# Map Reduce

## By Raymond Conn and Max Gale

With the extensive proliferation of the Internet, large companies have experienced the need to process and analyze extensive quantities of data. At the same time, recent advances in distributed computing have allowed massive distributed computing clusters. Using virtual machines, these clusters can appear to be homogenous, even though the virtual machines are being run on heterogeneous hardware. Google, a company with a massive amount of distributed infrastructure, was one of the first to successfully address the need for large data processing in a way that can be widely applied. In 2004, Google published a paper titled “MapReduce: Simplified Data Processing on Large Clusters.”

In Google’s paper, the engineers introduce MapReduce as two things. One component of MapReduce is a programming model, and the other component is an associated implementation of the model. Since the model is generally the same across implementations, the name MapReduce is generally associated with the name of the programming model. When referring to an implementation, the name of the implementation is often used. For instance, The Apache Foundation maintains an implementation named Hadoop. This name used when referring the Apache’s specific implementation of MapReduce. This paper discusses both programming model, and implementation specific areas.

The name MapReduce is derived from the words “map” and “reduce”. These two words are well known by functional programmers. The word map refers to an operation that can be performed on a set. In essence, map can be thought of as a mathematical map, which is also known as a function. When map is applied to a set of values, the set of values serve as input to the map function. The application of this function, in turn, produces an output set of values, which is then returned. An example of this can be seen in Figure 1. As you can see, the map function takes another function as input. Map uses this input function on each of the elements to produce an output set. In the example, the square of each element is calculated. The other part of MapReduce that was borrowed from functional programming is the reduce step. Reduce operates on a set of data, and takes a function as input, which is similar to map. The key difference between map and reduce is, map generally aims to transform a set of data into a new set of data. Reduce, on the other hand, looks to take a set of data and convert it into something smaller. An example of reduce is shown in Figure 2. In this example, the sum of a set is computed.



Figure 1 An example of the map function.



Figure 2 An example of a reduce function.

The motivation behind MapReduce can sometimes be hard to grasp. As programmers have traditionally programmed programs that run on only one machine, it can be hard to understand how and why someone would write code for a distributed system. A good motivational example for MapReduce is the word count problem. Suppose you received a printout of this paper, and you needed to calculate the number of words contained inside. The only caveat is, you cannot use a computer to do so. You may start by counting each word, but you will soon find that the task is too tedious, or takes too long. To speedup the work, you call 10 of your friends and you give each of them a page. The work is done in one tenth of the time, and you now have more free time to count more papers, or any other work you may have to do. MapReduce essentially allows you to call your friends to help you with your work.

Distributing computing is a model in which many computers interact with each other, each performing specific tasks to complete a common goal. Following this model allows for problems that are parallel in nature to be solved much quicker. The increased performance is the result of parallel computing, which is dividing the problem and solving each piece simultaneously. The increase performance comes along with many headaches. A programmer can no longer just write the code that they want executed, they must take care of synchronization and scheduling. All of the added overhead makes a distributed much harder to write and understand. To make distributed programming much easier and reliable the mapReduce framework was created.

MapReduce is a distributed framework that is used to process large amounts of data in a parallel manner. To process the large amounts of data the framework follows a master slave model allowing many individual machines to provide computation power. The framework handles all the complexities that come along with distributed computing and allows for the true power of the distributed model.

MapReduce works by following the master slave model. The master is in control of overseeing the whole process. It does all the scheduling and the handling of fault tolerances, as well as load balancing. It is also in control of assigning the slaves with file locations for input and output of their computations. The slave on the other hand is under complete direction from the master. All the slave does is provide CPU cycles for the computation that the master has supplied to it. The slave is also in constant communication with the master giving status updates as well as results. Aside from the communication with the master, the slave just computes as if it were running a program locally.

For the process to work the framework handles all inputs and outputs as key value pairs. The key value pair is a mapping that associates a certain key with values. An example of key value pair related to the word count example would be a word mapped to the number of times it has appeared so far. The framework operates on key value pairs because it follows the map and reduce functions of functional programming languages.

Since the framework follows the principles of mapping and reducing in functional programming languages, the user is only required to write these two methods. The map function, which is written by the user, takes in a key value pair and produces zero or more new key value pairs after completion of the computation. The reduce function, also written by the user, will take in a key and a list of values associated with the key. This key and list of value are key value pairs that were produced during the map phase of the process. After the reduce phase finishes there will be an output of one or more key value pairs depending on the nature of the problem. The outputted key value pairs from the reduce phase are the results of the computation.

## Overview of the process

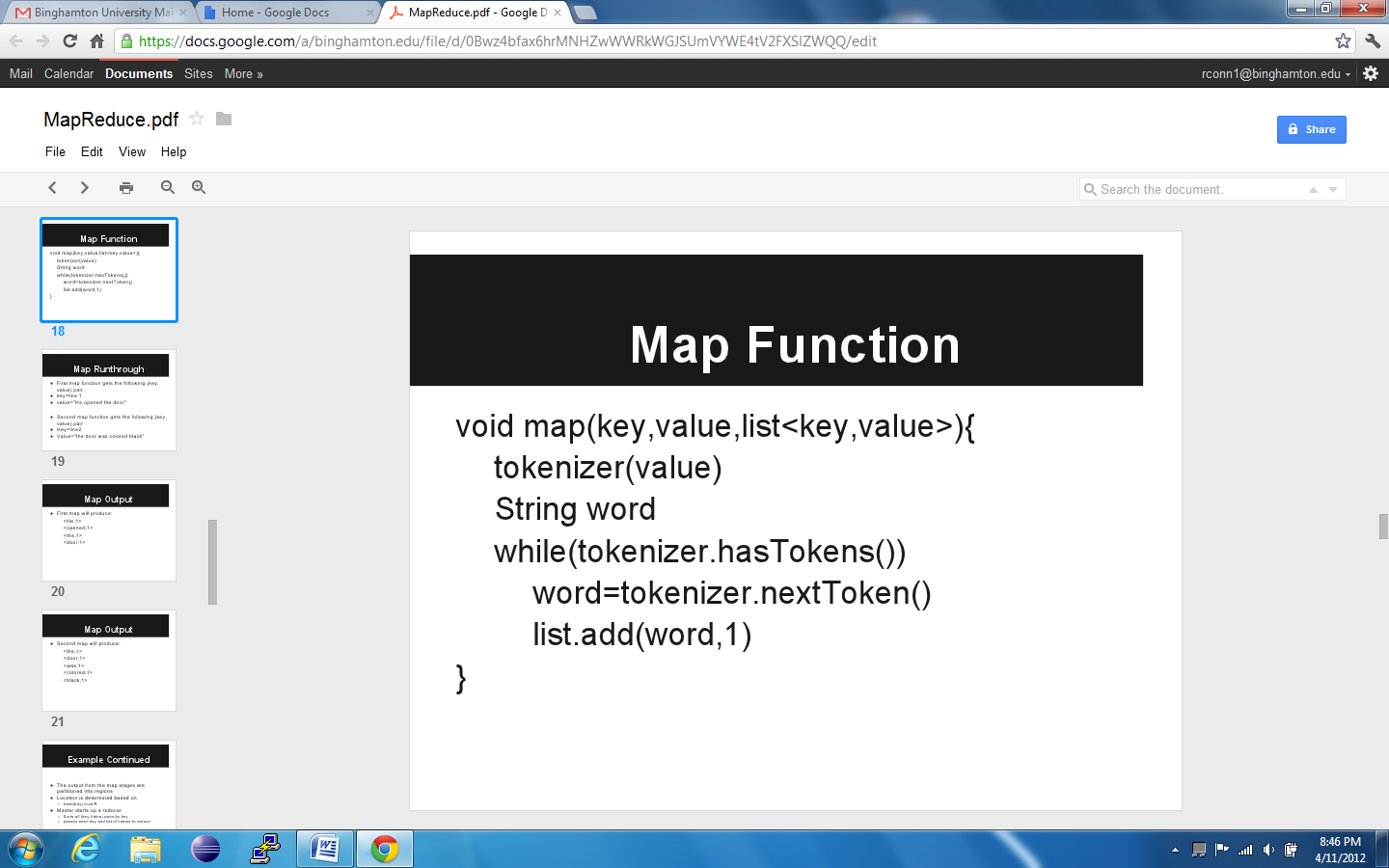


Figure Example of a map function for doing a frequency count of words

The MapReduce process begins with the user writing both the map and reduce functions for the problem they want solved. Figure 1 and Figure 2 are examples of the map reduce functions that perform a frequency count of words. Once the map and reduce are defined the process is ready to begin. At the start of the process, the master will be notified with the file locations of the input data. The master will then split this input data into chunks creating the first set of key value pairs. If the input to the process is text, a typical key would be a line number of the input file and the value is usually the string of text contained on that line. The master will then assign M map tasks and R reduce tasks to idle workers in the network. The numbers M and R are both dependent on the problem being solved as well as the number of slaves available to solve the task. When Google uses the framework to do a word count on 2,000 machines, M is usually chosen to be 200,000 and R is usually 5,000.

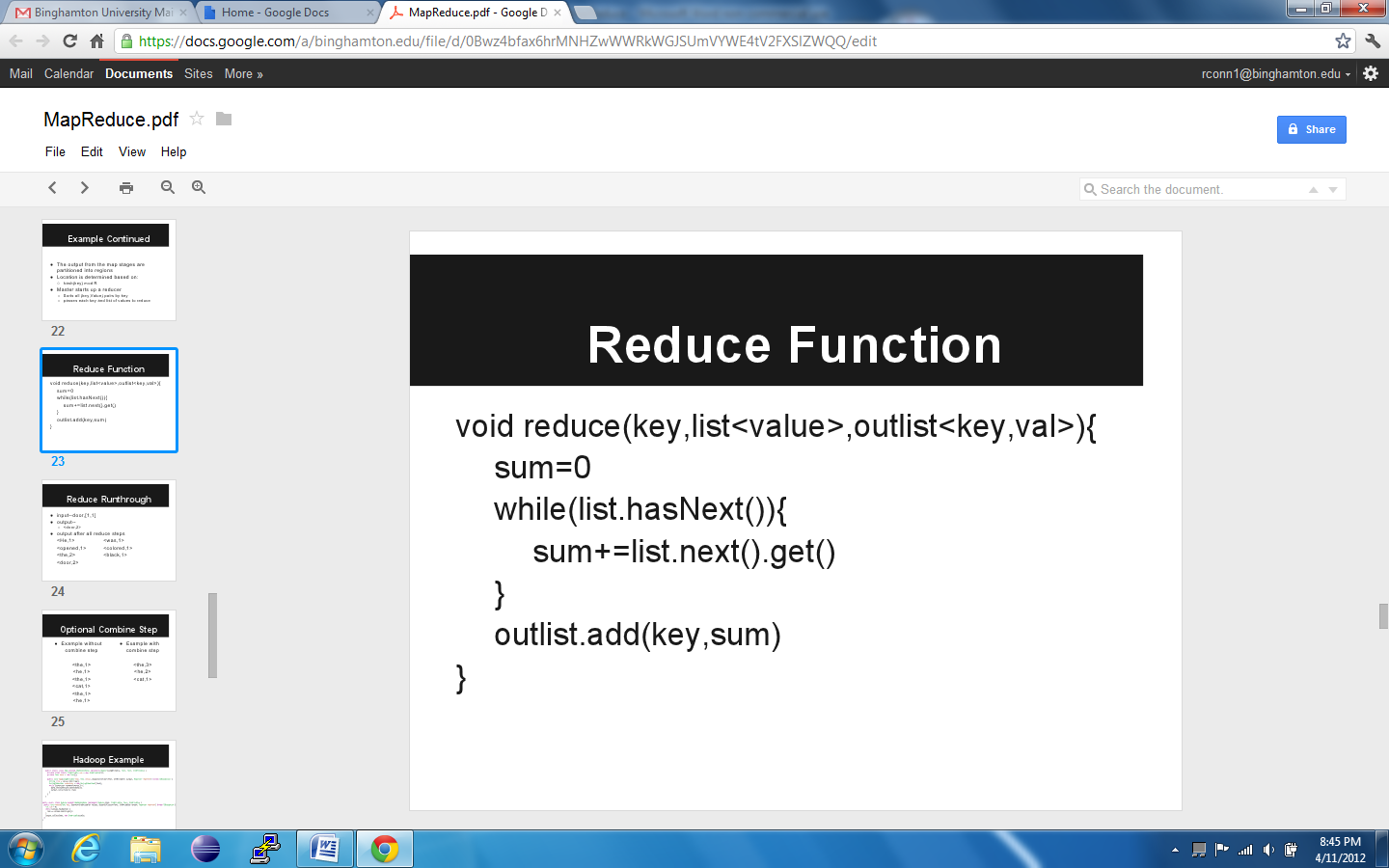


Figure Example of a reduce function for doing a frequency count of words

When a slave has been notified by the master to perform the map phase on a subset of the input data, it obtains this data based on the key received from the master. When the slave receives the data, the first step is to parse the data into new keys. For text input, this means that the new keys are the individual words from the input. The new keys are then given a value based on user defined map computation. The map phase will then store this new key value pair in a file location. The location is usually determined using hash (key) mod R. There are many different slaves performing this map phase simultaneously with all synchronization handled by the master.

As the master gets feedback from slaves saying that they have completed their map phase, the master starts to assign idle slaves to reduce phases. When a master starts up a reduce phase on a slave it informs the slave of the file location for the key value pairs that the slave is to perform work on. Usually there are many more keys than there are reducers. This means that there can be more than one key associated with a file partition. To solve this problem the first thing a reduce slave will do is sort the assigned file location by key. After the sort has finished the slave will then run the reduce code for each key and the list of values associated with that key. This will result in one key value pair holding the final result for that key. The results are then stored in a file location specified by the master. There is usually more than one file location that holds the results because the results are usually put through another mapReduce process. Once all the reduce phases have completed the process comes to the end and the master alerts the user program that the results have been computed, giving the user program the file location containing these results.

For problems that require a lot of reducing an optimization can be added to the process. Normally the map will parse a new key, assign a value to that key, and store its result. This means that the same map slave can produce the same key numerous times all, with each reproduction containing its own value. As a result, this leaves extra work for the map phases. To optimize this the map phase can do a reduction before it outputs its results. This means that there will only be distinct key value pairs produced from each map slave, resulting in more work for the map phase but less work during the reduce phase. If the problem being solved has many reduce steps than it is very beneficial for the map slaves to reduce before they complete. On the other hand for problems that don’t have a lot of reducing it is better to just leave the reducing up to the reduce slaves.

## Benefits

One of the main benefits of that the MapReduce programming model provides is fault tolerance. If you recall the motivational example provided, you could imagine the frustration that losing the word count would incur. The same is true for the MapReduce distributed model. If a job id deployed across thousands of machines, it would be unacceptable for the failure of one to cause the failure of the entire job. Due to the parallel nature of MapReduce jobs, a failed worker’s tasks can easily be reassigned to other workers. It should be noted that a worker stores its data locally, so when a worker fails, all of its completed and uncompleted tasks must be computed by another worker. The master machine then notifies all the other machines about the change of data sources. A more complicated scenario arises if the master machine fails. Luckily, there is only one master, so the probability of failure is low. In case of failure, the master makes periodic checkpoints of its internal data structures. If the master fails, a new master can be started from the old master’s checkpoints. Alternatively, the entire job could be restarted with a new master.

Load balancing is another important part of the MapReduce framework. Load balancing is handled entirely by the MapReduce implementation, and the programmer does not need to write code to aid in the process of load balancing. The programmer does, however, specify the number of map and reduce tasks, which does have an effect on how well the program is load balanced. Ideally, many more map jobs should be created than the number of workers present. If a worker fails, then all of its work must be reassigned. If that worker only had one job to computer, then that entire job must be assigned to another machine, essentially doubling that machine’s load. If a failed worker had many jobs to compute, those jobs could be distributed amongst all of the workers.

Another benefit of the MapReduce framework is the ease of use it provides. In the example above, very little code needs to be written. Splitting and synchronization are both handled by the implementation. This provides a huge timesaving benefit to the programmer. Additionally, the benefits of using a shared library are also gained. Performance improvements made to the implementation increase the performance of all users of the implementation. This allows for the wide dissemination of high performance code. The automatic synchronization of jobs also allows the programmer to keep their traditional, one machine mindset. The code one writes while using MapReduce looks very similar to code that one would write to execute on one machine. In fact, there is very little evidence in the user code that suggests the job can be distributed over hundreds or thousands of computers. Another example of ease of use is the ability to easily deploy new workers. As long as you have a user name on a machine you can assign it to be a worker. The framework will use your username to connect to that machine and start jobs on it. All of this happens with no intervention from the user.

The theme of using one machine vs. using a distributed cluster is discussed many times above. It may seem that for small jobs, it may be faster to use a multicore machine instead of taking the time to deploy the job to a small cluster. The numbers how otherwise, however. The problem is memory. On a single machine with multiple cores, memory becomes a huge bottleneck. Even though the machine may have eight processing cores, it may only have the memory bandwidth to server 6 of them. A distributed cluster on the other hand, has enough memory to serve all the processing power. Figure 5 shows the speedups that can be achieved using MapReduce over multicore processors.

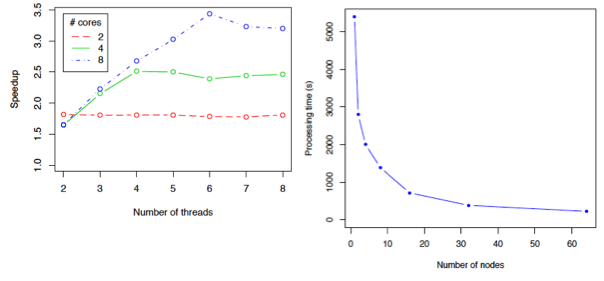


Figure 5 Multicore vs. MapReduce XML parsing.

In the industry, MapReduce has many applications. Many people experience one of its applications every day. When doing a Google search, auto complete suggestions appear in the search box. These suggestions are based off previous searches by users. To compute these, Google collects all the searches that have been made in the past time window (around a month to a week), and then uses MapReduce to count and sort the frequency. These words are then served to the user while they are typing in Google’s search box. Another use in the industry is distributed grep. This can be especially useful if a company has an extremely large codebase they would like to search. Since the codebase is too large to fit into the memory of one machine, it would take hours or days to use one machine to grep through the code. Using MapReduce reduces the grep time to seconds. Log query is another use case. Large data centers generate gigabytes of logs each day. These logs may need to be searched for various properties. Using MapReduce, these logs can quickly be searched.

MapReduce does have some limitations, however. Since the splitting is handled by the implementation, the implementation makes certain assumptions about the format of the data. Usually, the input is either plaintext or XML. The splitter cannot handle complex scientific data. Such a use case is an active area of research. Another problem with MapReduce is that the cluster of computers must be homogeneous. Allowing MapReduce to work with heterogeneous networks is another active area of research.

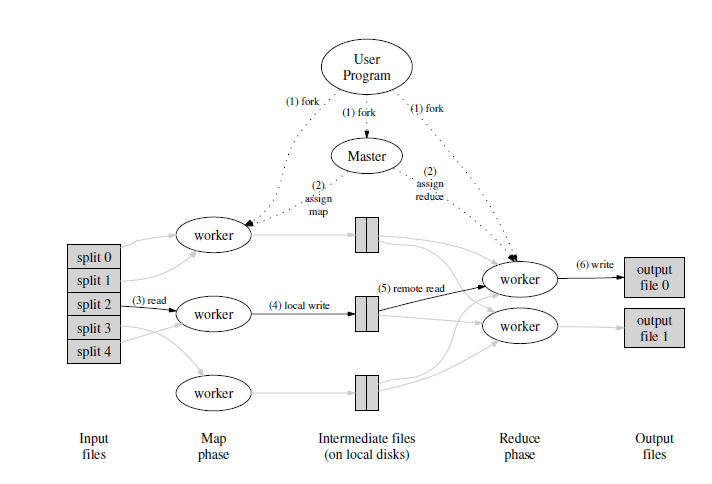


Figure overview of the map reduce process

[1] J. Dean and S. Ghemawat, “MapReduce: Simpliﬁed Data Processing on Large Clusters,”  
Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008