Paper Critique MapReduce

Kevin Cho

March 2016

I Introduction

MapReduce is a system that was made to process and generate large data sets. It was separated into two functions: map and reduce. The map function would link each single data to a key. Then, it would transfer all these pairs into intermediate files that the reduce task would read and organize. The reduce function just combined all pairs with the same key and gave an organized output value. All these tasks were overseen by a master system that would help the system run correctly.

The MapReduce system, which was described by Jeffrey Dean and Sanjay Ghemawat, provided a strong model for processing large data sets. The system was well designed in order that it be (1) easy to use for people with no little experience, (2) automated in most of its process, and (3) scalable to large clusters of machines. As a result, the system was widely applicable, but due to running it in a secure environment, the authors may have traded-off on security. This paper will discuss how the system is a strong tool, but may have some problems to it.

II Design Goals of MapReduce

Scalability. One the of the most crucial design goals of MapReduce is scalability. Jeffrey Dean and Sanjay Ghemawat created a system that was able to be scaled into clusters of machines that each had over thousands of machines. The way that they achieved this goal was mainly due to the modularity they had in their system. With its name being MapReduce, the system is seperated into a map function and a reduce function. This separation led to an effiecient way of modularity within clusters of machines.

The whole system is divided into (Section 3.1) three main modules: the master, map workers, and reduce workers. The master was the module that overviewed the whole execution of the system. The workers were divided by the task they were to do, whether it was a map task or a reduce task. It is, thus, evident that if the system were to expand to a much larger scale, it would actually be better for the system. There would be an increase in the number of machines that process either map or reduce tasks. Through this simple modularity, Dean and Ghemawat achieved a valuable system goal of scalability.

Fault Tolerance. Another crucial design goal for the MapReduce System is fault tolerance. The system was designed to handle almost all the errors and machine failures automatically. The way this goal was achieved was a very clever way. With the modularity of the master system, MapReduce is able to provide a valuable back-up plan for several failures. There are two major failures that the system should account for: worker failures and master failures (Section 3.3).

For worker failures, the master serves as the organizer. If a worker is not processing the way it is meant to, the master will close the worker and have all the jobs that were completed by that worker be redone by another functioning worker. In the case of master failures, Dean and Ghemawat neglect the automatic response and just have the MapReduce computation be aborted. The reason they did this was that they deemed the single master failing was unlikely

A subgoal of fault tolerance would be a reliable back-up system for slower machines. Although slowness is not really an error, it can be considered similar to a fault. The way MapReduce handles this problem (Section 3.6) of 'stragglers' is to have back-up jobs if one machine is taking particularly long. Thus, as long as the back-up job is finished or the 'straggler' finishes, the job is marked as finished.

Simplicity. The last crucial goal is simplicity. The statement that Dean and Ghemawat says numerously is that (Section 8) programmers without experience with parallel and distributed systems are able to use the system well. The way the system is very simple is that it requires minimum user interaction. Almost all the proccesses are automated, requiring little to no user input. The system (Section 8) hides all details of parallelization, fault-tolerance, locality optimization, and load balancing.

Security. The common design goal of security was not significant within the system. The reason behind this was that the area in which the system was running in is considered to be a very secure environment. Dean and Ghemawat were running MapReduce in Google's enviroment with Google's data. Another consideration to add is that all the MapReduce task does is organize data into a better and more useful view. In this aspect, security would not be an issue.

However, if files were to leak out such as the intermediate files that the Map tasks create, this could cause a huge problem to the system. it would lead to the MapReduce giving wrong answers without users knowing. It could also lead to files leaking out important data to unwanted users.

III Analysis of the System

As the authors of the paper states, the MapReduce system has a whole lot of application in the software world. There are several cases that one can take into consideration, but for this paper, only a few will be discussed.

The first case is a huge cluster of machines working together for a service like Google. If a machine-learning algorithm were to take place for a certain user such that search results would display searches similar to the user's preference, data must be collected from previous searches. In this case, MapReduce would be very effective in this case. With its simplicity, everyone would be able to use the system effectively to collect data by just running the system. Because fault-tolerance is almost always handled automatically, any user at Google would be able to run and maintain the system.

In addition to being able to be used by everyone, the MapReduce can produce a sorted, organized data set for the machine-learning algorithm to be run on. One can see how MapReduce is very valuable in this situation.

The second case would be a local machine that is able runs the MapReduce. If a user was curious about his daily internet habits such as which website he visits the most, MapReduce would be a great candidate to use. Because of its scalability and (Section 4.7) changes, it is able to run on smaller machines. To add on top of this, it provides the user with a simple way of finding what he needs, without the need to debug anything. The reason behind this is the automated fault-tolerance.

However, a problem that may arise is that if the master system were to fail, the system would just abort all tasks and quit the system from running. An average user would have trouble trying to figure out what the problem is and what to fix. In this aspect, a better error handling process would be good for when the master fails.

Another consideration is that if the user's computer contains a bug, it can easily erase the intermediate files that map tasks create in the local memory. This would cause problems to output files without the user knowing what was going on. In terms of security from outside bugs on local machines, MapReduce could run into problems.

IV Conclusion

In the paper, there is validity to the authors' claims to the design goals. The system is scalable, fault-tolerant, has modules, and is simple. These features makes the system easy to change, but there is a small neglection of security. Also, there was less fault-tolerance in the master system to have the system be more simple towards users. However, as a whole, MapReduce is a strong programming model that is applicable and easy to use as the authors say.

Works Cited

Dean, Jeffrey and Ghemawat, Sanjay (Google, Inc.). "MapReduce: Simplified Data Processing on Large Clusters."Operating Systems Design and Implementation (OSDI) '04. Usenix. Published 2004.