MapReduce Scheduler Using Classifiers for Heterogeneous Workloads

Visalakshi P† and Karthik TU††

†Assistant Professor, Department of Computer Science, PSG College of Technology, Coimbatore, India
†† Student, ME - CSE, PSG College of Technology, Coimbatore, India

Summary
Hadoop is a large-scale distributed processing infrastructure, designed to efficiently distribute large amounts of work across a set of machines. Scheduling of jobs or work is very important in order to achieve efficiency. The proposed work incorporates the design of a new scheduler that will not overload any TaskTracker at any time and thus preventing the unnecessary re-launch of tasks. The scheduler also classifies the job into CPU bound and IO bound. So, a balance is maintained between either of them.

Key words:
Hadoop, MapReduce, Scheduler, Heterogeneous

1. Introduction

MapReduce[5] is a framework, a pattern, and a programming paradigm that allows us to carry out computations over several terabytes of data in a matter of seconds. When it comes to massive-scale architecture and a huge amount of data, with built-in fault tolerance, there's nothing better than this. But when we come to define MapReduce programming, it is just combination of two functions-a map function, and a reduce function. This shows not just the amount of simplicity exposed by the framework in the terms of the efforts of the programmer, but also the sheer power and flexibility of the code that runs under the hood.

MapReduce is a good fit for problems that can easily be divided into a number of smaller pieces, which can thus be solved independently. The data is ideally (but not necessarily) in the form of lists, or just a huge chunk of raw information waiting to be processed-be it log files, geospatial data, genetic data to be used in biochemistry, or web pages to be indexed in search engines. The use of MapReduce is on the rise in Web analytics, data mining, and various other housekeeping functions in combination with other forms of databases. It is also used in complex fields ranging from graphics processing in Nvidia's GPUs, to animation and machine learning algorithm.

MapReduce borrows heavily from the languages of the functional programming model, like Lisp, etc., which are focused on processing lists. Although MapReduce programming gives programmers with no experience in distributed systems an easy interface, the programmer does have to keep in mind the bandwidth considerations in a cluster, and the amount of data that is being passed around. Carefully implemented MapReduce algorithms can go a long way in improving the performance of a particular cluster. Also, all the computations performed in a MapReduce operation are batch processes, as opposed to SQL, which has an interactive query like interface. While solving a problem using MapReduce, it is obvious that the problem has to be divided into two functions, i.e., map and reduce:

The map function inputs a series of data streams and processes all the values that follow in a sequence. It takes the initial set of key-value pairs, and in turn, produces an intermediate pair to be passed on to the reducer. The reduce function typically combines all the elements of processed data generated by the mappers. Its job is mainly to take a set of intermediate key-value pairs and output a key-value pair that is basically an aggregate of all the values received by it from the mapper. Combiner functions are sometimes used to combine data on the mapper node, before it goes to the reducer. Mostly, the code used to apply a combiner and a reducer functions is the same. This allows us to save a lot of data—transfer bandwidth, and can improve efficiency noticeably. But, this doesn't mean that the combiners should be implemented in every case, since if there is not much data to combine, it can take up unnecessary processing power that could be used in a better manner.

Hadoop is the open source implementation of MapReduce. Main areas in research in Hadoop are HDFS (Hadoop Distributed File System) and the scheduler. In this paper, a new scheduler for Hadoop is proposed.

Manuscript received April 5, 2011
Manuscript revised April 20, 2011

2. Background Work

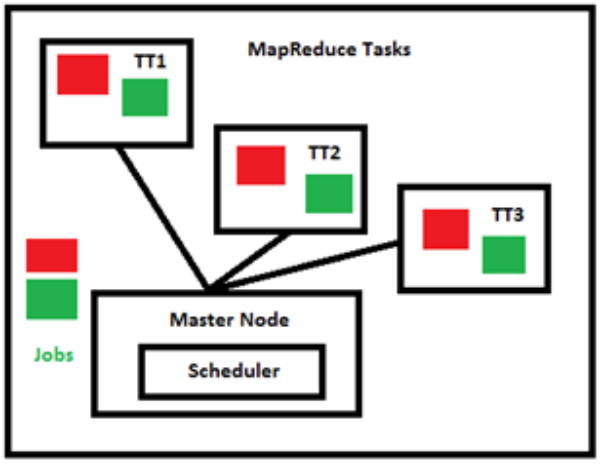

Fig. 1 MapReduce Architecture

Hadoop borrows much of its architecture from the original MapReduce system at Google. Figure 1 depicts the architecture of Hadoop's MapReduce implementation. Although the architecture is centralized, Hadoop is known to scale well from small (single node) to very large (up to 4000 nodes) installations. HDFS (Hadoop Distributed File Systems) deals with storage and is based on the Google File System and MapReduce deals with computation.

Each MapReduce job is subdivided into a number of tasks for better granularity in task assignment. Individual tasks of a job are independent of each other, and are executed in parallel. The number of Map tasks created for a job is usually proportional to size of input. For very large input size (of the order of petabytes), several hundred thousand of tasks could be created.

Scheduling [4] in Hadoop is centralized, and worker initiated. Scheduling decisions are taken by a master node, called the JobTracker, whereas the worker nodes, called TaskTrackers are responsible for task execution. The JobTracker maintains a queue of currently running jobs, states of TaskTrackers in a cluster, and list of tasks allocated to each TaskTracker. Every TaskTracker periodically reports its state to the JobTracker via a heartbeat mechanism. The contents of the heartbeat message are:

(i) Progress report of tasks currently running on sender TaskTracker.
(ii) Lists of completed or failed tasks.
(iii) State of resources – virtual memory, disk space, etc.
(iv) A Boolean flag (acceptNewTasks) indicating whether the sender TaskTracker should be assigned additional tasks. This flag is set if the number of tasks running at the TaskTracker is less than the configured limit.

Task or worker failures are dealt by re-launching tasks. The JobTracker keeps track of the heartbeats received from the workers and uses it in task assignment. If a heartbeat is not received from a TaskTracker for a specified time interval, then that TaskTracker is assumed to be dead. The JobTracker then re-launches all the tasks previously assigned to the dead TaskTracker that could not be completed. The Heartbeat mechanism also provides a communication channel between the JobTracker and a TaskTracker. Any task assignment is sent to the TaskTracker in the response of a heartbeat. The TaskTracker spawns each MapReduce task in a separate process, in order to isolate itself from faults due to user code in the tasks.

When Hadoop started out, it was designed mainly for running large batch jobs such as web indexing and log mining. Users submit jobs to a queue, and the cluster runs them in order. However, as organizations placed more data in their Hadoop clusters and developed more computations they wanted to run, another use case became attractive: sharing a MapReduce cluster between multiple users. The benefits of sharing are tremendous: with all the data in one place, users can run queries that they may never have been able to execute otherwise, and costs go down because system utilization is higher than building a separate Hadoop cluster for each group. However, sharing requires support from the Hadoop job scheduler to provide guaranteed capacity to production jobs and good response time to interactive jobs while allocating resources fairly between users.

Default Scheduler

Default Scheduler or Default Hadoop Scheduler is the Scheduler which is used in default with Hadoop without any extra configuration. This Scheduler schedules jobs in first in first out basis irrespective of job size. The main drawback here is starvation of small jobs in the event of resources being utilized by large jobs.

Fair Scheduler

Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time. When there is a single job running, that job uses the entire cluster. When other jobs are submitted, tasks slots that free up are assigned to the new jobs, so that each job gets roughly the same amount of CPU time. Unlike the default Hadoop scheduler, which forms a queue of jobs, this lets short jobs finish in reasonable time while not starving long jobs. It is also a

reasonable way to share a cluster between a number of users. Finally, fair sharing can also work with job priorities - the priorities are used as weights to determine the fraction of total compute time that each job should get.

The scheduler actually organizes jobs further into "pools", and shares resources fairly between these pools. By default, there is a separate pool for each user, so that each user gets the same share of the cluster no matter how many jobs they submit. However, it is also possible to set a job's pool based on the user's Unix group or any other *jobconf* property, such as the queue name property used by Capacity Scheduler. Within each pool, fair sharing is used to share capacity between the running jobs. Pools can also be given weights to share the cluster non-proportionally in the *config* file.

Capacity Scheduler

Capacity Scheduler is a pluggable Map/Reduce scheduler for Hadoop which provides a way to share large clusters. The scheduling is based on capacity of the resources. In capacity scheduling, queues are guaranteed a fraction of the guaranteed capacity. The free resources allocated to any queue beyond its guaranteed capacity are reclaimed within N minutes of need.

Whenever a TaskTracker is free, the Capacity Scheduler first picks a queue that needs to reclaim any resources the earliest (this is a queue whose resources were temporarily being used by some other queue and now needs access to those resources). If no such queue is found, it then picks a queue which has most free space (whose ratio of # of running slots to guaranteed capacity is the lowest).

Once a queue is selected, the scheduler picks a job in the queue. Jobs are sorted based on their priorities (if the queue supports priorities). Jobs are considered in order, and a job is selected if its user is within the user-quota for the queue, i.e., the user is not already using queue resources above his/her limit. The scheduler also makes sure that there is enough free memory in the TaskTracker to run the job's task, in case the job has special memory requirements.

Data locality and speculative execution are two important features of Hadoop's scheduling. Data locality is about executing tasks as close to their input data as possible. Speculative execution tries to rebalance load on the worker nodes and tries to improve response time by re-launching slow tasks on different TaskTrackers with more resources. The administrator specifies the maximum number of Map and Reduce tasks (*mapred.map.tasks.maximum* and *mapred.reduce.tasks.maximum* in Hadoop's configuration files) that can simultaneously run on a TaskTracker. If the number of tasks currently running on a TaskTracker is less than this limit, and if there is enough disk space available, the TaskTracker can accept new tasks. This limit should be specified before starting a Hadoop cluster. This mechanism makes some assumptions that are objectionable:

(i) In order to correctly set the limit, the administrator has detailed knowledge about the resource usage characteristics of MapReduce applications running on the cluster. Deciding the task limit is even more difficult in cloud computing environments such as the Amazon EC2, where the resources could be virtual.

(ii) All MapReduce applications have similar resource requirements.

(iii) The limit on max number of concurrent tasks correctly describes the capacity of a machine.

.

Clearly, these assumptions do not hold in real world scenarios given the range of applications for which Hadoop is becoming popular. As the above assumptions have been built into Hadoop, all the current schedulers available with Hadoop, the Hadoop default scheduler, FAIR scheduler and the capacity scheduler suffer from this limitation.

3 Proposed System

The task assignment algorithm is explained in this section. The algorithm runs at the JobTracker. Whenever a heartbeat from a TaskTracker is received at the JobTracker, the scheduler chooses a task from the MapReduce job that is expected to provide maximum utility after successful completion of the task.

First a pool of candidate jobs is built. Initially the requirements of the job are not known and they are not requested from the user too. So, one instance of map and one instance of reduce task for a job are obtained and they are scheduled on a worker node. The worker nodes are monitored. Once both the instances of the same job are complete, the requirements for the entire job can be determined. This is based on the fact that the jobs has the same characteristics as their map and reduce tasks.

Now, the job can be classified into IO bound job or CPU bound job. After classification to find the job type, say x and the other is y, the task trackers containing less tasks of the type x than y are selected.

The task trackers are prioritized based on the following factors.

(i) Number of failed tasks on the node.
(ii) Number of tasks from the same job that run earlier on that node
(iii) Resources available for disposal

After prioritizing the TaskTrackers, the best one is selected with the hope that the selected task will run at that node. Now, the selected task is classified into good or bad task with respect to the TaskTracker. If the task is good, then the task is scheduled to run on that node. If not, another TaskTracker is selected and the process is repeated over. This makes sure that any task that may overload a TaskTracker is not scheduled to run at all, which meets the objective of the scheduler.

4 Implementation

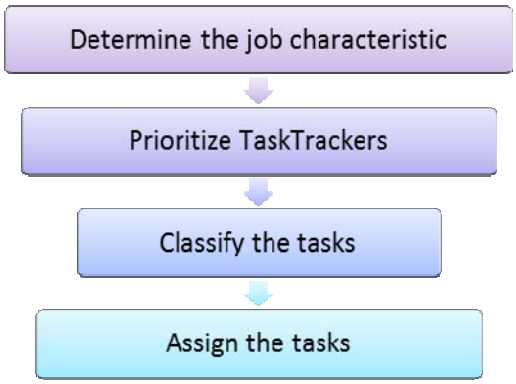

Fig 2 Scheduler Implementation

In figure 2, when the job is submitted for the first time, one instance of map and one instance of reduce task is submitted and the job type is determined. The TaskTrackers are prioritized and the tasks are classified good or bad with respect to the TaskTrackers. If, the task is good, the task is scheduled to that TaskTracker.

4.1 Heterogeneous Workloads

By including the concept of heterogeneous workloads [2] [3] [6], the project tries to improve the hardware utilization rate when different kinds of workloads run in the clusters of MapReduce framework. In practical, different kinds of jobs often simultaneously run in the same cluster. These different jobs make different workloads on the cluster, including the IO bound and CPU bound workloads. But currently, the characters of the jobs are not aware by the Hadoop's scheduler which prefers to simultaneously run map tasks from the same job on the top of the queue. This may reduce the throughput of the whole system which seriously influences the productivity of the data center, because the tasks from the same job always have the same character.

According to the utilization of IO and CPU, the job can be classified as IO bound job or CPU bound job. The ratio of the amount of map input data (MID) and the map output data (MOD) depends on the type of workload. A variable ρ defined in the equation (1)

$$MOD = \rho * MID \tag{1}$$

The quantity q is calculated using equation (2) when a new job is submitted.

$$\frac{(1+\rho)MID}{MTCT} \tag{2}$$

where MTCT is the Map Task Completed Time. If the quantity is less than the Disk IO rate, then the task is CPU bound else IO bound.

4.2 Task Classifier

Tasks of good jobs do not overload resources at the TaskTracker during their execution. Jobs labeled bad are not considered for task assignment. If the classifier labels all the jobs as bad, no task is assigned to the TaskTracker.

If after classification, there are multiple jobs belonging to the good class, then the task of a job is chosen that maximizes the following quantity:

$$E.U.(J) = U(J)P(J = good|F1, F2, ., Fn) \tag{3}$$

where, E.U.(J) is the expected utility, and U(J) is the value of utility function associated with the MapReduce job J. J denotes a task of job J, and P(J = good | F1, F2, . . . , Fn) denotes the probability that the task J is good. The probability is conditional upon the feature variables F1, F2, . . . , Fn. Feature variables are described in more detail later in this section.

The cluster is assumed to be dedicated for MapReduce processing, and that the JobTracker is aware and responsible for every task execution in the cluster. The proposed scheduling algorithm is local as it considers the state of only the concerned TaskTracker while making an assignment decision. The decision does not depend on state of resources of other TaskTrackers.

The assignment decisions are tracked. Once a task is assigned, effect of the task is observed from information contained in subsequent heartbeat from the same TaskTracker. If based on this information, the TaskTracker is overloaded; it is concluded that last task assignment was

incorrect. The pattern classifier is then updated (trained) to avoid such assignments in the future. If however, the TaskTracker is not overloaded, then the task assignment decision is considered to be successful.

Users configure overload rules based on their requirements. For example, if most of the jobs submitted are known to be CPU intensive, then CPU utilization or load average could be used in deciding node overload. For jobs with heavy network activity, network usage can also be included in the overload rule. In a cloud computing environment, only those resources whose usage is billed could be considered in the overload rule. For example, where conserving bandwidth is important, an overload rule could declare a task allocation as incorrect if it results in more network usage than the limit set by the user. The overload rules supervise the classifiers. But, as this process is automated, the learning in our algorithm is automatically supervised. The only requirement for an overload rule is that it can correctly identify given state of a node as being overloaded or under loaded. It is important that the overload rule remains the same during the execution of the system. Also, the rule should be consistent for the classifiers to converge.

4.2.1 Feature Variables

During classification, the pattern classifier takes into account a number of features variables, which might affect the classification decision. The features considered are described below:

Job Features

These features describe the resource usage patterns of a job. These features could be calculated by analyzing past execution traces of the job. It is assumed that there exists a system which can provide this information. In absence of such a system, the users can utilize these features to submit 'hints' about job performance to the classifier. Once enough data about job performance is available, user hints could be mapped to resource usage information. The job features considered are: job mean CPU usage, job mean network usage, mean disk I/O rate, and mean memory usage. The users estimate the usages on the scale of 10. A value of 1 for a resource means minimum usage, whereas 10 correspond to maximum usage. For a given MapReduce job, the resource usage variables of the Map part and the Reduce part are considered different.

Node Features

Node Static Features change very rarely, or remain constant throughout the execution of the system. These include number of processors, processor speed, total physical memory, total swap memory, number of disks, name and version of the Operating System at the TaskTracker, etc. Node Dynamic Features include properties that vary frequently with time. Examples of such properties are CPU load averages, % CPU usage, I/O read/write rate, Network transmit/receive rates, number of processes running at the TaskTracker, amount of free memory, amount of free swap memory, disk space left etc. Processor speed could be a dynamic feature on nodes where CPUs support dynamic frequency and voltage scaling.

4.2.2 Naive Bayes Classifier

A Bayes classifier [1] is a simple probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics) with strong (naive) independence assumptions. A more descriptive term for the underlying probability model would be "independent feature model".

In spite of their naive design and apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many complex real-world situations. In 2004, analysis of the Bayesian classification problem has shown that there are some theoretical reasons for the apparently unreasonable efficiency of naive Bayes classifiers. Still, a comprehensive comparison with other classification methods in 2006 showed that Bayes classification is outperformed by more current approaches, such as boosted trees or random forests.
The Bayes theorem can be applied to the task classification problem using the below formula.

$$U(J) * \frac{P(F1, F2, F3....Fn \mid tj = good)P(tj = good)}{P(F1, F2, F3...Fn)} \quad (4)$$

The denominator in the equation (4) can be treated as a constant as its value is independent of the jobs, and thus its calculation can be skipped during comparison.

Both P (J = good|F1, F2, . . . , Fn) and P(J = bad|F1, F2, . . . , Fn) is calculated. Job is labeled as good or bad depending on which of the two probabilities is higher. Under the assumption of Naive Bayes conditional independence,

$$P(F1, F2, F3....Fn \mid tj = good) = \prod_{i=1}^{n} P(Fi \mid tj = good) \quad (5)$$

Once the effects of task assignments are observed, the probabilities are updated accordingly so that future decisions could benefit from the lessons learnt from the effects of current decisions.

Here, it is assumed that the probabilities of all feature variables are conditionally independent of each other. This may not always be true. However, it is observed that this assumption can yield a much simpler implementation.

4.3 Utility Functions

Utility functions are used for prioritizing jobs and policy enforcement. An important role of the utility functions is to make sure that the scheduler does not always pick up 'easy' tasks. If the utility of all the jobs is same, the scheduler will always pick up tasks that are more likely to be labeled good, which are usually the tasks that demand lesser resources. Thus, by appropriately adjusting job utility it could be made sure that every job gets a chance to be selected. It is possible that a certain job is always classified as bad regardless of the values of feature vectors. This could happen if the resource requirements of the job are exceptionally high. However, this also indicates that the available resources are clearly inadequate to complete such a job without overloading. Utility functions could also be used in enforcing different scheduling policies. Examples of some such policies are given below. One or more utility functions could be combined in order to enforce hybrid scheduling policies.

Map before Reduce

In MapReduce, it is necessary that all Map tasks of a job are finished before Reduce operation begins. This can be implemented by keeping the utility of Reduce tasks zero until a sufficient number of Map tasks have completed.

First Come, First Serve

FCFS policy can be implemented by keeping the utility of the job proportional to the age of the job. Age of a job is zero at submission time.

Budget Constrained

In this policy, tasks of a job are allocated until the user of a job has sufficient balance in his/her account. As soon as the balance reaches zero, the utility of jobs of the said user becomes zero, thus no further tasks of jobs from the said user will be assigned to worker nodes.

Dedicated Capacity

In this policy a job is allowed a guaranteed access to a fraction of the total resources in the cluster.

Revenue oriented utility

In this policy, utility of a job is directly proportional to the amount the job's submitter is willing to pay for successful completion of the job. This makes sure that the algorithm always picks tasks of users who are offering more money for the service.

5 Results and Conclusion

The proposed scheduler for MapReduce will not overload any TaskTracker at point of time. Thus the burden of re-launching the tasks at different TaskTrackers is not necessary. This scheduler classifies the job into IO bound and CPU bound jobs. So, there will be a balance between the number of IO bound tasks and the number of CPU bound tasks running at every TaskTracker. This increases the hardware resource utilization.

References

[1] Jaideep Dhok; Vasudeva Varma (2010), "Using Pattern Classification for Task Assignment in MapReduce".

[2] Matei Zaharia; Andy Konwinski; Anthony D. Joseph; Randy Katz; Ion Stoica (2008), "Improving MapReduce Performance in Heterogeneous Environments." Proceedings of the 8th USENIX conference on Operating systems design and implementation.

[3] Thomas Kunz (1991); "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," IEEE transactions on software engineering, vol. 17, no. 7,.

[4] Saeed Iqbal; Rinku Gupta; Yung chin Fang (2005). "Job Scheduling in HPC clusters" DELL Power Solutions.

[5] "Hadoop Fair Scheduler Design Document" from jira Hadoop documentation.

[6] Chao Tian; Haojie Zhou; Yongqiang He; Li Zha (2009)."A DynamicMapReduce scheduler for Heterogeneous Workloads" Proceedings of the 8th IEEE International Conference on Grid and Cooperative Computing.

Dr. P. Visalakshi is currently working as an Assistant Professor in the Department of Computer Science and Engineering, PSG College of Technology, Coimbatore. She has 14 years of experience in teaching and 4 years of experience in research. She has published 9 International Journal papers and 4 National Journal papers. She has presented 2 paper in International conferences and 7 papers in National conferences. Her area of research interest is the implementation of evolutionary algorithms for solving real time problems.

Karthik T U received the B.Tech. degree from Vellore Institute of Technology Univ., India in 2009. He is currently pursuing M.E degree at PSG College of Technology, Coimbatore. He has published 2 International journal papers. His areas of interests are mobile application development and web mining