Proceedings of the 2013 Conference of the Computer Measurement Group, San Diego, CA, November 5-8, 2013.

THE ANATOMY OF MAPREDUCE JOBS, SCHEDULING, AND PERFORMANCE CHALLENGES

SHOUVIK BARDHAN DANIEL A. MENASCÉ

DEPT. OF COMPUTER SCIENCE, MS 4A5
VOLGENAU SCHOOL OF ENGINEERING
GEORGE MASON UNIVERSITY
FAIRFAX, VA 22030, USA
SBARDHAN@GMU.EDU MENASCE@GMU.EDU

Abstract

Hadoop is a leading open source tool that supports the realization of the Big Data revolution and is based on Google's MapReduce pioneering work in the field of ultra large amount of data storage and processing. Instead of relying on expensive proprietary hardware, Hadoop clusters typically consist of hundreds or thousands of multi-core commodity machines. Instead of moving data to the processing nodes, Hadoop moves the code to the machines where the data reside, which is inherently more scalable. Hadoop can store a diversity of data types such as video files, structured or unstructured data, audio files, log files, and signal communication records. The capability to process a large amount of diverse data in a distributed and parallel fashion with built-in fault tolerance, using free software and cheap commodity hardware makes a very compelling business case for the use of Hadoop as the Big Data platform of choice for most commercial and government organizations. However, making a MapReduce job that reads in and processes terabytes of data spanning tens of hundreds of machines complete in an acceptable amount of time can be challenging as illustrated here. This paper first presents the Hadoop ecosystem in some detail and provides details of the MapReduce engine that is at Hadoop's core. The paper then discusses the various MapReduce schedulers and their impact on performance.

Keywords: *Big Data, Hadoop Distributed File System, Performance, Scheduler, Hadoop Map/Reduce, YARN, Corona.*

1 Introduction

Hadoop as it is offered today consists of close to half a million lines of Java code and has hundreds of tunable parameters [13]. Configuring Hadoop's parameters for optimal performance for every kind of application is not a tenable goal. But, a deep understanding of the different phases of a MapReduce job allows developers and operations personnel the opportunity to better tune a Hadoop cluster and extract close-to-desired performance.

Hadoop has two main components at its core: the Distributed File System (DFS) and the MapReduce framework. MapReduce can solve a wide range of computational problems. A shared foundation for a number of systems like Pig [26], Hive [17], Accumulo [1] is provided by Hadoop so that these systems do not have to create their complex and distributed file processing storage and the accompanying methods to retrieve data. Hadoop is still very much MapReduce-based and, therefore, understanding the inner-working of the MapReduce framework is of paramount importance. Just the sheer number of configuration parameters makes it difficult to tune a MapReduce system. Besides, the current Hadoop architecture has potential performance drawbacks for very large clusters because of its architecture of a single job tracker per job. The Hadoop community is working to resolve these issues and we discuss the main ones in this paper.

Section 2 describes the Hadoop ecosystem and introduces MapReduce components through an example. Section 3 delves deeper into the different phases of a MapReduce job. Section 4 discusses the scheduling challenges of a MapReduce cluster and section 5 discusses a way to model some aspects of a MapReduce job. Section 6 discusses some recent work by Yahoo and Facebook to improve the overall performance and scalability issues in current Hadoop. Section 7 discusses some other relevant work. Section 8 provides some concluding remarks .

2 Hadoop Ecosystem and MapReduce

There is an extensive list of products and projects that either extend Hadoop's functionality or expose some existing capability in new ways. For example, executing SQL-like queries on top of Hadoop has spwaned several products. Facebook started this whole movement when it created HiveQL, an SQL-like query language. HBase [14] and Accumulo [1] are both NoSQL databases (i.e., semi-structured table storage) built on top of Hadoop. It is very common to see users writing MapReduce jobs that fetch data from and write data into a NoSQL storage like HBASE or Accumulo. Finally an Apache project called Pig [26] provides an abstraction layer with the help of scripts in the language Pig Latin, which are translated to MapReduce jobs. Other examples of projects built on top of Hadoop include Apache Sqoop [29], Apache Oozie [24], and Apache Flume [10]. Figure 1 shows some of the products built upon and that complement Hadoop.

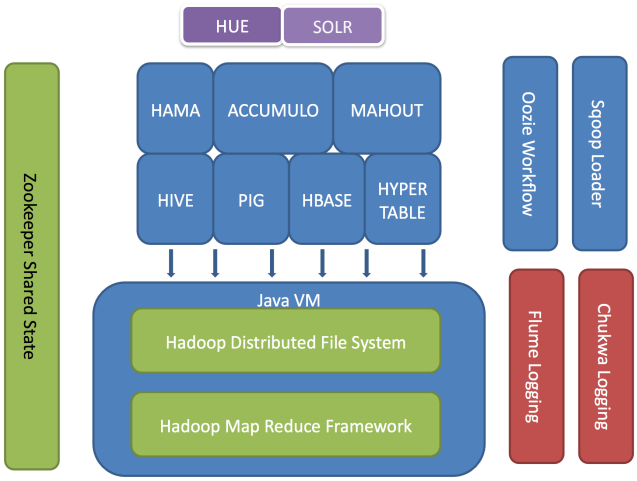

Figure 1: Hadoop ecosystem with multitude of products.

3 Anatomy of MapReduce

The details of MapReduce internals given in this section come from many different sources such as [23, 25, 33, 30] and from the experience of the authors of this paper. The MapReduce programming model consists of a map <k1; v1> function and a reduce <k2; list(v2)> function. Users implement their own processing logic by specifying a custom map() and reduce() function written in a general-purpose programming language such as Java or Python. The map <k1; v1> function is invoked for every key-value pair <k1; v1> in the input data to output zero or more key-value pairs of the form <k2; v2>. The reduce <k2; list(v2)> function is invoked for every unique key k2 and corresponding values list(v2) from the map output. The function reduce <k2; list(v2)> outputs zero or more key-value pairs of the form <k3; v3>.

The MapReduce programming model also allows other functions such as partition(k2), for controlling how the map output key-value pairs are partitioned among the reduce tasks, and combine <k2; list(v2)>, for performing partial aggregation on the map side. The keys k1, k2, and k3 as well as the values v1, v2, and v3 can be of different and arbitrary types. Figure 2 shows the data flow through the map and reduce phases. We discuss the input splits and the different phases of a MapReduce job in greater detail later in this section.

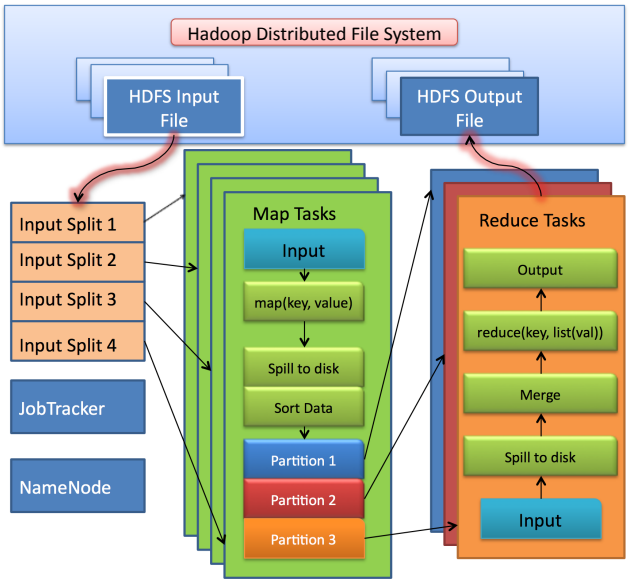

Figure 2: Hadoop MapReduce Data Flow

A Hadoop MapReduce cluster employs a master-slave architecture where one master node (known as JobTracker) manages a number of worker nodes (known as the TaskTrackers). Hadoop launches a MapReduce job by first splitting (logically) the input dataset into multiple data splits. Each map task is then scheduled to one TaskTracker node where the data split resides. A Task Scheduler is responsible for scheduling the execution of the tasks as far as possible in a data-local manner. A few different types of schedulers have been already developed for the MapReduce environment.

From a bird's eye view, a MapReduce job is not all that complex because Hadoop hides most of the complexity of writing parallel programs for a cluster of computers. In a Hadoop cluster, every node normally starts multiple map tasks (many times depending on the number of cores a machine has) and each task will read a portion of the input data in a sequence, process every row of the data and output a <key, value> pair. The reducer tasks in turn collect the keys and values outputted by the mapper tasks and merge the identical keys into one key and the different map values into a collection of values. An individual reducer then will work on these merged keys and the reducer task outputs data of its

choosing by inspecting its input of keys and associated collection of values. The programmer needs to supply only the logic and code for the map() and the reduce() functions. This simple paradigm can solve surprisingly large types of computational problems and is a keystone of the Big Data processing revolution.

In a typical MapReduce job, input files are read from the Hadoop Distributed File System (HDFS). Data is usually compressed to reduce file sizes. After decompression, serialized bytes are transformed into Java objects before being passed to a user-defined map() function. Conversely, output records are serialized, compressed, and eventually pushed back to HDFS. However, behind this apparent simplicity, the processing is broken down into many steps and has hundreds of different tunable parameters to fine-tune the job's running characteristics. We detail these steps starting from when a job is started all the way up to when all the map and reduce tasks are complete and the JobTracker (JT) cleans up the job.

3.1 Startup Phase

Job submission is the starting point in the life of a MapReduce job. A Hadoop job is submitted on a single machine and the job needs to be aware of the addresses of the machines where the Hadoop NameNode (a single master daemon in the Hadoop cluster) and JobTracker daemons are running. The framework will first store any resources (e.g., java jar and configuration files) that must be distributed in HDFS. These are the resources provided via various parameters on the command-line arguments, as well as the JAR file indicated as the job JAR file. This step is executed on the local machine sequentially. The XML version of the job configuration data is also stored in HDFS. The framework will then examine the input data set, using the InputFormat class specified in the job setup to determine which input files must be passed whole to a task and which input files may be split across multiple tasks. The framework uses a variety of parameters to determine how map tasks need to be executed. InputFormat details may override a parameter like *$mapred.map.tasks* (which is a job parameter to signify the number of map tasks the user wants), for example a particular input format may force the splits to be made by line count.

In a Hadoop cluster, many times the number of slots for tasks is connected to the number of CPU cores available on the worker nodes. The ideal split size is one file system block size or a multiple, as this allows the framework to attempt to provide data locally for the task that processes the split. The upshot of this process is a set of input splits that are each tagged with information about which machines have local copies of the split data. The splits are sorted in size order so that the largest splits are executed first. The split information and the job configuration information are passed to the JobTracker for execution via a job information file that is stored in HDFS.

A typical MapReduce job main class (WordCount as an example) snippet lines are given below.

```
public static void main(String[] args) throws
    Exception {
        Configuration conf = new Configuration();
        String[] otherArgs =
            new GenericOptionsParser(conf, args).
                getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: wordcount <
                in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class)
            ;
        FileInputFormat.addInputPath(job, new Path
            (otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new
            Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ?
            0 : 1);
  }
```

The Job and its companions classes such as JobClient are responsible for submitting a MapReduce job. Having submitted the job, waitForCompletion() polls the job's progress once a second, and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by the JobClient's submitJobInternal() method does the following in brief:

- Asks JobTracker for a new job ID (by calling getNewJobId() on JobTracker).
- Checks the output specification of the job. For example, an error is thrown and job submission is aborted if the output directory has not been specified or if it already exists.
- Computes the input splits for the job. In case of an error, the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file and the computed input splits, to the JobTracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the *$mapred.submit.replication* property, which defaults to 10) so that there are sufficient number of copies across the cluster for the TaskTrackers to access when they run tasks for the job.

· Tells the JobTracker that the job is ready for execution (by calling submitJob() on JobTracker).

When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler picks it up and initializes it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the status of the job and its progress.

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the JobClient from the shared filesystem. It then creates one map task for each split. The number of reduce tasks to create is determined by the *$ mapred.reduce.tasks* property in the JobConf object, which is set by the setNumReduceTasks() method, and the scheduler simply creates this number of reduce tasks to be run. Tasks are given IDs at this time.

TaskTrackers run a simple loop that periodically sends heartbeat method calls to the JobTracker. There are as many TaskTrackers in a Hadoop cluster as the number of worker nodes (in the 100s or even 1000s). Heartbeats tell the JobTracker that a TaskTracker is alive, but they also double as a carrier for messages. As part of the heartbeat, a TaskTracker will indicate whether it is ready to run a new task, and if it is, the JobTracker will allocate a task, which it communicates to the TaskTracker using the heartbeat return value. The current architecture of a single JobTracker communicating with hundreds of TaskTrackers is a source of performance issues on huge Hadoop clusters today. This paper details some alternative designs the Hadoop community is working on in later sections of this paper.

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 file system to the TaskTracker's file system. It also copies any files needed from the distributed cache by the application to the local disk. Secondly, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Thirdly, it creates an instance of the TaskRunner object to run the task. The TaskTracker now launches a new Java Virtual Machine (JVM) to run each task so that any bugs in the user-defined map and reduce functions do not affect the TaskTracker (by causing it to crash or hang, for example). It is however possible to reuse the JVM between tasks. The child process communicates with its parent through the umbilical interface (previously mentioned heartbeat). This way it informs the parent of the task's progress every few seconds until the task is complete. The MapReduce framework can be configured with one or more job queues, depending on the scheduler it is configured with. While some schedulers work with only one queue, others support multiple queues. Queues can be configured with various properties.

The TaskStatus class declares the enumeration for the various statuses of a MapReduce job. The different phases are STARTING, MAP, SHUFFLE, SORT, REDUCE and CLEANUP.

Algorithm 1 below shows the main loop of the JobTracker. We have already discussed in some detail the STARTING phase of a job. We will now delve into the other phases.

Algorithm 1 JobTracker Main Flow

```
loop
  waitForMessage();
  if msg == HEARTBEAT then
    startMapTasks(); {Initialize Map Tasks}
    startReduceTasks(); {Initialize Reduce Tasks}
  else if msg == MAP_TASK_DONE then
    NotifyReduceTasks(); {Intermediate Data Fetch}
  else if msg == REDUCE_TASK_DONE then
    if allReduceTasksComplete then
      SignalJobComplete();
    end if
  else
    houseKeepingWork();
  end if
end loop
```

Figure 3 shows the calls between the different framework pieces when a MapReduce job is started with the client invocation up to the point a new JVM is created to run the task. The cleanup activities are not shown in this sequence diagram.

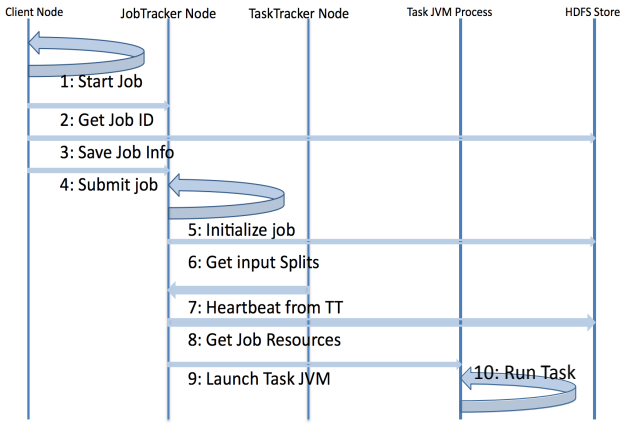

Figure 3: Call sequence for a Hadoop job

3.2 Map Phase

Let us consider the map phase in detail. A MapReduce job exists to read input data splits through a number of map tasks. The Job object created the information

about all the splits and wrote a file with the split information and many other job related info so that the JobTracker has access to that file. The JobTracker is actually responsible for starting the tasks for the submitted job. Below we describe job input and job output in some detail.

InputFormat describes the input-specification for a MapReduce job. The MapReduce framework relies on the InputFormat of the job to

- Validate the input-specification of the job.
- Split-up the input file(s) into logical InputSplit instances, each of which is then assigned to an individual Mapper.
- Provide the RecordReader implementation used to create input records from the logical InputSplit for processing by the Mapper.

Most MapReduce programs implement their own InputFormat class but the default one is TextInputFormat that breaks files into lines. Keys are the position in the file, and values are the line of text (delimited by either line feed and/or carriage return). InputSplit represents the data to be processed by an individual Mapper. FileSplit is the default InputSplit. It sets *$map.input.file* to the path of the input file for the logical split. RecordReader reads <key, value> pairs from an InputSplit. Typically, the RecordReader converts the byte-oriented view of the input, provided by the InputSplit, and presents a record-oriented view to the mapper() implementations for processing. RecordReader thus assumes the responsibility of processing record boundaries and presents the tasks with keys and values appropriate for the domain.

OutputFormat describes the output-specification for a MapReduce job. The MapReduce framework relies on the OutputFormat of the job to:

- Validate the output-specification of the job; for example, check that the output directory does not already exist.
- Provide the RecordWriter implementation used to write the output files of the job. Output files are stored in the Hadoop FileSystem. TextOutputFormat is the default OutputFormat.

If the HDFS directory pointed to by the parameter *$mapreduce.job.dir* is listed while a job is in the process of executing on a Hadoop cluster, we would see the following files.

```
[mc2233]# hadoop fs -ls /tmp/hadoop/mapred/staging
    /
            condor/.staging/job_201303040610_0423
Found 4 items
job_201303040610_0423/job.jar
job_201303040610_0423/job.split
job_201303040610_0423/job.splitmetainfo
job_201303040610_0423/job.xml
```

The JobSplit class groups the fundamental classes associated with reading/writing splits. The split information is divided into two parts based on the consumer of the information. These two parts are the split meta information and the raw split information. The first part is consumed by the JobTracker to create the tasks' locality data structures. The second part is used by the maps at runtime to know how to setup the InputFormat/RecordReader so that data can be read and fed into the mapper() method. These pieces of information are written to two separate files as is shown above in the directory listing.

Any split implementation extends the base abstract class - InputSplit, defining a split length and locations of the split. A split length is the size of the split data (in bytes), while locations is the list of node names where the data for the split would be local. Split locations are a way for a scheduler to decide on which particular machine to execute this split.

Executing a map task on the node where its own data exists is an important aspect since that means data does not have to be transported across the network. This type of map task is called a data-local map task. Locality can mean different things depending on storage mechanisms and the overall execution strategy. In the case of HDFS, a split typically corresponds to a physical data block size and locations on a set of machines (with the set size defined by a replication factor) where this block is physically located.

Now we pick up at the point where TaskTracker has started a child process (a class called Child) which has a main() method. This class is responsible for starting the map task. Not surprisingly, the name of the class that encapsulates the map task is called MapTask (it extends Task as does the class ReduceTask), which handles all aspects of running a map for the job under consideration. While a map is running, it is collecting output records in an in-memory buffer called MapOutputBuffer. If there are no reducers, a DirectMapOutputCollector is used, which makes writes immediately to disk. The total size of this in-memory buffer is set by the *$io.sort.mb* property and defaults to 100 MB. Out of these 100 MB, *$io.sort.record.percent* are reserved for tracking record boundaries. This property defaults to 0.05 (i.e. 5% which means 5 MB in the default case) Each record to track takes 16 bytes (4 integers of 4 bytes each) of memory which means that the buffer can track 327680 map output records with the default settings. The rest of the memory (104,857,600 bytes - (16 bytes $\times$ 327,680) = 99,614,720 bytes) is used to store the actual bytes to be collected (in the default case this will be about 95 MB). While map outputs are collected, they are stored in the remaining memory and their location in the in-memory buffer is tracked as well. Once one of these two buffers reaches a threshold specified by *$io.sort.spill.percent*, which defaults to 0.8 (i.e., 80%), the buffer is flushed to disk. For the actual data this value is

79,691,776 (0.8 $\times$ 99,614,720) and for the tracking data the threshold is 262,144 (0.8 $\times$ 327,680).

All of this spilling to disk is done in a separate thread so that the map can continue running. That is also the reason why the spill begins early (when the buffer is only 80% full) so it does not fill up before a spill is finished. If one single map output is too large to fit into the in-memory buffer, a single spill is done for this one value. A spill actually consists of one file per partition, meaning one file per reducer.

After a map task has finished, there may be multiple spills on the TaskTracker. Those files have to be merged into one single sorted file per partition which is then fetched by the reducers. The property *$io.sort.factor* indicates how many of those spill files will be merged into one file at a time. The lower the number the more passes will be required to arrive at the goal. The default is set to 100. This property can make a pretty huge difference if the mappers output a lot of data. Not much memory is needed for this property but the larger it is the more open files there will be. To determine such a value, a few MapReduce jobs expected in production should be run and one should carefully monitor the log files.

So, at this point the input file was split and read, the mapper() function was invoked for each <key, value> pair. The mapper(), after processing each pair, outputted another <key, value> pair which was collected by the framework in a memory buffer, sorted and partitioned. If memory got full, then they were spilled to the disk and merged into files. The number of files is the same as the number of different partitions (number of reducers). Now, the reducer tasks are coming into action and we will see how reducer tasks will obtain the map output data.

3.3 Shuffle Phase

The shuffle phase is one of the steps leading up to the reduce phase of a MapReduce job. The main actors of the shuffle phase are the reducer tasks and the TaskTrackers, which are holding on to the map output files. In other words, a map output file is stored on the local disk of the TaskTracker that ran the map task (note that although map outputs are always written to the local disk of the map TaskTracker, reduce outputs may not be), but now the address of this TaskTracker that finished the map task is needed by the TaskTracker that is about to run the reduce task for the partition. Furthermore, the reduce task needs the map output for its particular partition from several map tasks across the cluster. The map tasks may finish at different times, so the reduce tasks start copying their outputs (also called shuffling the output) as soon as each completes. This is also known as the copy phase of the reduce task. The reduce task has a small number of copier threads (5 by default) so that it can fetch map outputs in parallel. Though the default is 5 threads, this number can be changed by setting the *$mapred.reduce.parallel.copies* property.

Since the reducers have to know which TaskTrackers to fetch map output from, as map tasks complete successfully, they notify their parent TaskTracker of the status update, which in turn notifies the JobTracker. These notifications are transmitted over the heartbeat communication mechanism described earlier. Therefore, for a given job, the JobTracker knows the mapping between map outputs and TaskTrackers. A thread in the reducer periodically asks the JobTracker for map output locations until it has retrieved them all. TaskTrackers do not delete map outputs from disk as soon as the first reducer has retrieved them, as the reducer tasks may fail. Instead, they wait until they are told to delete them by the JobTracker, which is after the job has completed.

The map outputs then are copied to the reducer's memory if they are small enough (the buffer size is controlled by *$mapred.job.shuffle.input.buffer.percent*, which specifies the proportion of the heap to use for this purpose); otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size (controlled by *$mapred.job.shuffle.merge.percent*), or reaches a threshold number of map outputs (*$mapred.inmem.merge.threshold*), it is merged and spilled to disk.

3.4 Reduce Phase

Let us turn now to the reduce part of the process. As the copies accumulate on disk, a background thread merges them into larger sorted files. This saves some time merging later on in the process. Note that any map outputs that were compressed (by the map task) have to be decompressed in memory in order to perform a merge on them. When all the map outputs have been copied, the reduce task moves into the sort phase (which should actually be called the merge phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 100 map outputs, and the merge factor was ten (the default, controlled by the *$io.sort.factor* property, just like in the map's merge), then there would be 10 rounds. Each round would merge ten files into one, so at the end there would be 10 intermediate files.

Rather than have a final round that merges these ten files into a single sorted file, the merge saves a trip to disk and additional I/O by directly feeding the reduce function in what is the last phase: the reduce phase. This final merge can come from a mixture of in-memory and on-disk segments. The number of files merged in each round is actually more subtle than the above explanation suggests. The goal is to merge the minimum number of files to get to the merge factor for the final round.

During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of

this phase is written directly to the output filesystem, typically HDFS. In the case of HDFS, since the TaskTracker node is also running a DataNode, the first block replica will be written to the local disk.

Among the many steps described above in starting and running of a MapReduce program, some of the responsibilities lie with the programmer of the job. Hadoop provides the remaining facilities. Most of the heavy lifting is done by the framework and Fig. 4 shows the different steps performed by the users and how they map to the various steps provided by the framework itself.

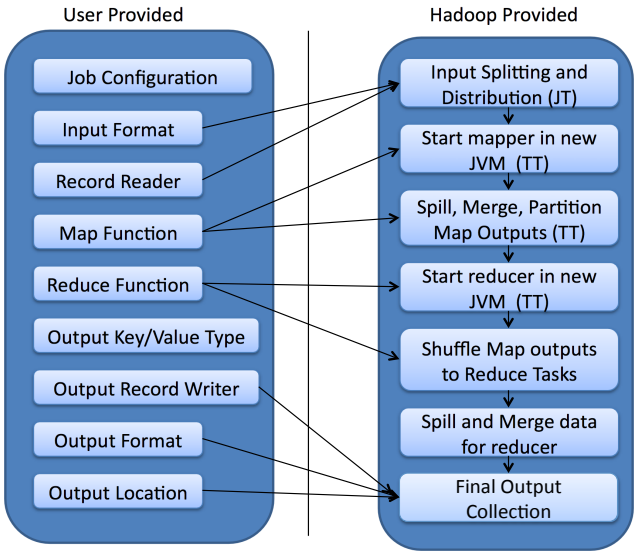

Figure 4: MapReduce Steps - Who does what

3.5 Shutdown Phase

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours or even several days to run. Because this is a significant length of time, it is important for the user to obtain 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, failed), the progress 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 and they are communicated back to the client. When a task is running, it keeps track of its progress, that is, the proportion of the task completed. For map tasks, this is the proportion of the input that has been processed. For reduce tasks the system can 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 reducer.

Measurement of progress is somewhat nebulous but it tells Hadoop that a task is doing something. For example, a task writing output records is making progress, even though it cannot be expressed as a percentage of the total number that will be written, since the latter figure may not be known, even by the task producing the output.

Progress reporting is important so that Hadoop does not fail a task that is making progress. All of the following operations constitute progress:

- Reading an input record (in a mapper or reducer)
- Writing an output record (in a mapper or reducer)
- Setting the status description on a reporter (using the Reporter's setStatus() method)
- Incrementing a counter (using the Reporter's incrCounter() method)
- Calling the Reporter's progress() method

Tasks also have a set of counters that count various events as a task runs, either those built into the framework, such as the number of map output records written, or counters defined by users. If a task reports progress, it sets a flag to indicate that the status change should be sent to the TaskTracker. The flag is checked in a separate thread every few seconds, and if set it notifies the TaskTracker of the current task status. Meanwhile, the TaskTracker is sending heartbeats to the JobTracker every five seconds (this is a minimum, as the heartbeat interval is actually dependent on the size of the cluster: for larger clusters, the interval is longer and a source of much consternation since this value can delay the time to start up a job's task(s)), and the status of all the tasks being run by the TaskTracker is sent in the call. Counters are sent less frequently than every five seconds, because they can be relatively high-bandwidth. The JobTracker combines these updates to produce a global view of the status of all the jobs being run and their constituent tasks. Finally, the JobClient receives the latest status by polling the JobTracker every second. Clients can also use JobClient's getJob() method to obtain a RunningJob object instance, which contains all of the status information for the job.

When the JobTracker receives a notification that the last task of a job is complete, it changes the status of the job to "successful". Then, when the JobClient polls for status, the JobClient learns that the job has completed successfully and it prints a message to tell the user and then returns from the runJob() method. The JobTracker also sends a HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the job.end.notification.url property. Lastly, the JobTracker cleans up its working state for the job, and instructs TaskTrackers to do the same (the intermediate output is deleted and other such cleanup tasks are performed).

4 MapReduce Scheduling Challenges

Hundreds of jobs (small/medium/large) may be present on a Hadoop cluster for processing at any given time. How the map and reduce tasks of these jobs are scheduled has an impact on the completion time and consequently on the QoS requirements of these jobs. Hadoop uses a FIFO scheduler out of the box. Subsequently, two more schedulers have been developed. Firstly, Facebook developed the Fair Scheduler which is meant to give fast response time for small jobs and reasonable finish time for production jobs. Secondly, Capacity Scheduler was developed by Yahoo and this scheduler has named queues in which jobs are submitted. Queues are allocated a fraction of the total computing resource and jobs have priorities. It is evident that no single scheduling algorithm and policy is appropriate for all kinds of job mixes. A mix or combination of scheduling algorithms may actually be better depending on the workload characteristics.

Essentially, Hadoop is a multi-tasking software product that can process multiple data-sets for multiple jobs for many users at the same time. This means that Hadoop is also concerned with scheduling jobs in a way that makes optimum usage of the resources available on the compute cluster. Unfortunately, Hadoop started off not doing a good job of scheduling in an efficient fashion and even today its scheduling algorithms and schedulers are not all that sophisticated.

At the inception of Hadoop, five or so years ago, the original scheduler was a first-in first-out (FIFO) scheduler woven into Hadoop's JobTracker. Even though it was simple, the implementation was inflexible and could not be tailored. After all, not all jobs have the same priority and a higher priority job is not supposed to languish behind a low priority long running batch job. Around 2008, Hadoop introduced a pluggable scheduler interface which was independent of the JobTracker. The goal was to develop new schedulers which would help optimize scheduling based on particular job characteristics. Another advantage of this pluggable scheduler architecture is that now greater experimentation is possible and specialized schedulers are possible to cater to an ever increasing types of Hadoop MapReduce applications.

Before it can choose a task for the TaskTracker, the JobTracker must choose a job to select the task from. 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 example, a TaskTracker may be able to run two map tasks and two reduce tasks simultaneously. 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.

Based on published research [31, 3], it is very evident that a single scheduler is not adequate to obtain the best possible QoS out of a Hadoop MapReduce cluster subject to a varying workload. Also, we have seen that almost none of the schedulers consider the contention placed on the nodes due to the running of multiple tasks.

5 MapReduce Performance Challenges

Hundereds of thousands of jobs of varying demands on CPU, I/O and network are executed on Hadoop clusters consisting of several hundred nodes. Tasks are scheduled on machines, in many cases with 16 or more cores each. Short jobs have a different completion time requirement than long jobs and similarly production level high priority jobs have a different quality of service requirement compared to adhoc query type jobs. Predicting the completion time of Hadoop MapReduce jobs is an important research topic since for large enterprises, correctly forecasting the completion time and an efficient scheduling of Hadoop jobs directly affects the bottom line. A plethora of work is going on in the field of Hadoop MapReduce performance. We briefly talk about a few prominent recent ones which are most relevant to this paper. In his 2011 technical report [16], Herodotou describes in detail a set of mathematical performance models for describing the execution of a MapReduce job on Hadoop. The models can be used to estimate the performance of MapReduce jobs as well as to find the optimal configuration settings to use when running the jobs. A set of 100 or so equations calculate the total cost of a MapReduce job based on different categories of parameters - Hadoop parameters, job profile statistics and a set of parameters that define the I/O, CPU, and network cost of a job execution. In a 2010 paper, Kavulya et al. [18] analysed ten months worth of Yahoo MapReduce logs and used an instance-based learning technique that exploits temporal locality to predict job completion times from historical data. Though the thrust of the paper is essentially analyzing the log traces of Yahoo, this paper extends the instance based (nearest-neighbor) learning approaches. The algorithm consists of first using a distance-based approach to find a set of similar jobs in the recent past, and then generating regression models that predict the completion time of the incoming job from the Hadoop-specific input parameters listed in the set of similar jobs. In a 2010 paper [31], Verma et al. design a performance model that for a given job (with a known profile) and its SLO (soft deadline), estimates the amount of resources required for job completion within the deadline. They also implement a Service Level Objective based scheduler that determines job ordering and the amount of resources to allocate for meeting the job deadlines.

In [3], the authors describe an analytic model they built to predict a MapReduce job's map phase completion time. The authors take into account the effect of contention at the compute nodes of a cluster and use

a closed Queuing Network model [22] to estimate the completion time of a job's map phase based on the number of map tasks, the number of compute nodes, the number of slots on each node, and the total number of map slots available on the cluster.

6 MapReduce - Looking Ahead

The discussion in the preceding sections was all based on current Hadoop (also known as Hadoop V1.0). In clusters consisting of thousands of nodes, having a single JobTracker manage all the TaskTrackers and the entire set of jobs has proved to be a big bottleneck. The architecture of one JobTracker introduces inherent delays in scheduling jobs which proved to be unacceptable for many large Hadoop users.

6.1 Facebook's Corona

The bulk of the workload on the Hadoop clusters in Facebook are Hive [17] queries. These queries translate into small Hadoop jobs and it is important that the latency experienced by these jobs be small. Therefore, small job latency is crucial for Facebook's environment. Facebook noticed that the single JobTracker is the bottleneck and many times the startup time for a job is several tens of seconds [7]. The JobTracker also could not handle its dual responsibilities of managing the cluster resources and the scheduling of all the user jobs adequately. At peak load at Facebook, cluster utilization dropped precipitously due to scheduling overhead using the standard Hadoop framework [7]. Along with the problems described above, Facebook also found that the polling from TaskTrackers to the JobTracker during a periodic heartbeat is a big part of why Hadoop's scheduler is slow and has scalability issues [7]. And lastly, the slot based model is a problem when the slot configuration does not match the job mix. As a result, during upgrades to change the number of slots on node(s), all jobs were killed, which was unacceptable.

Facebook decided that they needed a better scheduling framework that would improve this situation by bringing better scalability and cluster utilization, by lowering latency for small jobs and by scheduling based on actual task resource requirements rather than a count of map and reduce tasks [28].

To address these issues, Facebook developed Corona [7], a new scheduling framework that separates cluster resource management from job coordination. Corona introduces a cluster manager whose only purpose is to track the nodes in the cluster and the amount of free resources. A dedicated job tracker is created for each job, and can run either in the same process as the client (for small jobs) or as a separate process in the cluster (for large jobs). One major difference from the previous Hadoop MapReduce implementation is that Corona uses push-based, rather than pull-based, scheduling. After the cluster manager receives resource requests from the job tracker, it pushes the resource grants back to the job tracker. Also, once the job tracker gets resource grants, it creates tasks and then pushes these tasks to the task trackers for running. There is no periodic heartbeat involved in this scheduling, so the scheduling latency is minimized.

The cluster manager does not perform any tracking or monitoring of the job's progress, as that responsibility is left to the individual job trackers. This separation of duties allows the cluster manager to focus on making fast scheduling decisions. Job trackers track only one job each and have less code complexity. This allows Corona to manage a lot more jobs and achieve better cluster utilization. Figure 5 depicts a notional diagram of Corona's main components.

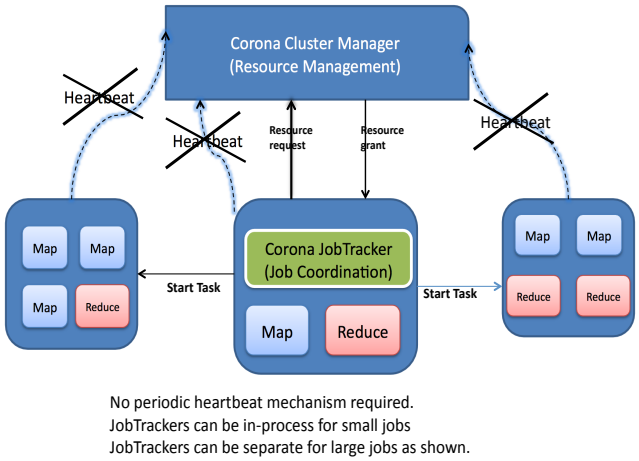

Figure 5: Corona - A new Hadoop

Once they developed Corona, Facebook measured some of these improvements with several key metrics:

Average time to refill slot - This metric gauges the amount of time a map or reduce slot remains idle on a TaskTracker. Given that the scalability of the scheduler is a bottleneck, improvement was observed with Corona when compared with a similar period for MapReduce. Corona showed an improvement of approximately 17% in a benchmark run in the simulation cluster.

Cluster utilization - Cluster utilization also improved along with the refill-slot metric. In heavy workloads during simulation cluster testing, the utilization in the Hadoop MapReduce system topped out at 70%. Corona was able to reach more than 95%.

Scheduling fairness - Resource scheduling fairness is important since it ensures that each of the pools actually gets the resources it expects. The old Hadoop MapReduce system is found to be unfair in its allocation, and a dramatic improvement was seen with the new system.

Job latency - Average job latencies were expected to

go down just by virtue of the new design in Corona. The best metric for measuring this was found to be a latency job that was run every four minutes. It was seen that the latency of the test job was cut in half (25 seconds, down from 50 seconds).

6.2 Yahoo's YARN

Yahoo has been a major player in the Hadoop ecosystem already. MapReduce version-2 was conceived and designed by Yahoo when it was found that JobTracker is a bottleneck in their huge Hadoop clusters. Current Hadoop has indeed reached a scalability limit of around 4,000 nodes. Scalability requirement of 20,000 nodes and 200,000 cores is being asked for and was not possible with the current JobTracker/TaskTracker architecture. YARN [35] stands for Yet Another Resource Negotiator and the next generation of Hadoop (0.23 and beyond, most places are running 0.20 or 0.21 today) and is also called MapReduce2.0 or MRv2. YARN breaks the function of the JobTracker into 2 parts (see Fig. 6 for a block level diagram of YARN's main components). The job lifecycle management portion of JobTracker will be handled by Resource Manager (RM) in YARN clusters which will be responsible for assignment of compute resources to applications and the job life-cycle management portion of the responsibility will be handled per application by the Application Master (AM). The AM is created and lives for the duration of the application (for MapReduce and other kinds of apps) and then exits. There is also a per machine Node Manager (NM) daemon that manages the user processes on the machine. Resource Manager therefore has 2 parts - Applications Manager (RM-ASM) and Scheduler (RM-S). Scheduler is a plugin. Resource Manager talks to Node Manager and Applications Master. It looks like being a MapReduce platform is not the only goal of YARN. Running applications of different kind other than current MapReduce seems to be one of the motivations for the YARN effort. TaskTracker seems to be not present in the YARN framework at all [35].

The Scheduler models only memory in YARN v1.0. Every node in the system is considered to be composed of multiple containers of minimum size of memory (say 512 MB or 1 GB). The ApplicationMaster can request any container as a multiple of the minimum memory size. Eventually, YARN wants to move to a more generic resource model, however, for YARN v1 the proposal is for a rather straightforward model.

The resource model is completely based on memory (RAM) and every node is made up of discreet chunks of memory. Unlike Hadoop, the cluster is not segregated into map and reduce slots. Every chunk of memory is interchangable and this has huge benefits for cluster utilization - currently a well known problem with Hadoop MapReduce is that jobs are bottlenecked on reduce slots and the lack of fungible resources is a severe limiting factor.

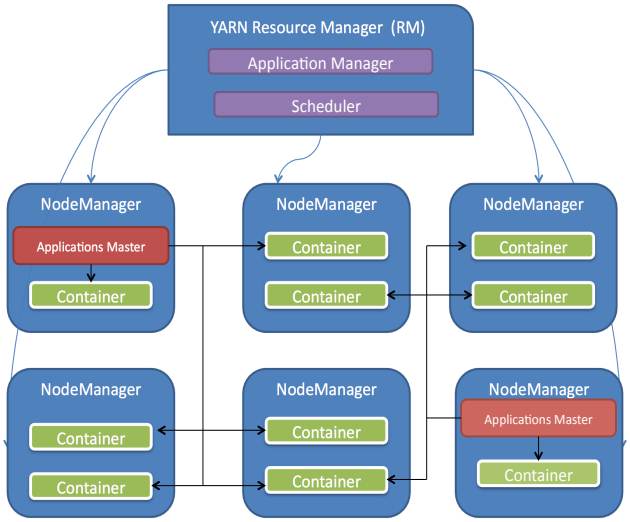

Figure 6: YARN - Yet another new Hadoop

The lifecycle of a Map-Reduce job running in YARN is as follows:

1. Hadoop MR JobClient submits the job to the YARN ResourceManager (ApplicationsManager) rather than to the Hadoop MapReduce JobTracker.
2. The YARN RM-ASM negotiates the container for the MR Application Master with the Scheduler (RM-S) and then launches the MR AM for the job.
3. The MR AM starts up and registers with the RM-ASM.
4. The Hadoop MapReduce JobClient polls the ASM to obtain information about the MR AM and then directly talks to the AM for status, counters, etc.
5. The MR AM computes input-splits and constructs resource requests for all maps and reducers to the YARN Scheduler.
6. The MR AM runs the requisite job setup APIs of the Hadoop MR OutputCommitter for the job.
7. The MR AM submits the resource requests for the map/reduce tasks to the YARN Scheduler (RM-S), gets containers from the RM and schedules appropriate tasks on the obtained containers by working with the Node Manager for each container.
8. The MR AM monitors the individual tasks to completion, requests alternate resource if any of the tasks fail or stop responding.
9. The MR AM also runs appropriate task cleanup code of completed tasks of the Hadoop MR OutputCommitter.

10. Once the entire map and reduce tasks are complete, the MR AM runs the requisite job commit APIs of the Hadoop MR OutputCommitter for the job.

11. The MR AM then exits since the job is complete.

7 Relevant Work

There is significant work focusing on the performance of Hadoop in general and the scheduling of jobs in the Hadoop MapReduce framework. We detail some of the relevant findings in this section. All map to all reduce communication of data at the end of map tasks is called the shuffle phase in a MapReduce job as we have described in this paper. This data can be huge for shuffle heavy MapReduce programs. In [2], the paper proposes a scheme of running aggressive sorts and partial reduce functions concurrently along with the mapper so that shuffle overlap with the map tasks is as high as possible. The authors saw around 20% improvement with this scheme in their implementation over stock Hadoop. CPU contention for these extra tasks are kept on check by running these in low priority. The authors of [5] worked with MapReduce workload traces from Facebook and Cloudera customers in telecommunications, media, retail and e-commerce and examined over a year's worth of data. The inspected jobs moved 1.6 exabytes of data on over 5,000 machines and totalled over two million jobs. Some of the main observations were that five out of seven workloads are only map-only jobs, common job types changed over a year, most jobs handled GB quantity or fewer amounts of data and those workloads are bursty. One key conclusion is that not many jobs had a high shuffle volume and thus it is extremely important that we concentrate on the map phase of Hadoop jobs.

It was observed in [20] that analytic processing clusters experience heterogeneous loads—some processes are CPU intensive and some are I/O or network intensive. A cluster has to provide dynamic provisioning of resources and also a cluster can have heterogeneous machines—some more powerful in terms of CPU or I/O than others. Resource allocation and job scheduling is researched in this work and considered given the fluctuating resource demands. The main strategy that the authors came up with is to divide the cluster into core and accelerator nodes. A cloud driver component, which sits atop the MapReduce environment, adds accelerator nodes as needed based on its knowledge of the type of jobs being seen in the environment. In [21], the author provides a mechanism to handle many nodes going down due to spot price increase in Amazon EC2. The proposed implementation, called "Spot Cloud MapReduce," works well in situations when a group spot machines go offline. Essentially, queues and local DBs are used to shift messages off of machines that can go down and to store checkpoint information about the amount of progress that the map tasks have made running on the instances that could be terminated based on pricing increases.

In [27], the authors contend that a fixed number of slots on the nodes are not efficient for heterogeneous job mixes. They propose Resource-aware Adaptive Scheduler (RAS). The notion of task specific slots is not considered and instead a job specific slot is introduced. The RAS scheduler is based on a resource model and uses a utility function based on desired completion time of the job to drive their scheduling algorithm. There is no mathematically sound performance model backing up this scheduler as far as we can tell. And lastly, in [15], the authors propose a system called *Elasticizer* that can respond to user queries about cluster sizing matters and job configuration. Cluster sizing queries pertaining to the use of Amazon EC2 environment, moving of jobs from test to production can be successfully answered with *Elasticizer*. A pay-as-you-go model in many public clouds make this a very timely work. Multi-tenancy is left as a future topic of study by the authors.

8 Concluding Remarks

We discussed in some detail the way current Hadoop's MapReduce handles the completion of a job through different phases. Hadoop is going to look and feel quite different though once the 2.0 version is released during the summer of 2013. So why did Facebook go ahead and implement Corona instead of getting into YARN and adapt that to their need? Some of the main points are detailed below [28].

Firstly, YARN is still not push based scheduling and thus would not have solved one of the main problems of lowering job latency. Secondly, in Corona, JobTracker and the Job Client can run in the same JVM. In YARN, the JobTracker equivalent has to be scheduled as a separate process and thus extra steps are required to start the job. YARN is more geared towards Capacity scheduler vs. Fair Scheduler for Corona. Next, Corona used Thrift based RPC vs. YARN's protocol buffers. Facebook was not all that interested in working with protocol buffers. Lastly and perhaps most importantly, Facebook already had huge investment in Hadoop and basically had many changes on the 0.20 Hadoop distribution. Adopting YARN, which is 0.23 and consequently essentially a new forked code of Hadoop, would have been very time consuming for Facebook. And as such, Corona is a small codebase, not as diverse from current Hadoop as YARN is.

References

[1] Accumulo, http://accumulo.apache.org/

[2] F. Ahmad, S. Lee, M. Thottethodi, and T.N. Vijaykumar, *MapReduce with communication over-*

lap (MaRCO), J. Parallel and Distributed Computing, Elsevier, 2013.

[3] S. Bardhan and D.A.Menascé, *Queuing Network Models to Predict the Completion Time of the Map Phase of MapReduce Jobs*, CMG Intl. Conf., Las Vegas, NV, Dec. 3-7, 2012.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, ACM Trans. Comput. Syst., Vol. 26, No. 2, June 2008.

[5] Yanpei Chen, Sara Alspaugh, and Randy H. Katz, *Design insights for mapreduce from diverse production workloads*, No. UCB/EECS-2012-17, U.C. Berkeley, Dept. of Electrical Engineering and Computer Science, 2012.

[6] Cloudera, http://www.cloudera.com/

[7] Corona, https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920

[8] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, Proc. Sixth Symp. Operating System Design and Implementation, San Francisco, CA, Dec. 6-8, Usenix, 2004.

[9] J. Dean and S. Ghemawat, *MapReduce: A Flexible Data Processing Tool*, Communications of the ACM, Vol. 53 No. 1, pp. 72-77.

[10] Flume, http://flume.apache.org/

[11] A. Ganapathi et al., *Statistics-driven workload modeling for the Cloud*, Data Engineering Workshops (ICDEW), 2010 IEEE.

[12] A. Ganapathi et al., *Predicting Multiple Performance Metrics for Queries: Better Decisions Enabled by Machine Learning*, Proc International Conference on Data Engineering, 2009.

[13] Hadoop, Documentation and open source release, http://hadoop.apache.org/core/

[14] HBase, http://hbase.apache.org/

[15] Herodotos Herodotou, Fei Dong, and Shivnath Babu, *No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics*, Proc. 2nd ACM Symposium on Cloud Computing, ACM, 2011.

[16] Herodotos Herodotou, *Hadoop performance models*, Cornell University Library, Report Number CS-2011-05, arXiv:1106.0940v1 [cs.DC].

[17] Hive, http://hive.apache.org/

[18] S. Kavulya et. al., *An analysis of traces from a production mapreduce cluster*, 10th IEEE/ACM Intl. Conf. Cluster, Cloud and Grid Computing (CCGrid), 2010.

[19] E. Krevat, T. Shiran, E. Anderson, J. Tucek, J.J. Wylie, and Gregory R Ganger, *Applying Performance Models to Understand Data-Intensive Computing Efficiency*, Carnegie-Mellon University, Parallel Data Laboratory, Technical Report CMU-PDL-10-108, May 2010.

[20] Gunho Lee, Byung-Gon Chun, and Randy H. Katz, *Heterogeneity-aware resource allocation and scheduling in the cloud*, Proc. 3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud, Vol. 11. 2011.

[21] Huan Liu, *Cutting MapReduce cost with spot market*, 3rd USENIX Workshop on Hot Topics in Cloud Computing, 2011.

[22] D.A. Menascé, V.A.F. Almeida, and L.W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, Upper Saddle River, 2004.

[23] http://odbms.org/download/Pro Hadoop Ch. 6.pdf

[24] Oozie, http://oozie.apache.org/

[25] http://answers.oreilly.com/topic/459-anatomy-of-a-mapreduce-job-run-with-hadoop/

[26] Pig, http://pig.apache.org/

[27] Jorda Polo et. al., *Resource-aware adaptive scheduling for mapreduce clusters*, Middleware 2011, Springer Berlin Heidelberg, 2011, pp. 187–207.

[28] http://www.quora.com/Facebook-Engineering/How-is-Facebooks-new-replacement-for-MapReduce-Corona-different-from-MapReduce2

[29] Sqoop, http://sqoop.apache.org/

[30] http://hadoop.apache.org/docs/r0.18.3/mapred_tutorial.html

[31] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell, *ARIA: automatic resource inference and allocation for mapreduce environments*, Proc. 8th ACM Intl. Conf. Autonomic computing, ACM, 2011.

[32] E. Vianna et al., *Modeling the Performance of the Hadoop Online Prototype*, 23rd Intl. Symp. Computer Architecture and High Performance Computing (SBAC-PAD), 26-29 Oct. 2011, pp.152-159.

[33] http://developer.yahoo.com/blogs/hadoop/anatomy-hadoop-o-pipeline-428.html

[34] Hailong Yang et al., *MapReduce Workload Modeling with Statistical Approach*, J. Grid Computing, Volume 10, Number 2 (2012).

[35] YARN, http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html