairs with the similar key complete at the same Reduce task. MapReduce runs on a large cluster of commodity machines. A large server cluster can use MapReduce to sort peta bytes of data in only a few hours. The parallelism also offers some possibility of fault-tolerant. If one Map task or Reduce task fails, the work can be recovered by rescheduling.

B. Hadoop:

Apache Hadoop is an open-source implementation of the MapReduce programming model. Hadoop follows the master/slave architecture and consists of one master machine responsible for organizing the distribution of work, and a set of worker machine responsible for executing the work assigned by the master. Hadoop focuses on distributed storing and processing on large data[3]. It is designed to scale up from a single node to thousands of ones, with a very high degree of fault tolerance. In the Hadoop environment, the MapReduce framework consists of a single JobTracker on the master node and one TaskTracker per slave node in the cluster.

First, client applications submit jobs to JobTracker. Jobs are split into many tasks by default size. JobTracker communicates to the NameNode to determine the location of the data, and then submits the Map or Reduce task to the chosen TaskTracker nodes. When the job is completed, JobTracker updates its status and stores the output data.

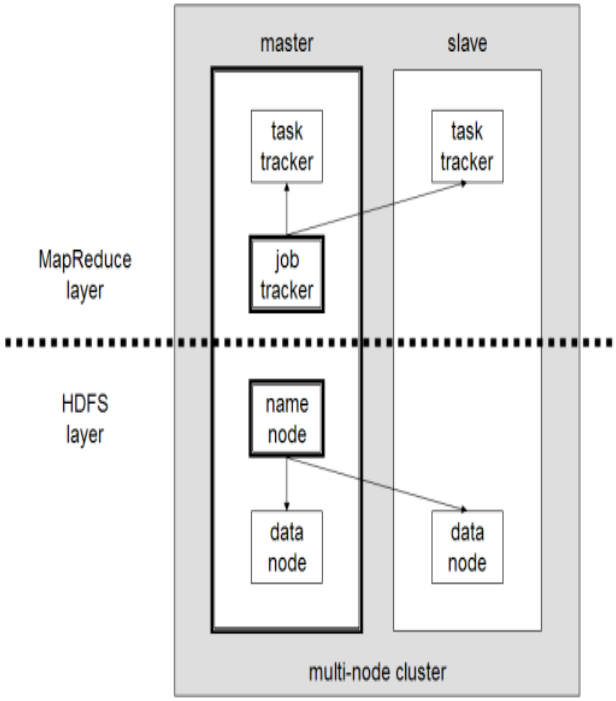


Figure 2.Hadoop Architecture

In Hadoop, resources are depicted by the concept of 'slot'. On the other hand, each TaskTracker is responsible for a specific number of slots on each node. The slot number figure out the max running number of tasks which are allowed to be run in parallel on that node at a time. The scheduling policy in Hadoop is based on the fixed slot number for the lifetime of each node. The slot number depicts the computation ability on the node and can be configured in an XML file.

Originally, JobTracker chooses the processing TaskTracker in the cluster by one rule - keeping the work as close to the data as possible. That is, when JobTracker tries to schedule a task with the MapReduce operations, it first sees for an empty slot on the same server that hosts the DataNode containing the processing data. If not, it allocates work to one TaskTracker nearest to the data with an available free slot. During the processing time, each TaskTracker send out heartbeat messages to JobTracker every certain time period to notice JobTracker that it is still alive. If JobTracker does not receive the heartbeat signals from TaskTracker because of node failure or timeout, the corresponding job would be rescheduled to another node. JobTracker is responsible for scheduling the tasks on the slaves, monitoring TaskTracker and again scheduling the failed tasks. TaskTracker only needs to execute the Map or Reduce task that is issued by JobTracker.

III. LITERATURE SURVEY

Much work have been done in the space of scheduling for MapReduce. Since the number of slots in a Hadoop cluster is fixed throughout the lifetime of the cluster, most of the proposed solutions can be reduced to a variant of the task-assignment or slot-assignment problem. The Capacity Scheduler is a pluggable scheduler developed by Yahoo! which partition resources into pools and provides priorities for each pool. Hadoop's Fair Scheduler allocates equal shares to each tenant in the cluster.

Quincy scheduler proposed for the Dryad environment also shares similar fairness goals. All these schedulers are built on top of the slot model and do not support user-level goals. The performance of MapReduce jobs has attracted much interest in the Hadoop community. Stragglers, tasks that take an unusually long time to complete, have been shown to be the most common reason why the total time to execute a job increases[7] . Speculative scheduling has been widely adopted to complement of one other that impact of stragglers,. Under this scheduling strategy, when the scheduler detects that a task is taking longer than expected it spawns multiple instances of the task and takes the results of the first completed instance, killing the others .

In Mantri the effect of stragglers is mitigated via the `kill and restart' of tasks which have been noticed as potential stragglers.

The main disadvantage of these techniques is that killing and duplicating tasks results in wasted or loss of resources [9, 5]. In RAS we take a more proactive approach, in that we prevent stragglers resulting from resource contention. Furthermore, stragglers caused by distorted data cannot be avoided at run-time by any existing technique. In RAS the slow-down effect that these stragglers have on the end-to- end completion time of their corresponding jobs is mitigated by allocating more resources to the job so that it can still complete in a specified timely manner.

Recently, there has been increasing interest in user-centric data analytics. One of the seminal works in this space is. In this work, the authors propose a scheduling scheme that enables soft-deadline support for MapReduce jobs[7]. It differs from RAS in that it does not take into consideration the resources in the system. Flex is a scheduler proposed as an add-on to the Fair Scheduler to provide Service-Level-Agreement (SLA) guarantees. More recently, ARIA : Automatic Resource Inference and Allocation for MapReduce Environments," introduces a fair resource management framework that consists of a job profiler, a model for MapReduce jobs and a SLO-scheduler based on the Earliest Deadline First scheduling strategy. Flex and Aria are both slot-based and therefore suffers from the same limitations we mentioned earlier. One of the first works in considering resource awareness in MapReduce clusters by J. Dhok and V. Varma.

In this paper the scheduler classifies tasks into good and bad tasks depending on the load they impose in the worker machines. More recently, the Hadoop community has also recognized the importance of developing a resource-aware scheduling for MapReduce. Arun Murthy[8]. Next Generation Hadoop outlines the vision behind the Hadoop scheduler of the future. The framework proposed introduces a resource model consisting of a `resource container' which is|like our `job slot'|fungible across job tasks. We think that our proposed re- source management techniques can be leveraged within this framework to enable better resource management.

The Hadoop architecture follows the master/slave paradigm. It consists of a master machine responsible for coordinating the distribution of work and execution of jobs, and a set of worker machine responsible for performing work assigned by the master. The master and slaves roles are performed by the 'JobTracker' and 'TaskTracker' processes, respectively.

The singleton JobTracker partitions the input data into 'input splits' using a splitting method defined by the programmer, populates a local task-queue based on the number of obtained input splits, and distributes work to the TaskTrackers that in turn process individual splits. Work units are represented by 'tasks' in this framework. There is one map task for every input split generated by the JobTracker. The number of reduce tasks is defined by the user. Each TaskTracker controls the execution of the tasks assigned to its hosting machine.

The driving principles of RAS are *resource awareness* and continuous *job performance management*. The former is used to decide task placement on TaskTrackers over time, and is the main objective. The latter is used to estimate the number of tasks to be run in parallel for each job in order to meet some performance objectives, expressed in RAS in the form of completion time Goals.

These goals are treated as soft deadlines in RAS as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management. In order to enable this resource awareness, we introduce the concept of 'job slot'.

A job slot is an execution slot that is bound to a particular job, and a particular task type (reduce or map) within that job. A slot is bound only to a task type regardless of the job. This extension allows for a finer-grained resource model for MapReduce jobs. Additionally, RAS determines the number of job slots, and their placement in the cluster, dynamically at run-time. This contrasts sharply with the traditional approach of requiring the system administrator to statically and homogeneously configure the slot count and type on a cluster. This eases the configuration burden and improves the behaviour of the MapReduce cluster. The number of slots per TaskTracker determines the maximum number of concurrent tasks that are allowed to run in the worker machine.

Heartbeat is a program that runs specialized scripts automatically whenever a system is initialized or rebooted. Originally designed for two-node Linux-based clusters, Heartbeat is extensible to larger configurations. In a system running Heartbeat, nodes communicate by exchanging packets called "heartbeats" at the rate of approximately 2 Hz (twice per second). The heartbeat packets contain the number of map and reduce tasks remaining in that particular node[5].

The master machine upon receiving heartbeat packets updates the status of map and reduce slots to be used by scheduler in coordination with the JobTracker to process the incoming jobs which are partitioned into input splits and been assigned with execution slots for performing either a map task or reduce task.

IV. ALGORITHM

Algorithm: Placement Algorithm runs at each Control Cycle

Inputs P^M(job, tt) : Placement Matrix of Map tasks, P^R (job, tt) : Placement Matrix of Reduce tasks, J: List of Jobs in the System, D: Resource demand profile for each job, TT: List of TaskTrackers in the System, Γ_j and Ω_{tt}: Resource demand and capacity for each Job each TaskTracker correspondingly, as used by the auxiliary function *room_for_new_job_slot*.

{——————— Place Reducers ———————}

1: **for** job in J **do**
2: Sort TT in increasing order of overall number of reduce tasks placed (first criteria),
and increasing order of number of reducers job placed (second criteria).
3: **for**tt in TT **do**
4: **if** room for new job slot (job, tt) &r^{job}_{pend}> 0 **then**
5: P^R (job, tt) = P^R (job, tt) + 1
6: **end if**
7: **end for**
8: **end for**

{——————— Place Mappers ———————}

9: **for** round = 1. . . rounds**do**
10: **for**tt in TT **do**
11: job_{in} ← min U(job_{in}, P), room for new job slot(job_{in}, tt),
12: job_{out} ← max U(job_{out}, P), P^M(job_{out}, tt) > 0
13: **repeat**
14: P_{old} ← P
15: job_{out} ← max U(job_{out}, P), P(job_{out}, tt) > 0
16: P^M(job_{out}, tt) = P^M(job^{out}, tt) − 1
17: job_{in} ← min U(job_{in}, P), room for new job slot(job_{in}, tt)
18: **until**U(job_{out}, P) < U(job_{in}, P_{old})
19: P ← P_{old}
20: **repeat**
21: job_{in} ← min U(job_{in}, P), room for new job slot(job_{in}, tt)
22: P^M(job^{in}, tt) = P^M(job_{in}, tt) + 1
23: **until** job such that room for new job slot(job, tt)
24: **end for**
25: **end for**
26: **if** map phase of a job is about to complete in this control cycle **then**
27: switch profile of placed reducers from shuffle to reduce and wait for Task Scheduler to drive the transition.
28: **end if**

A. Job Initialization:

Map and reduce tasks run in parallel, as the tasks progresses the number of map tasks reduces, and the number of reduce tasks increases. When the JobTracker receives a call to its submit Job() method, it puts the job into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and log information to keep track of the tasks' status and progress.

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the system from the shared filesystem. It then creates one map task for each split. The number of reduce tasks to be created is determined by the property in the configuration files, and the scheduler simply generates this number of reduce tasks to be run. Tasks are given IDs at this point.

B. Task Assignment:

Heartbeats signal tell the JobTracker that a TaskTracker is alive. The TaskTrackers in the slaves dynamically generates heartbeat packets which also carry the map and reduce slots indicating the status of remaining slots in each of the worker machines. As a part of the heartbeat, a TaskTracker will indicate whether it is ready to run a new task, and if it is ready, the JobTracker will allocate it a task, which it communicates to the TaskTracker using the heartbeat return value.

Before it can choose a task for the TaskTracker, the JobTracker must choose a job to select the task from. There are various scheduling algorithms (example FIFO), but the default one simply maintains a priority list (arrival time) of jobs. Having chosen a job, the JobTracker now chooses a task for the job.

TaskTrackers have a fixed number of slots for map tasks and for reduce tasks: for ex, a TaskTracker may be able to run two map tasks and two reduce tasks continuously. The default scheduler fills empty map task slots before reduce task slots, so if the TaskTracker has at least one empty map task slot, the JobTracker will select a map task; otherwise, it will select a reduce task.

To choose a reduce task, the JobTracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations. For a map task, however, it takes account of the TaskTracker's network location and picks a task whose input split is as close as possible to the TaskTracker. In the optimal case, the task is *data-local*, that is, running on the same node that the divides resides on. Alternatively, the task may be *rack-local*: on the same rack, but not the same node, as the split. Some tasks are neither data-local nor rack-local and retrieve their data from a different rack from the one they are running on.

C. Task Execution:

Now that the TaskTracker has been assigned a task, the next step is for it to run the task. *First*, it localizes the job JAR by copying it from the shared filesystem to the TaskTracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk. *Second*, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. *Third*, it creates an instance of process in it to run the task.

Map and reduce tasks run simultaneously and as the tasks progresses, the number of map task reduces and the number of reduce task increases. This way, the child process informs the parent of its task's progress every few seconds until the task is completed.

D. Progress and Status updates:

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant distance of time, it's important for the user to get feedback on how the job is progressing.

A job and each of its tasks have a *status*, which includes such things as the state of the job or task (e.g., running, successfully completed), the results of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code). These statuses

change over the course of the job, so how do they get communicated back to the client[4].

When a task is running, it keeps track of its *progress*, that is, the proportion of the task completed. For map tasks, the progress is the proportion of the input that has been processed. For reduce tasks, it's a little more complex, but the system can still estimate the proportion of the reduce input processed. It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle ("Shuffle, Sort and Reduce"). For example, if the task has run the reducer on half of its input, then the task's progress is ⅚, since it has completed the copy and sort phases (⅓ each) and is halfway through the reduce phase (⅙).

E. Job completion:

When the JobTracker receives a notification that the last task for a job is complete, it changes the status for the job to "successful." Then the JobTracker also sends an HTTP job notification if it is configured to do so. Last, the JobTracker cleans up its working state for the job and instructs TaskTrackers to do the same (so intermediate output is deleted) and it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the main method.

V. EVALUATION

The results are shown based on the configurations made on multimode cluster of 5 nodes with the capacity of 4GB of RAM in all the three machines with the storage capacity of 250GB hard disk and the processor speed of 2.2GHZ . The snapshots of the execution of the jobs and the graph representing them.

A. Submission of jobs:

```
14/05/14 20:56:14 WARN snappy.LoadSnappy: Snappy native library not loaded
14/05/14 20:56:15 INFO mapred.JobClient: Running job: job_201405142052_0002
14/05/14 20:56:16 INFO mapred.JobClient:  map 0% reduce 0%
14/05/14 20:56:35 INFO mapred.JobClient:  map 1% reduce 0%
14/05/14 20:56:54 INFO mapred.JobClient:  map 2% reduce 0%
14/05/14 20:57:13 INFO mapred.JobClient:  map 3% reduce 0%
14/05/14 20:57:32 INFO mapred.JobClient:  map 4% reduce 0%
14/05/14 20:57:45 INFO mapred.JobClient:  map 5% reduce 0%
14/05/14 20:58:03 INFO mapred.JobClient:  map 5% reduce 1%
14/05/14 20:58:04 INFO mapred.JobClient:  map 6% reduce 1%
14/05/14 20:58:21 INFO mapred.JobClient:  map 6% reduce 2%
14/05/14 20:58:23 INFO mapred.JobClient:  map 7% reduce 2%
14/05/14 20:58:41 INFO mapred.JobClient:  map 8% reduce 2%
14/05/14 20:58:55 INFO mapred.JobClient:  map 9% reduce 2%
14/05/14 20:59:07 INFO mapred.JobClient:  map 9% reduce 3%
14/05/14 20:59:32 INFO mapred.JobClient:  map 10% reduce 3%
14/05/14 21:00:04 INFO mapred.JobClient:  map 11% reduce 3%
14/05/14 21:00:41 INFO mapred.JobClient:  map 12% reduce 3%
14/05/14 21:01:07 INFO mapred.JobClient:  map 12% reduce 4%
14/05/14 21:01:15 INFO mapred.JobClient:  map 13% reduce 4%
14/05/14 21:01:52 INFO mapred.JobClient:  map 14% reduce 4%
```

Figure.4 Start of job1 (20 GB)

This snapshot represents the jobid and submission time of job1. The jobid is job_201405142052_0002 and job is submitted at time 20:56:16. The map task starts at submission time and reduce task starts at time 20:57:45.

```
hduser@master:~$ /usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/hadoop*examples*.jar wordcount /user/hduser/E /user/hduser/Adaptive-out2
Warning: $HADOOP_HOME is deprecated.

14/05/14 20:58:38 INFO input.FileInputFormat: Total input paths to process : 1
14/05/14 20:58:38 INFO util.NativeCodeLoader: Loaded the native-hadoop library
14/05/14 20:58:38 WARN snappy.LoadSnappy: Snappy native library not loaded
14/05/14 20:58:38 INFO mapred.JobClient: Running job: job_201405142052_0003
14/05/14 20:58:39 INFO mapred.JobClient:  map 0% reduce 0%
14/05/14 20:59:12 INFO mapred.JobClient:  map 1% reduce 0%
14/05/14 20:59:31 INFO mapred.JobClient:  map 2% reduce 0%
14/05/14 20:59:50 INFO mapred.JobClient:  map 3% reduce 0%
```

Figure.5 Start of job2 (10 GB)

This snapshot represents the jobid and submission time of job2. The jobid is job_201405142052_0003 and job is submitted at time 20:58:39. The map task starts at submission time.

B. End of the submitted jobs:

```
14/05/14 21:37:08 INFO mapred.JobClient:  map 91% reduce 30%
14/05/14 21:37:23 INFO mapred.JobClient:  map 92% reduce 30%
14/05/14 21:37:42 INFO mapred.JobClient:  map 93% reduce 30%
14/05/14 21:37:56 INFO mapred.JobClient:  map 93% reduce 31%
14/05/14 21:38:01 INFO mapred.JobClient:  map 94% reduce 31%
14/05/14 21:38:19 INFO mapred.JobClient:  map 95% reduce 31%
14/05/14 21:38:32 INFO mapred.JobClient:  map 96% reduce 31%
14/05/14 21:38:41 INFO mapred.JobClient:  map 96% reduce 32%
14/05/14 21:38:51 INFO mapred.JobClient:  map 97% reduce 32%
14/05/14 21:39:09 INFO mapred.JobClient:  map 98% reduce 32%
14/05/14 21:39:26 INFO mapred.JobClient:  map 99% reduce 32%
14/05/14 21:39:38 INFO mapred.JobClient:  map 99% reduce 33%
14/05/14 21:39:41 INFO mapred.JobClient:  map 100% reduce 33%
14/05/14 21:39:46 INFO mapred.JobClient:  map 100% reduce 100%
14/05/14 21:39:46 INFO mapred.JobClient: Job complete: job_201405142052_0002
14/05/14 21:39:46 INFO mapred.JobClient: Counters: 29
14/05/14 21:39:46 INFO mapred.JobClient:   Job Counters
14/05/14 21:39:46 INFO mapred.JobClient:     Launched reduce tasks=1
14/05/14 21:39:46 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=6859003
14/05/14 21:39:46 INFO mapred.JobClient:     Total time spent by all reduces waiting after reserving slots (ms)=0
14/05/14 21:39:46 INFO mapred.JobClient:     Total time spent by all maps waiting after reserving slots (ms)=0
14/05/14 21:39:46 INFO mapred.JobClient:     Launched map tasks=305
```

Figure.6 End of job1

The job1 with jobid job_201405142052_0002 completes its map tasks at time 21:39:41 and reduce tasks at time 21:39:46. The number of map tasks launched is 305 and reduce tasks is 1.

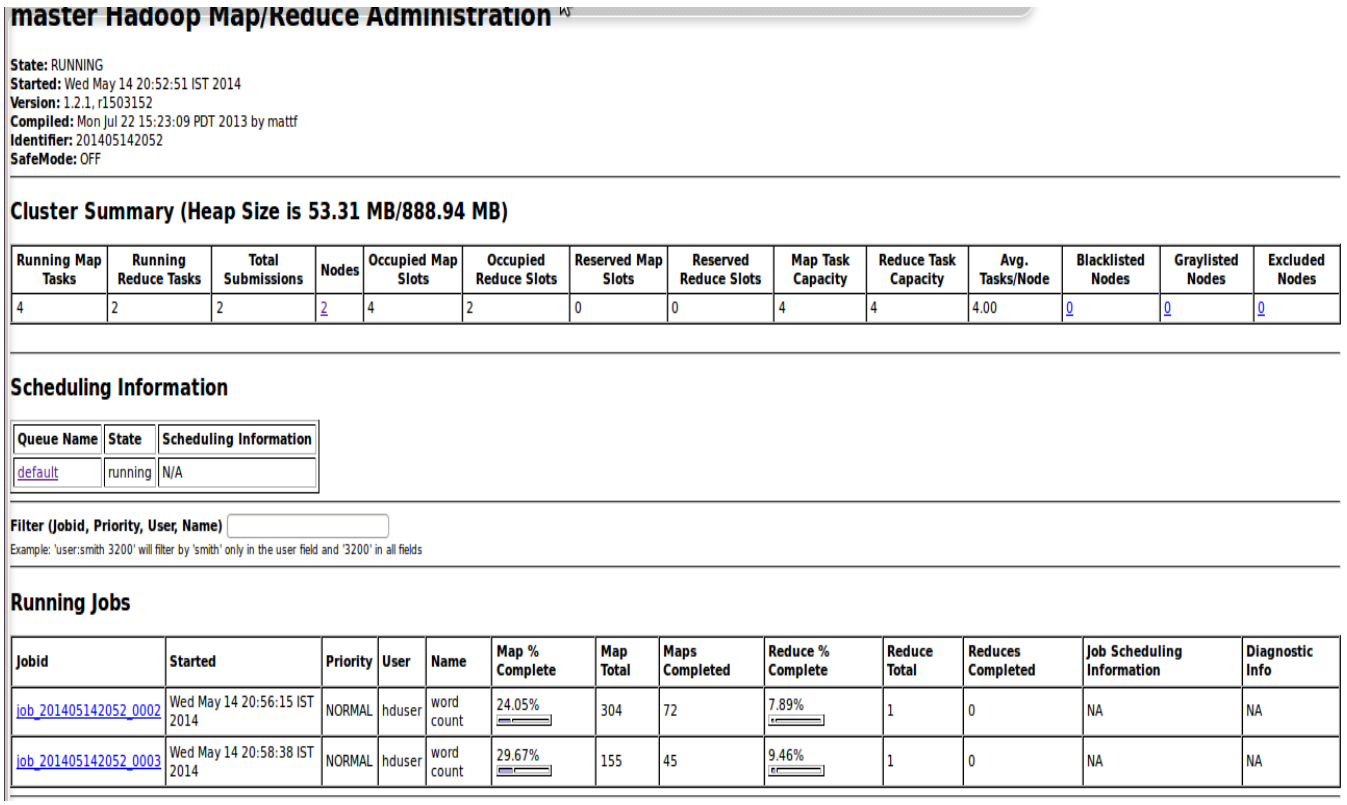
```
14/05/14 21:26:16 INFO mapred.JobClient:  map 93% reduce 30%
14/05/14 21:26:28 INFO mapred.JobClient:  map 94% reduce 30%
14/05/14 21:26:41 INFO mapred.JobClient:  map 94% reduce 31%
14/05/14 21:26:48 INFO mapred.JobClient:  map 95% reduce 31%
14/05/14 21:27:05 INFO mapred.JobClient:  map 96% reduce 31%
14/05/14 21:27:24 INFO mapred.JobClient:  map 97% reduce 31%
14/05/14 21:27:26 INFO mapred.JobClient:  map 97% reduce 32%
14/05/14 21:27:38 INFO mapred.JobClient:  map 98% reduce 32%
14/05/14 21:27:56 INFO mapred.JobClient:  map 99% reduce 32%
14/05/14 21:28:06 INFO mapred.JobClient:  map 100% reduce 32%
14/05/14 21:28:11 INFO mapred.JobClient:  map 100% reduce 33%
14/05/14 21:28:14 INFO mapred.JobClient:  map 100% reduce 100%
14/05/14 21:28:15 INFO mapred.JobClient: Job complete: job_201405142052_0003
14/05/14 21:28:15 INFO mapred.JobClient: Counters: 29
14/05/14 21:28:15 INFO mapred.JobClient:   Job Counters
14/05/14 21:28:15 INFO mapred.JobClient:     Launched reduce tasks=1
14/05/14 21:28:15 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=3470282
14/05/14 21:28:15 INFO mapred.JobClient:     Total time spent by all reduces waiting after reserving slots (ms)=0
14/05/14 21:28:15 INFO mapred.JobClient:     Total time spent by all maps waiting after reserving slots (ms)=0
14/05/14 21:28:15 INFO mapred.JobClient:     Launched map tasks=155
14/05/14 21:28:15 INFO mapred.JobClient:     Data-local map tasks=155
14/05/14 21:28:15 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=1669868
14/05/14 21:28:15 INFO mapred.JobClient:   File Output Format Counters
14/05/14 21:28:15 INFO mapred.JobClient:     Bytes Written=1137472
```

Figure.7 End of job2

The job2 with jobid job_201405142052_0003 completes its map tasks at time 21:28:06 and reduce tasks at time 21:28:14. The number of map tasks launched is 155 and reduce tasks is 1.

C. JobTracker Status:

JobTracker includes the user who submitted the job, priority of the job, name of the program that is executed and status of execution of current map and reduce tasks executing.



master Hadoop Map/Reduce Administration

State: RUNNING
Started: Wed May 14 20:52:51 IST 2014
Version: 1.2.1, r1503152
Compiled: Mon Jul 22 15:23:09 PDT 2013 by mattf
Identifier: 201405142052
SafeMode: OFF

Cluster Summary (Heap Size is 53.31 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
4	2	2	2	4	2	0	0	4	4	4.00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

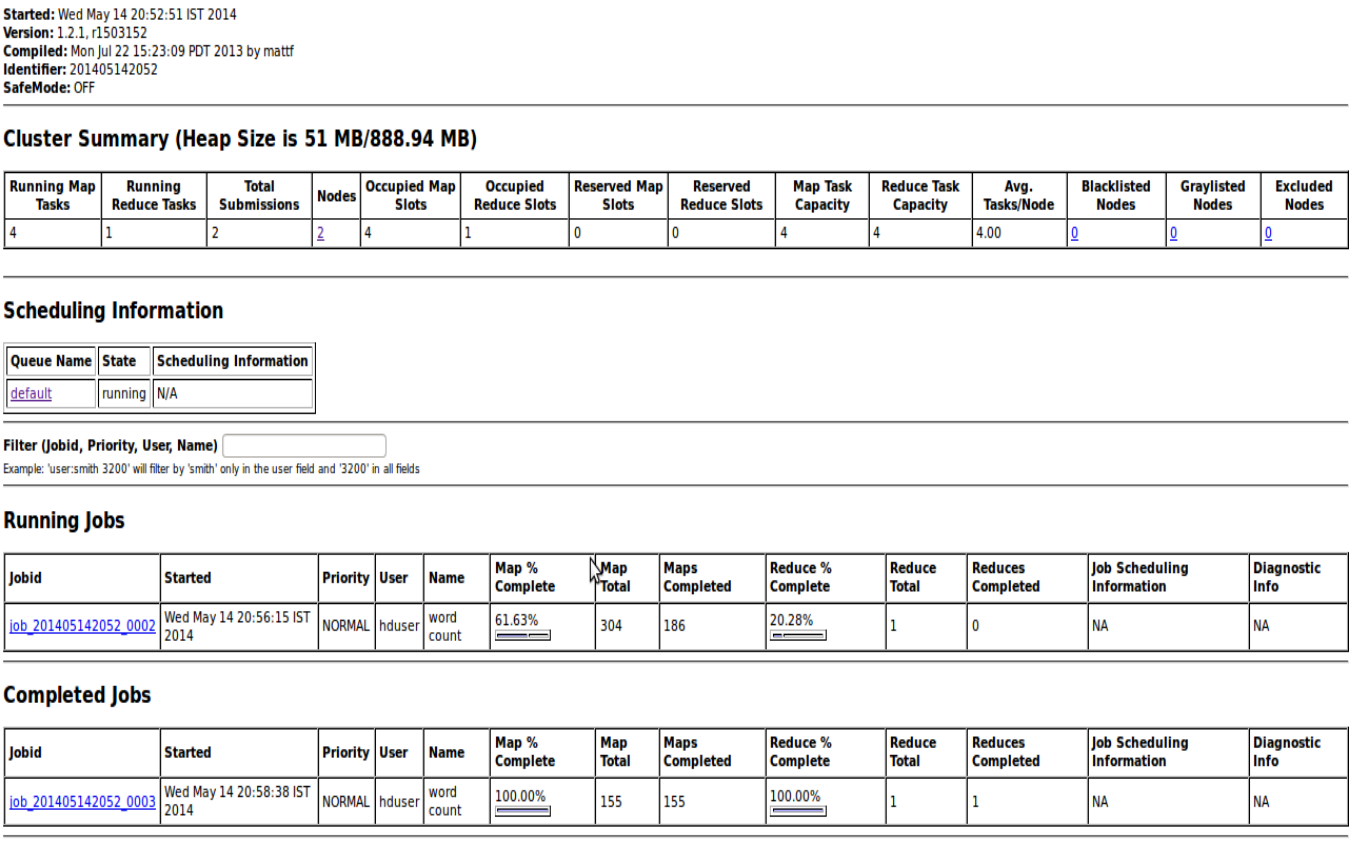
Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Started	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201405142052_0002	Wed May 14 20:56:15 IST 2014	NORMAL	hduser	word count	24.05%	304	72	7.89%	1	0	NA	NA
job_201405142052_0003	Wed May 14 20:58:38 IST 2014	NORMAL	hduser	word count	29.67%	155	45	9.46%	1	0	NA	NA

Figure.8 Simultaneous execution of job1, job2

At this instance, the job2 is submitted with 155 map tasks and 1 reduce task launched along with the execution of job1. The status of map and reduce tasks completed of job1 and job2 is also indicated.



Started: Wed May 14 20:52:51 IST 2014
Version: 1.2.1, r1503152
Compiled: Mon Jul 22 15:23:09 PDT 2013 by mattf
Identifier: 201405142052
SafeMode: OFF

Cluster Summary (Heap Size is 51 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
4	1	2	2	4	1	0	0	4	4	4.00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Started	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201405142052_0002	Wed May 14 20:56:15 IST 2014	NORMAL	hduser	word count	61.63%	304	186	20.28%	1	0	NA	NA

Completed Jobs

Jobid	Started	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201405142052_0003	Wed May 14 20:58:38 IST 2014	NORMAL	hduser	word count	100.00%	155	155	100.00%	1	1	NA	NA

Figure.9 Completion status of job2

ISSN No. 0976-5697

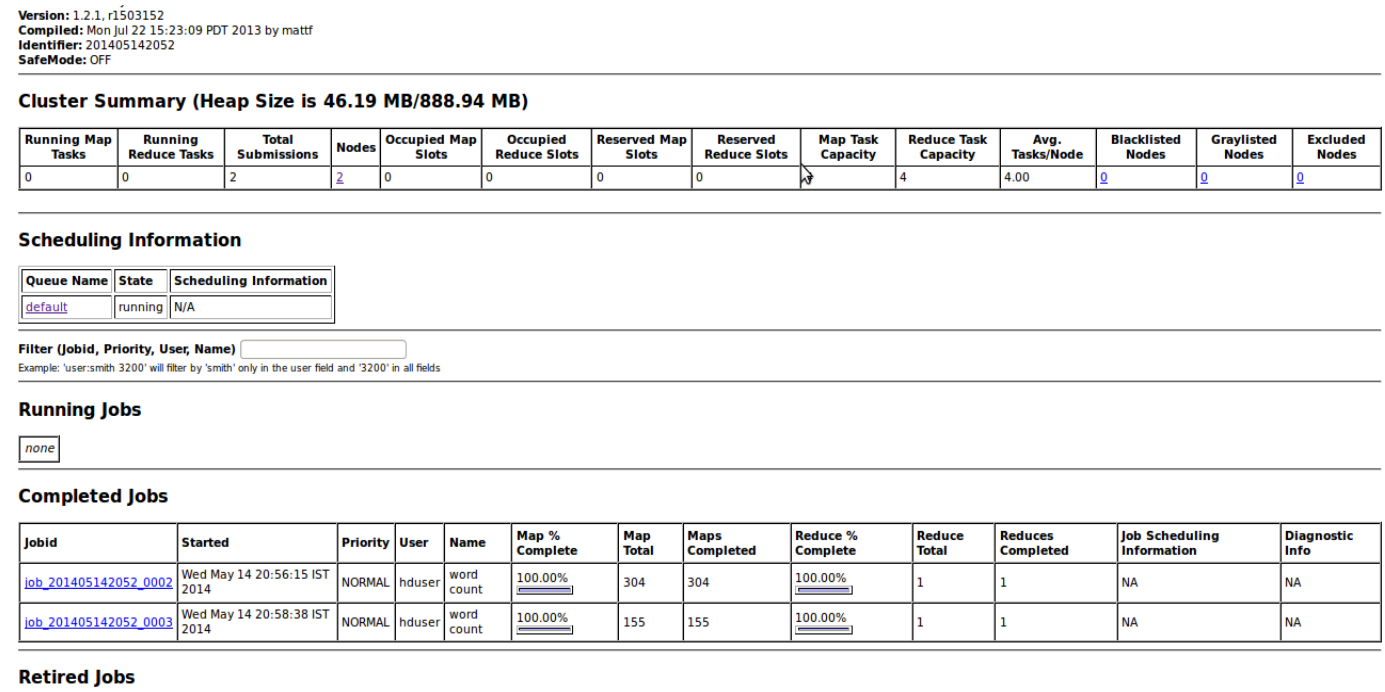
Volume 5, No. 6, July-August 2014

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Since, the size of job2 considered is less than job1, the job2 is completed before the job1 is completed, even though it is submitted later.



Version: 1.2.1, r1503152
Compiled: Mon Jul 22 15:23:09 PDT 2013 by mattf
Identifier: 201405142052
SafeMode: OFF

Cluster Summary (Heap Size is 46.19 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
0	0	2	2	0	0	0	0		4	4.00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Started	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201405142052_0002	Wed May 14 20:56:15 IST 2014	NORMAL	hduser	word count	100.00%	304	304	100.00%	1	1	NA	NA
job_201405142052_0003	Wed May 14 20:58:38 IST 2014	NORMAL	hduser	word count	100.00%	155	155	100.00%	1	1	NA	NA

Retired Jobs

Figure.10 Completion status of job1

After the successful completion of all map and reduce tasks of job1 utilizing complete resources of the nodes (after completion of job1), the job1 ends.

D. Comparison of Default and RAS scheduler:

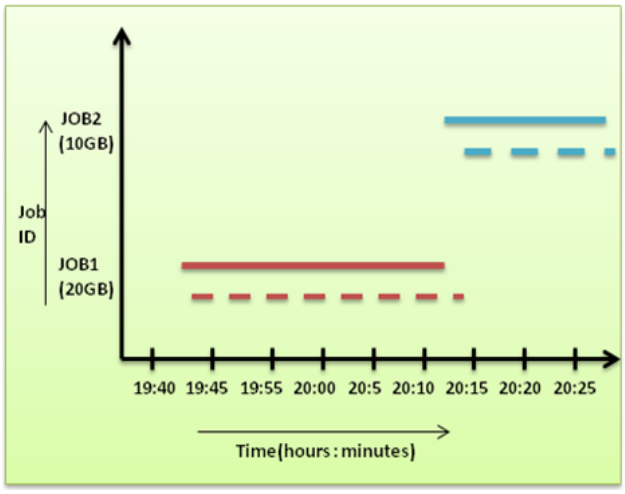


Figure.11 Default Execution

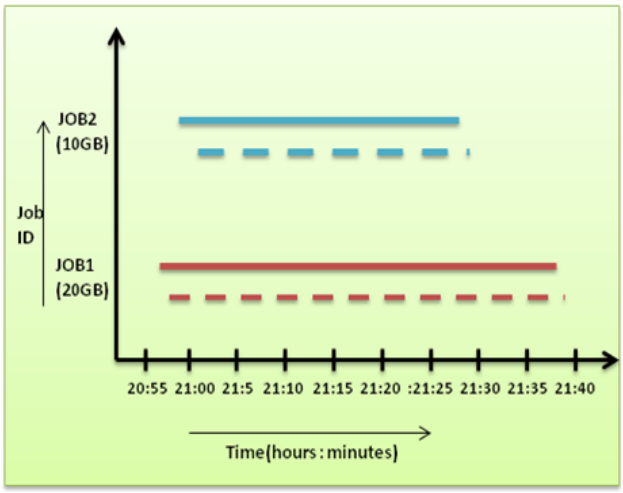


Figure.12RAS Execution

From the graphs plotted (Fig.11),it is observed that, in case of default scheduler (FIFO), only after completion of Job1(20GB), Job2(10GB) is being started. Thus, this demonstrates the fact that even though the resources are available, they are not being utilized.

Existing MapReduce schedulers define a static number of slots to represent the capacity of a cluster, creating a fixed number of execution slots per machine. This abstraction works for homogeneous workloads, but fails to capture the different resource requirements of individual jobs in multi-user environments. But in the case of RAS, technique leverages job profiling information to dynamically adjust the number of slots on each machine, as well as workload

placement across them, to maximize the resource utilization of the cluster.

From the RAS graph plotted (Fig.12) we can infer that both Job1(20GB) and Job1(10GB) executes simultaneously, thus maximising the resource utilization for the MapReduce clusters. The experiments which we conducted on RAS and Default scheduler we notice that RAS will take less time when compared to Default scheduler .we notice upto 25% of improvement in Execution time.

VI. CONCLUSION & FUTURE SCOPE

In this paper we have implemented the Resource-aware Adaptive Scheduler, RAS, which introduces a novel resource management and job scheduling scheme for MapReduce. RAS is capable of improving resource utilization and job perfor- mance. The cornerstone of our scheduler is a resource model] based on a new resource abstraction, namely `job slot'. This model allows for the formulation of a placement problem which RAS solves by means of a utility-driven algorithm. This algorithm in turn provides our scheduler with the adaptability needed to respond to changing conditions in resource demand and availability.

The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions. Profiling of MapReduce jobs that run periodically on data with similar characteristics is an easy task, which has been used by many others in the community in the past. RAS pioneers a novel technique for scheduling reduce tasks by incorporating them into the utility function driving the scheduling algorithm. It works in most circumstances, while in some others it may need to rely on preempting reduce tasks (not implemented in the current prototype) to release resources for jobs with higher priority. Managing reduce tasks in this way is not possible due to limitations in Hadoop and hence it affects all existing schedulers.

In RAS we consider three resource capacities: CPU, memory and I/O. In our experiments we considered mainly on CPU resource and I/O. It can be extended easily to incorporate network infrastructure bandwidth and storage capacity of the TaskTrackers. Nevertheless, network bottlenecks resulting from poor placement of reduce tasks cannot be addressed by RAS without additional monitoring and prediction capabilities.

VII. REFERENCES

[1]. Improving Resource Utilization in a Heterogeneous Cloud Environment Hsin-Yu Shih Department of Electronic Engineering National Taiwan University of Science and Technology Taipei, Taiwan M9902105@mail.ntust.edu.tw, APCC 2012

[2]. J. Dean and S. Ghemawat, \MapReduce: Simpli_ed data processing on large clusters," in OSDI'04, San Francisco, CA, December 2004, pp. 137{150.

[3]. Hadoop MapReduce. http://hadoop.apache.org/mapreduce/.

[4]. A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, \Data warehousing and analytics infrastructure at facebook," in Proceedings of the 2010 international conference on Management of data, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1013{1020.

[5]. G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, \Reining in the outliers in map-reduce clusters using mantri," in OSDI'10. Berkeley, CA, USA: USENIX Asoc., 2010, pp. 1{16.

[6]. J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, \Performance-driven task co-scheduling for MapReduce environments," in Network Operations and Management Symposium, NOMS. Osaka, Japan: IEEE, 2010, pp. 373{380.

[7]. Resource-aware Adaptive Scheduling for MapReduce Clusters - Jordda Polo, Claris astillo,David Carrera, Yolanda Becerra, Ian Whalley, MalgorzataSteinder, Jordi Torres, and Eduard Ayguade.

[8]. Scheduling and Energy Efficiency Improvement Techniques for Hadoop Map-reduce: State of Art and Directions for Future Research - NidhiTiwari, Department of Computer Science and Engineering, Indian Institute of Technology, Mumbai.

[9]. A Comparative review of job scheduling for MapReduce-DongjinYoo, Kwang Mong Sim - Published in Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference.