**MapReduce: Simplified Data Processing on Large Clusters**

Authored by Jeffrey Dean and Sanjay Ghemawat

Summary

**Introduction**

In 2003, engineers at Google designed a system that solves the issues of parallel computation and data distribution. It hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library and therefore the programmers who do have any knowledge of parallel and distributed systems can utilize it easily.

**Model**

MapReduce is a functional programming model for processing and generating large datasets. It basically consists of two functions: *Map* and *Reduce.* They both are user specified functions. Map function takes an input pair and produces library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Suppose we want to count the number of occurrences of each word in a large collection of documents. The map function outputs each word and its number of occurrences. The reduce function groups all the output for particular word and sums together the count.

**Implementation**

The paper suggests that many implementation of MapReduce interface is possible, from small shared memory machine to large multi-processor or even a large collection of networked machines. The following is the flow of sequence of actions that occurs when a user program calls MapReduce. The MapReduce library in the user program splits the input data into small (16mb or 64mb) M pieces and starts the program on a cluster of machines.

The Master program assigns work to the idle Workers and assign each one a map task or a reduce ask. There are M map tasks and R reduce tasks to be assigned. The Worker reads the content of the input split assigned to him and parses the key/value pairs out of the input data and passes to the user defines Map function.

The map function then produces the intermediate key/value pairs that are buffered in the memory. The buffered data is periodically written to R partitioned disks. The location of these buffered pairs is passed back to the master who forwards these locations to the reduce workers.

Then the reduce worker uses remote procedure calls to read the buffered data from the local disks of the map workers. After reading all the intermediate data it sorts it by the intermediate keys so that all occurrence of the same key is grouped together.

The reduce worker iterates over the intermediated sorted data and passes the key and corresponding value to the user’s Reduce function. The output of the reduce function is written to the final output file for this reduce partition.

After all the map and reduce tasks are completed, the master passes the control to the user program.

The master keeps the information in several data structures. For each Map and Reduce tasks, it stores the state (idle, in-progress, completed) and the identity of the worker machine (for non-idle tasks).

**Fault Tolerance**

MapReduce library handles the failures well as it is designed to help process large amount of data using a lot of machines.

To check for worker failure, the master pings every worker periodically. Upon receiving no response in a certain amount of time, the worker is marked a failed by the master. All the map tasks either completed or in progress on a failed worker are reset to idle and are re-executed. Map tasks needs to be re-executed because their output is stored on local disks of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed on a failure because their output is stored in global file system.

If a Master fails, then the complete MapReduce computation is aborted.

One of the common cause that lengthens the total time taken for a MapReduce is a “straggler” which is basically a machine that takes unusual large amount of time to complete the last map or reduce jobs. Straggler can arise because of bad disk, over assigning of the tasks or low memory, CPU or network bandwidth. To get rid of this problem MapReduce schedules backup executions of the “in-progress” tasks when MapReduce operation is close to completion. The task is marked complete when either the primary or backup task is completed. User has an option to switch on the Backup Task or not.

**Refinements**

In addition to the basic MapReduce, the paper also suggests some extensions that are as follows:

Partitioning function: The users can specify the number of reduce tasks/output (R).

Ordering Guarantees: In a particular partition, the intermediate key/value pairs can be sorted in ascending order.

Combiner Function: User has the option to partially merge the data before sent over the network by executing the map task on each machine that performs the map task.

Input and Output Types: MapReduce provides support for several data types for input and output. MapReduce has the capability of detecting and skipping the bad records that are causing periodic failures.

To tackle the debugging problems in MapReduce, engineers at google developed an alternative implementation for users that executes all the operations and jobs on a local machine.

To predict the completion time or resource allocation, the user can use the functionality of status pages. These status reports show a bunch of useful administrative details.

User can use the counter functionality in MapReduce library to count the number of occurrences of various events. They are useful for sanity checking.

**Performance**

The whole goal of MapReduce is to increase performance of processing large amount of data. The paper explains it using two computation running a large cluster of machines. First computation searches through 1 terabyte of data for a particular pattern, the second computation sorts that 1 terabyte of data.

The first program called the Grep program scans through data and searches for three character words. The conclusion was that the rate was increased as more workers were assigned. The sort program was much larger than the grep program and hence the map tasks spent half of their time doing the same. Around 300 seconds into the computation, some batches finish and shuffling operation is performed for the remaining reduce tasks. Everything is completed about 600 seconds into computation. After 960 seconds, all except five reduce tasks were completed and these stragglers took another 300 seconds to complete.

It is also important to observe that the execution time of sort program is high because of stragglers. It is low if the backup tasks are enabled.

**Conclusion**

The MapReduce programming model has been very successful at Google for many different purposes, like:

* It’s easy to use. Programmers without prior knowledge of parallel distributed system can implement this.
* It can be applied to a lot of types of problems. E.g. web crawling, data mining, machine learning.
* It is highly scalable.
* It optimizes the utilization of network bandwidth.