Proc. 2012 International Computer Measurement Group Conf., Las Vegas, NV, December 3-4, 2012.

Queuing Network Models to Predict the Completion Time of the Map Phase of MapReduce Jobs

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

Big Data processing is generally defined as a situation when the size of the data itself becomes part of the computational problem. This has made divide-and-conquer type algorithms implemented in clusters of multi-core CPUs in Hadoop/MapReduce environments an important data processing tool for many organizations. Jobs of various kinds, which consists of a number of automatically parallelized tasks, are scheduled on distributed nodes based on the capacity of the machines. A key challenge in provisioning such jobs in a Hadoop/MapReduce cluster is to be able to predict their completion times based on various job characteristics. Standard makespan computations that ignore the contention on compute nodes significantly underestimate a job's completion time. This paper proposes a mathematically sound model based on closed Queuing Networks for predicting the execution time of the map phase of a MapReduce job. The model captures contention at compute nodes and parallelism gains due to increased number of slots available to map tasks. Experiments validated the model on a single as well as a 2-node Hadoop environment. We ran experiments for different input split sizes and different map slot sizes to validate our model.

1 Introduction

Big Data processing is generally defined as a situation when the size of the data itself becomes part of the computational problem. A recent trend in handling large volumes of unstructured data generated at a very high rate has made it difficult to work with standard relational database systems. Instead, the path forward seems to rely on massively parallel software running on hundreds or even thousands of computing nodes. Innovative algorithms have afforded the capability to create huge amounts of additional information from the analysis of a single large set of related data. This has allowed one to find important correlations in business analytics, disease prevention, and combat to terrorism and crime, to mention a few.

The federal government announced in late March plans to spend $200 millions of dollars a year on a new "big data initiative" for research and development into technology to "access, store, visualize, and analyze" massive and complicated data sets.

A popular paradigm for handling big data applications is the MapReduce programming model introduced by Google [6, 7]. An open source implementation of MapReduce is available from Apache under the name of Hadoop [8].

Many commercial firms and government organizations use Hadoop and its complimentary products to process vast amounts of data on multi-node commodity hardware. The motivation for performing computations on this kind of platform stems from the fact that Hadoop in its core is able to handle a vast amount of data in its native file system and supports the MapReduce paradigm.

A Hadoop job consists of two types of tasks: *map* and *reduce* tasks. In a typical execution, a certain number of map tasks will be executing in parallel reading data from an input file and generating data for the reduce tasks. The reduce tasks can only start when all map tasks have finished.

A Hadoop cluster consists of many compute nodes where map and reduce tasks execute. Each compute node has a number of slots for the execution of map and reduce tasks. Each compute node has a *TaskTracker* daemon that monitors the execution of the tasks running on that node, allocates tasks to slots, and communicates with the *JobTracker* daemon (one per Hadoop

cluster). The JobTracker daemon is responsible for scheduling the map and reduce tasks on different compute nodes where TaskTrackers are running. TaskTrackers are ultimately responsible for starting and managing the individual tasks.

There have been some prior work on the performance of jobs in a MapReduce environment. Verma et al. [2] build a job profile that summarizes critical performance characteristics of the map, shuffle, and reduce phases of a Hadoop job. They then design a MapReduce performance model for a job with the known profile and estimate the completion time. Contention based on multiple jobs or varying number of tasks on a machine is not taken into account. Ganapathi et al. [11] use statistical calculations to predict the completion time of Hive generated Hadoop jobs. In [12] they use statistical techniques to predict query performance in parallel databases. Yang et al. [17] propose a statistical analysis approach to identify the relationships among workload characteristics. They apply cluster analysis to 45 different metrics, which derive relationships between workload characteristics and corresponding performance under different Hadoop configurations. However, they do not consider the delays due to contention caused by multiple running jobs. Vianna et al. [16] model the performance of a different implementation of MapReduce called Hadoop Online Prototype, which allows data to be pipelined between tasks and between jobs. Their work does indeed utilize approximate Mean Value Analysis (AMVA) methods. They use simulation rather than experimentation to validate their model.

The completion time of a job in the Hadoop environment is the time needed for all its tasks to complete. The complexity of estimating a job completion time arises because various jobs of different length can share a Hadoop cluster. In addition, jobs can be either CPU, I/O or memory bound (or indeed a mixture of those). The motivation for our research is that once the job completion times can be predicted for a certain workload and concurrency within a MapReduce cluster, that information can then be used for a myriad of activities such as efficient scheduling of the jobs to maintain a service level agreement, capacity planning, and energy preservation. In this paper, we present a methodology and a queuing network based analytic model for assessing the completion time of a MapReduce job consisting of map tasks only. Experiments have been carried out to validate the model in both a single node as well as a 2-node Hadoop environment.

The rest of the paper is organized as follows. Section 2 presents a short description of Hadoop's MapReduce framework. Section 3 describes the chosen workload and the execution environment for the experiments we conducted. The next section presents our analytic model for the execution time of the map phase of a Hadoop job. Section 5 presents the results of the experiments and of the validation of the model against experiments. Finally, section 6 presents some concluding remarks as well as plans for future work.

2 Hadoop Distributed File System and MapReduce

Hadoop [8] is an open source implementation based on Google's MapReduce and is a commonly used commodity-server based cluster for processing large amounts of data in parallel. Fig. 1 shows a typical Hadoop cluster with four compute nodes and one head node. The head (also called Master) node components (NameNode and Job Tracker daemons) are responsible for managing the data and for scheduling jobs executed by the processing nodes. Each compute node hosts one DataNode that manages the data stored on the machine. Additionally, each compute node hosts one TaskTracker daemon, which is responsible for managing the startup and shutdown of the tasks (Map and Reduce) that are scheduled by the JobTracker.

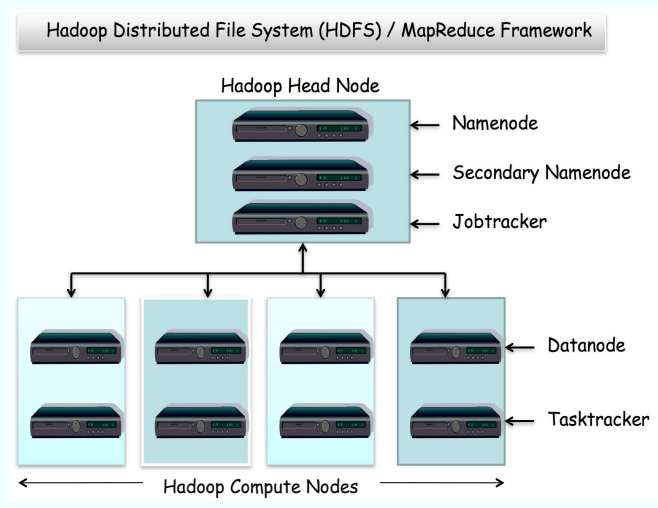

Figure 1: Hadoop Distributed File System.

Figure 2 shows a situation with a JobTracker and three task queues. The tasks of each job are either map or reduce tasks. The TaskTrackers shown in the figure manage task slots that are used to run map or reduce tasks as needed.

The Hadoop Distributed File System (HDFS) is an open source implementation of Google's File System. HDFS consists of many DataNode daemons that are responsible for storing file system data in a replicated fashion and run as distributed components on a cluster of servers (see Fig. 1). In a production or integration environment, the hundreds or thousands of processing nodes can help an application scale out linearly and handle petabytes of data. HDFS is designed for use by MapReduce jobs that read input in large chunks, process it, and write potentially large chunks of output. HDFS does not handle random access particularly well. For reliability, file data is simply mirrored to multi-

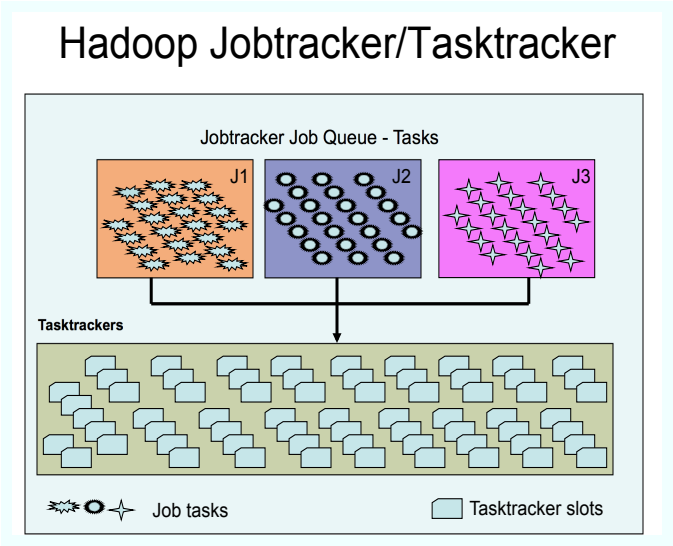

Figure 2: Hadoop JobTracker and TaskTrackers.

ple storage nodes. This is referred to as replication in the Hadoop community. As long as at least one replica of a data chunk is available, the consumer of that data will not notice storage server failures. HDFS services are provided by two processes: a) NameNode handles management of the file system meta-data, and provides management and control services. b) DataNode provides block storage and retrieval services. There is one NameNode process in an HDFS file system and multiple DataNode processes within the cluster, with typically one DataNode process per storage node in a cluster.

2.1 Hadoop Cluster

Each job in a MapReduce environment consists of map tasks, which output a key and a value, followed by reduce tasks, whose methods are guaranteed to receive all the values for a particular key together. These MapReduce based algorithms have been proved to solve a variety of problems including sales data analysis, log, customer behavior analysis, graph processing algorithms and web click stream analytics.

Hadoop can be installed on one machine in a pseudo distributed mode whereby all the Hadoop daemons mentioned previously (i.e., NameNode, SecondaryNameNode, JobTracker, one copy of TaskTracker and one copy of DataNode) run on a single machine. This is very useful for the development of MapReduce programs. Even though in this configuration, it is not possible to work with large volumes of input data (hundreds of GB or more), it is perfectly adequate for functionality testing of MapReduce applications.

Hadoop is built primarily in Java. Each Hadoop daemon is an individual Java process on the machine where it runs. A phenomenal number of knobs are available from the standpoint of OS, Java, and Hadoop itself to fine-tune a Hadoop cluster with hundreds or even thousands of machines. It is not uncommon to hear engineers doing Linux kernel level patches to make swapping work better with Hadoop workloads or to be able to run Hadoop workloads on GPUs. However, there are many cases of Hadoop clusters consisting of 15-50 machines, working out of the box just fine and able to keep up with running a variety of MapReduce jobs, processing massive amount of data on commodity servers.

2.2 MapReduce Examples

Hadoop distribution comes with a set of typical MapReduce jobs such as WordCount, calculating the value of Π, and sorting data in a file. These examples show how the map and reduce phases can be used in a parallel fashion on a multitude of nodes to solve a computational problem. We explain now how we can read a file and count how often words occur and outputs the result in a text file. This example is known as the WordCount program previously mentioned.

In the WordCount program, each mapper task takes a line from the input file, breaks it into words, and emits a key/value pair consisting of the word and a value of 1 (see Fig. 3). Each reducer sums the counts for each word and emits a single key/value with the word and the sum. Below is a small example of a typical map and reduce task of the WordCount program in a Java-type pseudocode.

```
WordCount Map Method

IntWritable one = new IntWritable(1);
Text word = new Text();
public void map (Object key, Text value,
Context context) throws .... {
 StringTokenizer itr =
  new StringTokenizer (value.toString());
 while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
 }
}
```

When we give an input file containing many lines to this program, the input file is split into many parts, based on the Job configuration, and each split of the file is handled by one map task. Each line of each split is an input to one method call to the Map method shown above. The Map method tokenizes this input line (identified by the second parameter, `value`, in the map method) and then for each token (i.e., each word in the input line), it emits the token and the number 1 (identified by the IntWritable variable called `one`).

After all the map tasks finish emitting these key/value pairs, the reduce methods come into the picture. The listing below shows part of the reduce method for the WordCount program. It receives each unique emitted

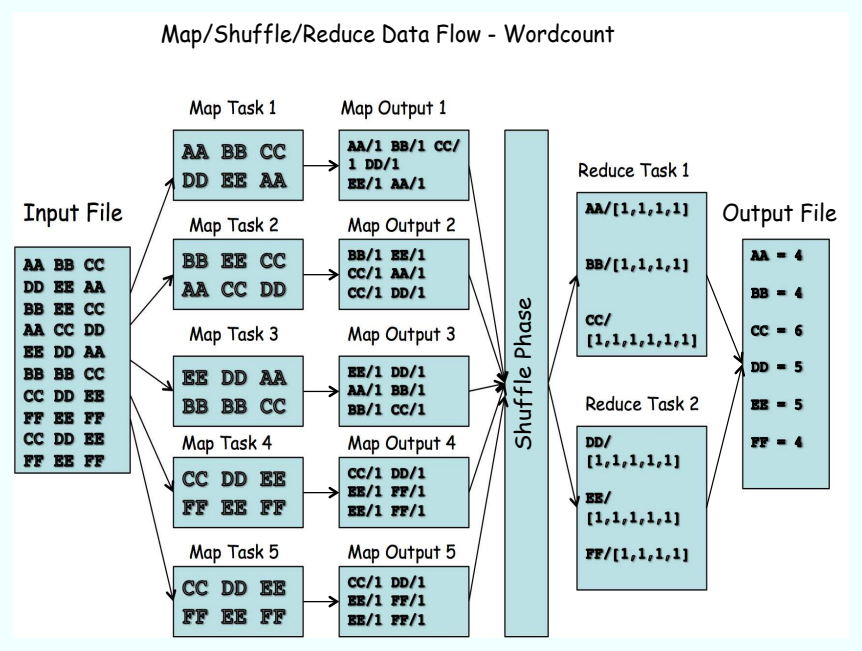

Figure 3: Example of WordCount Execution in a MapReduce Environment.

word only once along with a collection of values that maps outputted for that word. So for example, if all the mappers emitted the word “Pilot” 5 times, the reduce method will be called once with the key (first parameter) of “Pilot” and a collection of IntWritables (second parameter) with 5 elements each with a value of 1. Reduce method then simply iterates over the collection and figures that the word “Pilot” appeared 5 times in the input document.

```
WordCount Reduce Method

IntWritable result = new IntWritable();
public void reduce(Text key,
  Iterable<IntWritable> values,
  Context context) throws ..... {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
```

To run the example WordCount MapReduce job, we have to execute

bin/Hadoop jar Hadoop-*-examples.jar wordcount [in-dir] [out-dir]

We now discuss another example of a slightly uncommon MapReduce job. We call it Indexer MapReduce. Over the last several years along with BigData and Hadoop, we have also been working with NoSQL databases. Google’s BigTable [5] is the first example of a widely talked about NoSQL database. The phrase has been coined by the industry with the explicit intention of separating this type of databases from the SQL/RDBMS databases prevalent all over. HBase [9], Cassandra [3], Accumulo [1] are examples of NoSQL databases and they all provide a structured view of data somewhat close to a TABLE in a Relational DBMS. All three above mentioned NoSQL databases use the Hadoop Distributed File System (HDFS) as its underlying storage mechanism. Accumulo, in particular, is a BigTable [5] inspired distributed key/value storage developed by the National Security Agency (NSA). Consider an example of a simple Accumulo table called UserZipCode (see Table 1). Assume that the UserZipCode table contains UserName as the row key along with the ZipCode for the user in the Column Family (CF) / Column Qualifier (CQ) portion of the row. It is not important to understand what exactly CF and CQ are for this discussion, but if we inserted some rows and scanned the table, the data will show up as follows.

Table 1: Rows for Accumulo Table - UserZipCode

Row Key	Col. Family	Col. Qualifier	Value
JimS	ZIP	20149	XXXX
JohnH	ZIP	22102	XXXX
JohnJ	ZIP	22149	XXXX
LarryM	ZIP	20146	XXXX

A couple of important things of note are that the key comprises of Row Key, Column Family and Column Qualifier and while scanning the table, the keys

are always sorted. So JimS appers before JohnH who appears before JohnJ and so on. The keys are made of many other parts (e.g., timestamp) but they are not shown and are not relevant to our discussion. Now, if a user wants to see all the ZipCodes for people whose names start with a J, we could simply set a scanner on this table with the start range of "J" (inclusive) and end range of "K" (exclusive) and scan the table. That will allow Accumulo to very efficiently fetch all the users whose names start with J. However, if the intention was to get all the users whose zip code started with the number 201, then it would mean that we will have to scan the whole table, since unlike RDBMS, we do not get automatic index creation with the NoSQL tables. And scanning the whole table is not a palatable idea since we could potentially have many millions of rows in the table and a full table scan will be time prohibitive.

We can however create an index with the help of a MapReduce job for the second access pattern. The MapReduce job will read in this original table and its map tasks will emit the zip code and the user name as the key/value pair. The reduce will get all the users associated with one zip code and then it is a simple matter of outputting that into a new table—we will call it ZipCodeUser table. The keys of the new table will look as shown in Table 2. Now, for the second access requirement, it is an easy matter to set the start range on the scanner to "201" (inclusive) and the end range to "202" (exclusive) and Accumulo will be able to fetch all the users whose zip codes start with 201 very efficiently. The MapReduce job creating and maintaining the index table will have to be running continuously thus keeping the main table and its index in sync as far as possible. NoSQL databases operate with the understanding of "Eventual Consistency" and thus a window of time when the original data and the index are not in sync is not considered a big deal.

Table 2: Rows for Accumulo Index Table - ZipCodeUser

Row Key	Col. Family	Col. Qualifier	Value
20146	USER	LarryM	XXXX
20149	USER	JimS	XXXX
22102	USER	JohnH	XXXX
22149	USER	JohnJ	XXXX

3 Workload and Execution Environment

As discussed in the previous section, various different kinds of MapReduce jobs are executed in an enterprise and the execution environment varies too between clusters with only a few nodes to hundreds of nodes.

3.1 Execution Environment

We used two different Hadoop configurations for our experiments. The first is a single-node configuration with an i5 quad-core CPU machine with 6 GB of memory. We ran all the Hadoop daemons (NameNode, SecondaryNameNode, JobTracker, DataNode, TaskTracker) and all the tasks on this machine. The daemons communicate amongst themselves to schedule jobs through the JobTracker and TaskTracker. In the 2-node configuration, we ran the single daemon components (NameNode, SecondaryNameNode and JobTracker) on a dual-core CPU with 4 GB of memory. The rest of the daemons (DataNode and TaskTracker) ran on the machine with the quad-core CPU with 6 GB of memory. MapReduce job tasks (i.e., maps) ran alongside TaskTrackers. Thus, in the first case, all the tasks ran on one machine alongside all the Hadoop daemons and in the second case the tasks ran on the second machine, which we call the Hadoop compute node. The jobs we used in our experiments do not demand significant network bandwidth. Moreover, the input file and the tasks that read the data are co-located. Therefore, we have not considered network contention and its effect in our model. For a full fledged job, running on several compute nodes where the shuffle phase (the intermediate phase between map and reduce) could potentially transfer large amounts of data, we will need to pay attention to the contention on the cluster brought by network traffic.

We briefly describe the steps for installing Hadoop on a 2-node cluster with Cloudera's [4] Hadoop distribution running Ubuntu Linux version 11.10. After receiving the tarball from the download website, we simply extracted it to a directory. We used the same directory on both machines to keep matters simple. We called our first machine 'Master' and the second machine 'Worker'. Hadoop has 5 important configuration files among others. They all reside in the `conf` directory of the standard distribution. The file named 'masters' defines on which machines Hadoop will start SecondaryNameNode in our multi-node cluster. In our case, this is just the Master machine. The primary NameNode and the JobTracker will also be running on this machine. The second file, 'slaves', lists the hosts, one per line, where the Hadoop slave daemons (DataNodes and TaskTrackers) will be run. We wanted only the Worker box to act as Hadoop slave because we wanted to collect performance and timing data on only one processing node. Next we consider the 3 XML configuration files - core-site.xml, hdfs-site.xml and mapred-site.xml. The first two had standard entries in our experiment, but mapred-site.xml had two entries of particular interest. The first is a property called "mapred.tasktracker.map.tasks.maximum". This determines the number of concurrent map tasks that the JobTracker will schedule on this particular machine. We varied this value between our experimental runs to see the effect of the number of slots on the

completion time of our job. The second parameter of interest is "mapred.child.java.opts". We set this value to 400 MB to give each of our map tasks a maximum of 400 MB of Java heap space. Since our goal for the experiment was to model the CPU and I/O usage, our job did not have a high memory demand and thus we were able to run a high number of map tasks without having to use swap space on the Worker machine.

We start all the Hadoop daemons from the Master box with one command: `bin/start-all.sh`. This invokes several scripts starting NameNode, SecondaryNameNode and JobTracker on the Master machine. The scripts also SSH into the Worker machine (in clusters with hundreds of machines it does SSH to all the Slave nodes), and starts the DataNode and TaskTracker daemons. Hadoop provides a convenient way to monitor the entire cluster with web pages so that one can monitor the health of the Hadoop file system and also the progress of the MapReduce jobs that are started on the system. Once we were sure that the environment was fully up, we copied our input file with the following command.

bin/Hadoop -fs copyFromLocal /local/dir/inputFile /apps/common/in1

This has the effect of copying the POSIX file inputFile into the distributed file systems of Hadoop. In our case, with one DataNode, our 5-MB input file has a replication factor of 1 and is copied only to the Worker node. However, for a very large file, on a cluster with hundreds of DataNodes (Worker machines), the file will be split into many pieces and stored on many boxes along with their replicated copies. Once we copied the file, we were ready to run our job with the following command.

bin/Hadoop jar Hadoop*example*.jar CpuDIskLoad

Our goal was to measure the CPU and Disk utilitization of the Worker machine for a varying number of map slots. We describe the workload and the interplay with the number of slots in more detail in the next subsection. All measurements were performed on the Worker machine with the help of Unix `iostat` command. Data was collected for both CPU and Disk at an interval of 5 sec with the command `iostat -x -t 5`. We wrote several small utility programs to calculate the average utilization from the `iostat` output file. Also, we analyzed the Hadoop MapReduce userlogs to determine the completion time for each task.

3.2 Workload Characteristics

Our goal for this paper was to create a model for only the map tasks of a MapReduce job. A standard MapReduce job consists additionally of the shuffle and reduce phases. But we intend to consider those phases and the contention due to those activities in a future paper.

We first show below the relevant parts of our Job configuration code. MapperClass property of the Job object is set to TokenizeMapper class and it is shown in the next listing. Number of reduce tasks is set to 0 (setNumReduceTasks(0)) so that we deal with only the map phase. The input file is a Hadoop DFS file and it is stored and read from /apps/common/in1. The last line of the listing shows that we set the setMaxInputSplitSize anywhere between 100K and 600K. Since our input file size is 5MB, the varying input split size will determine the number of map tasks created to complete this job. As an example, when the split size is 200 K, we get 26 map tasks, which bring the total data handled to around 5 MB (the input file size).

```
Job Configuration Method

Job job = new Job(conf, "word count");
job.setJarByClass(LoadCreator.class);
job.setMapperClass(TokenizerMapper.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setNumReduceTasks(0);
FileInputFormat.addInputPath(job,
new Path("/apps/common/in1"));
FileInputFormat.setMaxInputSplitSize(job,
200000); // 100K to 600K
```

Next is the listing of the map method of our MapReduce job. With every invocation of the map method, it gets a line from the input file. We do not do any processing with that input though. Our goal is to create CPU and Disk Load. We do that by calling the generateSentence() method, which in this case creates and returns a Java String (Text object is just a Hadoop String wrapper for our purposes) with 3,000 words separated with spaces. Once we get the two Text objects `keyWords` and `valueWords`, we then use the context object to write this key/value pair out in a file which produces load on the Disk. The CPU load is also generated as a result of work done by the code plus due to the act of TaskTracker starting up and shutting down the task JVMs and the communication between and by various Hadoop daemons.

Once we had the code below, we went and modified Hadoop's example driver to add this class as a new test class and then we were able to build Hadoop example jar and execute with the standard `bin/Hadoop jar` command.

```
public void map(Object key, Text value,
Context context) throws ... {
  StringTokenizer itr =
   new StringTokenizer(value.toString());
  Text keyWords=generateSentence(3000);
  Text valueWords=generateSentence(3000);
```

```
  context.write(keyWords, valueWords);
}

private Text generateSentence(int
noWords) {
  StringBuffer sentence =
new StringBuffer();
  String space = " ";
  for (int i = 0; i < noWords; ++i) {
    sentence.append("arbitraryword");
    sentence.append(space);
  }
  return new Text(sentence.toString());
}
```

4 Analytic Model

We now describe the analytic model we built to predict a MapReduce job's map phase completion time. Let M be the number of map tasks of a job and let n be the total number of map slots in the worker nodes. Let c be the number of compute nodes. We assume in the following discussion that the n slots are equally distributed among the c compute nodes and that $n = k \times c$, i.e., each compute node has k slots.

The first n maps are executed in parallel using k slots in each of the c compute nodes. Then, the next n maps are executed in that fashion until the remaining n' map tasks are executed. The number of times that n tasks are executed in parallel is $\lfloor M/n \rfloor$ and $n' = \mathrm{mod}(M, n) = M - \lfloor M/n \rfloor \times n$. For example, consider that a Hadoop job has 54 map tasks and that there are 8 slots in 4 compute nodes (each node has 2 slots). Then, there are 6 waves of 8 parallel map tasks followed by a wave of 6 parallel tasks. Thus, in this example, $M = 54, n = 8, c = 4, k = 2, \lfloor M/n \rfloor = 6$ and $n' = \mathrm{mod}(54, 8) = 6$.

It turns out that all map tasks that execute in a compute node compete for the computational resources (e.g., CPU, disks, network bandwidth, etc.) of that node. Therefore, the execution times of map tasks is elongated by the contention (i.e., queueing for resources) generated by the concurrent execution of tasks at a compute node. We address this issue by using Mean Value Analysis (MVA) to solve the closed Queuing Network (QN) model that captures all computational resources of a node and its workload of tasks [13]. Let $T_{\mathrm{MVA}}(s)$ be the execution time of tasks at a compute node when there are s tasks executing at the same time. This time is obtained by solving the QN that corresponds to a computational node. We are assuming that a compute node executes tasks of the same job and therefore they are similar. To extend this to a more general situation we need to use well-known multiclass closed QNs [13].

We can now compute the total execution time, $T(M, n, c)$, of the map phase of a Hadoop job as

$$T(M, n, c) = T_{\mathrm{MVA}}(n/c) \times \lfloor M/n \rfloor + T_{\mathrm{MVA}}(mod\,(M, n)/c) \quad (1)$$

The first term in equation (1) accounts for the first waves in which n tasks are executed in parallel. The second term accounts for the time needed to execute the balance of the tasks.

We observed in the experiments that the number of concurrent map tasks in execution is slightly lower than the number of slots in a compute node. This is due to the fact that it takes some time for the TaskTracker to detect the completion of a task, notify the JobTracker of this fact through a heartbeat message, obtain the response from the JobTracker with a request to start the following task, and lastly spawn a Java VM for a map task. We than observed that there is a linear relationship of the form

$$\mathrm{MapTaskConcurrency} = \gamma \times \mathrm{NumSlots} \quad (2)$$

as exemplified in Fig. 4.

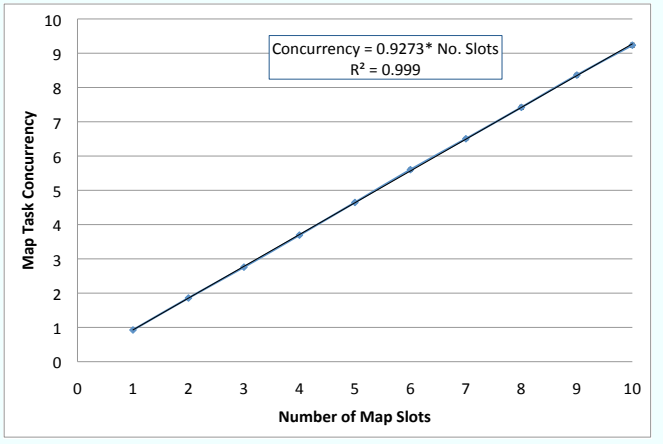

Figure 4: Concurrency of Map Tasks vs. No. Slots for 54 Map Tasks.

The slope γ in equation 2 is then used to adjust the value of the execution time $T(M, n, c)$. The adjusted execution time then becomes $T(M, n, c)/\gamma$.

If contention at compute nodes is not considered, the job execution time, T_{NoCont} would be

$$T_{\mathrm{NoCont}}(M, n) = T_m \times \lfloor M/n \rfloor + T_m \times \epsilon(mod\,(M, n)) \quad (3)$$

where T_m is the execution time of a map task under no contention and $\epsilon(x)$ returns 1 if $x > 0$ and 0 if $x = 0$.

5 Experimentation Results and Model Validation

We first provide some results obtained from the experimental setup described in previous sections. Figure 5 shows the variation of the execution of our Hadoop job with 54 map tasks as a function of the number of slots. The figure also shows the 95% confidence intervals for the experimental measurements. As it can be seen, the execution time initially decreases as more slots are available, which increases the parallelism. However, after a certain number of slots (4 slots in the figure), the

execution time starts to increase because the effect of the increase in contention at the compute node exceeds the benefits of additional parallelism.

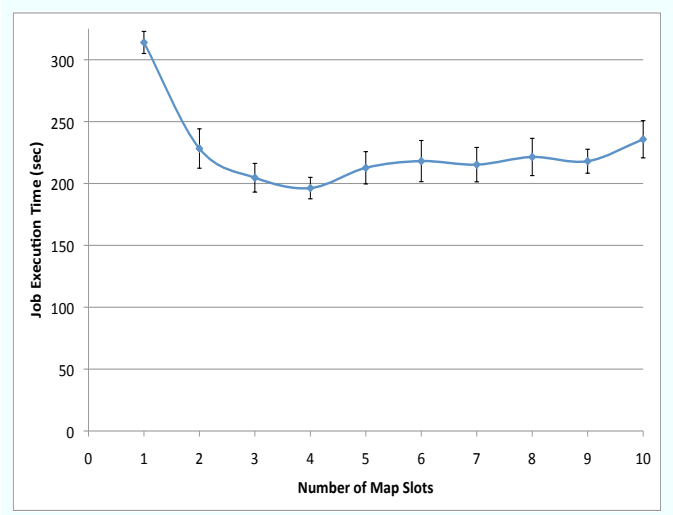

Figure 5: Job Execution Time (in sec) vs. No. Slots for 54 Map Tasks.

Figure 6 shows the variation of the job execution time as a function of the number of map slots for three different values of the number of map tasks: 11, 28, and 54. For each value of the number of map tasks we observe a similar behavior as in Fig. 5; the effect is more pronounced for 54 map tasks though. We also observe that there is not much difference between 11 and 28 map tasks but the execution time for 54 map tasks is significantly higher than for 11 and 28 map tasks. Note that in the 54-map case, the TaskTracker has to create and stop a much larger number of JVMs, which increases the completion time.

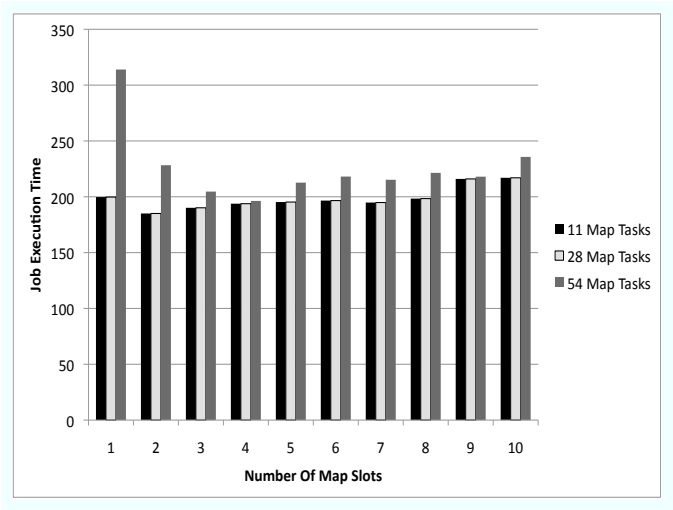

Figure 6: Job Execution Time (in sec) vs. No. Slots for 54, 28, and 11 Map Tasks.

In order to validate the model, we compared the values obtained from experimentation with the analytic model values. First, we had to obtain the service demands to be used as parameters for the closed QN model. As indicated below, the compute node uses a quad-core CPU. The machine also had a single disk. Thus, to obtain the service demand values, we ran experiments with a single slot configuration (to eliminate contention) and computed the service demands for the map tasks using the Service Demand Law ($D_i = U_i/X_0$; i.e., the service demand at resource i is equal to its utilization divided by the system throughput) [13].

Since our experiments were all conducted on multicore machines, we use an approximation proposed by Seidmann et al. [15] to model multiprocessors. The basic idea is that a single queue with m servers and a service demand value of D at each server can be replaced with two queues in series. The first is a single-server load independent queue with service demand equal to D/m, i.e., with a server that works m times faster than any of the original servers. The second queue in the series is a delay server, where no queuing takes place. The service demand for the delay server is equal to $D(m-1)/m$ [14]. Experimentation has shown that the error due to this approximation is between 5% and 11% depending on whether the load is light or heavy.

Figure 7 shows a comparison between the job execution time obtained from experiments, from our predictive model, and from a model where no contention at the compute nodes is considered. As it can be seen, the predictive model tracks reasonably well the measurements while a basic model that does not consider contention (i.e.., T_{NoCont}) and only considers gains due to increased parallelism and does not capture the negative effect of contention.

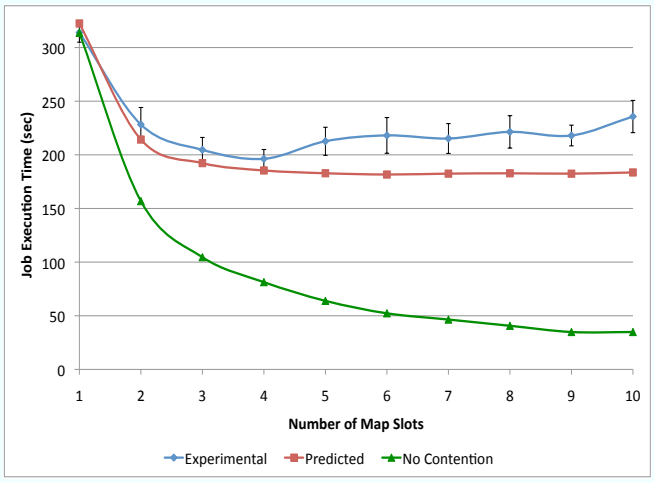

Figure 7: Job Execution Time (in sec) vs. No. Slots for 54 Tasks: Experimental, Predicted using MVA , and No Contention.

6 Concluding Remarks

In this paper we have mainly concentrated on the Map phase of a MapReduce job and provided a model for predicting the completion time of the map tasks based on their service demands on CPU and disk using Mean Value Analysis (MVA). We intend to enhance this work to include the shuffle and reduce phase tasks for a job and also consider other resource types e.g. memory and network. We have performed the experiments on a single node and a 2-node Hadoop environments. We intend to validate the model for a substantially larger number of nodes. In this paper we have considered only a single class of job. It will be also instructional to see how our analytical model behaves with multiple classes of jobs.

References

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

[2] Abhishek Verma et al., *ARIA: Automatic Resource Inference and Allocation for MapReduce Environments*, HPLAB, 2010.

[3] Cassandra, http://cassandra.apache.org/

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

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

[6] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, Proc. Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, Dec. 68, Usenix Association, 2004.

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

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

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

[10] 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.

[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] 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.

[14] D.A. Menascé and V.A.F. Almeida, *Capacity Planning for Web Services: metrics, models, and methods*, Prentice Hall, 2001, ISBN 0-13-065903-7.

[15] A. Seidmann et al., *Computerized Closed Queueing Models of Flexible Manufacturing Systems*, Large Scale Syst. J.,North Holland, vol12, 1987.

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

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