6.033 System Critique - MapReduce

Jenny Xue (jennyxue@mit.edu)

March 16, 2018

1 Introduction

MapReduce is a distributed programming model for processing and generating large data sets [1.Abstract]. The system is comprised of two key components: Map and Reduce, as inspired by the map and reduce primitives of functional programming languages. At a high level, MapReduce generates and merges key/value pairs based on a set of inputs and the user's specifications.

The advantages of distributed computing is evidently enticing: efficient use and sharing of resources, enhanced performance, reduced processing time, etc. However, behind the scenes, distributed computing requires extensive specialized knowledge. Prior to MapReduce, engineers at Google must manually handle machine failures, network latency, and race conditions for each of their special-purpose distributed computations [1.1].

The designers of Google's MapReduce focused on creating a simple yet scalable system that could be efficiently used by developers with minimal distributed computing experience. Security is not a relevant major design goal because the specifics of parallelization and fault-tolerance are automatically handled behind the scenes as part of the design.

2 System Design

2.1 Implementation

The implementation of Google's MapReduce relies primarily on modularization to achieve automatic parallelization of large-scale computations [1.1]. The three main modules involved in the system are the user program, master, and workers [1.3.1].

2.1.1 User Program

The user program includes the input data, the partitioning function, and the user-specified map and reduce operations.

2.1.2 Master

When the user program calls the MapReduce operation, the system automatically forks this process into a single master and numerous workers on a cluster of

machines. The master is in charge of assigning and managing task distribution across all workers. When the program is complete, the master wakes up the user program and signals the user.

2.1.3 Workers

Workers can be further broken down into two submodules: map workers and reduce workers.

Map Workers: Map workers parse the key/value pairs from the input data and pass them to the user-defined *Map* function. It outputs intermediate key/value pairs.

Reduce Workers: Reduce workers read and sort all intermediate data and pass the intermediate key/value pairs into the user-defined *Reduce* function. The output is appended to a final output file.

2.2 Simplicity

For the MapReduce designers, the main goal was to build a system that could be used with minimal distributed systems knowledge. This is primarily accomplished through simplicity in the interface.

Much of MapReduce's simplicity stems from the abstraction of automatic parallelization. Fault-tolerance, data distribution, load balancing, and machine failures are all handled behind the scenes, leaving behind merely the user tasks of defining the map and reduce processes, providing input data, and writing the partitioning function.

Furthermore, MapReduce provides numerous methods to simplify the debugging process. Debugging in a distributed environment is complex and difficult, but MapReduce achieves this simplification by allowing for the local and sequential execution of the code. The system also provides controls to the user, such as invoking the program with a special flag, so the computation can be limited to particular map tasks [1.4.7].

2.3 Scalability

MapReduce is a distributed system, which means it is by nature scalable. A typical MapReduce computation processes many terabytes of data on thousands of machines [1.Abstract]. It achieves this by splitting the input files into smaller, 16 to 64 megabyte pieces and starting up copies of the program on a cluster of machines [1.3.1]. This increases the dynamic load balancing and data distribution in the system.

Unfortunately, one disadvantage that comes with this scalability is cost. The large clusters of commodity PCs combined with the commodity networking hardware can prove to be an increasing financial burden as the system grows [1.3].

2.4 Fault Tolerance

Often times, a computer cluster could consist of hundreds or thousands of machines, which means machine failures are common [1.3]. MapReduce was designed to be especially thorough in fault-tolerance, since the whole point of the system is to abstract away the details of distributing computing.

There are three types of failures that MapReduce accounts for.

2.4.1 Worker Failure

The master pings every worker periodically for a response. If no response is received from a worker in a certain threshold of time, the master marks the worker as failed and resets the worker to be idle [1.3.3].

2.4.2 Master Failure

Designers have placed an assumption that given there is only a single master, its failure is unlikely. If the master fails, the implementation aborts the MapReduce computation, but a new copy can be restarted form the last checkpoint. Therefore, it is up to the client to check for this condition and retry the MapReduce operation [1.3.3].

2.4.3 Input Failure

When there is an error in the user code or the input, MapReduce can detect which records are causing the crashes and skips these faulty records [1.4.6].

2.5 Performance

While the main goal of having a distributed computing system such as MapReduce is to increase performance, the shortcomings in the MapReduce system are hard to ignore.

One major detail that the MapReduce designers seem to have overlooked is the number of idle workers at any point in the computation. When the user program process is forked, the master picks idle workers and assigns each a map task *or* a reduce task [1.3.1]. This means that at points of time, there are idle reduce workers are waiting for the map workers to finish and consequentially points of time where map workers are left with no more task to perform. However, one can argue that this performance flaw is a small price to pay when other aspects of distributed computing are taken into account.

3 Conclusion

Despite some of its shortcomings in performance, MapReduce still manages to be an overall simple, scalable, and fault-tolerant system. The map and reduce abstractions is pivotal in allowing developers with minimal distributed computing experience to automatically parallelize data across a large cluster of machines. MapReduce has been shown to be successful at Google: upwards

of one thousand MapReduce jobs are executed on Google's clusters everyday [1.Abstract]

References

[1] Dean, Jeffery and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters* OSDI. 2004. Web.