CS 520 Research Paper 1 – MapReduce

Caroline Harrison

May 8, 2020

In modern Big Data analysis, the MapReduce distributed algorithm allows a user to regularly process immense amounts of data by taking advantage of the idea of parallel processing. In the past, processing ever growing amounts of data in memory required more powerful computers, however many companies’ data processing needs were quickly outgrowing the hardware limits of those computers. The solution was to scale horizontally and distribute data processing tasks across multiple machines. Rather than processing a dataset in its entirety, the MapReduce algorithm distributes processing tasks by breaking up the dataset into smaller, more manageable chunks and processing each in parallel. The results of those processes are then grouped together, and the aggregation of that data is then returned to the user.

The MapReduce distributed algorithm was first introduced by Google in 2003, as a method to improve their web page indexing system and to improve the speed of PageRank calculations for the Google web search service[1]. Additionally, Google used the algorithm for large-scale machine learning problems, clustering, data extraction, and graph computations. The developers of MapReduce were inspired by the functional programming language Lisp and the language’s built in map and reduce functions. The solution was necessary as the world wide web was growing rapidly, and while the calculations required for these tasks were straight forward, the processing time was still becoming more and more expensive and developers needed to utilize parallel computing to run these calculations in a reasonable amount of time. Prior to this, distributed computing was often a cumbersome task that required developers to focus on low-level system details. Everything from what machines would be used, to process scheduling and worker synchronization needed to be explicitly stated by the user. MapReduce abstracts the system details from the developer and allows them to focus entirely on the data processing tasks needed[2]. On top of MapReduce’s ability to be highly scalable as more and more data needs to be processed, other developers at Google found the model easy to implement in their own work to help the service grow.

The MapReduce process can be broken out into three main steps – map, shuffle, and reduce – and is written with two user defined functions, the map function and reduce function. The algorithm splits up work amongst a cluster of machines where the one controlling the process is known as the master, and the rest are known as workers. The set of workers is further subdivided with each being assigned a task of either map or reduce.

The first step, mapping, works by accepting an input file or dataset and splitting that data into *M* number of pieces. These are typically 16-64MB in size and are duplicated before being distributed to the cluster of machines[3]. The user’s program is then copied onto the same cluster of machines, where one is designated as the master. The master assigns the map tasks to a set of workers who will each process the user’s map function over their assigned data chunk and produce a set of key-value tuples. These key-value pairs, known as intermediate keys, are stored in memory on the worker machine and the memory location is sent to the master[1]. In some cases when there is significant repetition in the intermediate keys, the user can specify that a combiner function be included in the map task process. A combiner function acts as a mini-reducer for a set of intermediate keys and groups together repeated keys into a single key-(list of values) tuple[4].

The next main step in the MapReduce process, shuffling, begins after all the map workers have completed their tasks and stored their intermediate keys into memory. The number of *R* reduce workers assigned by the master is defaulted to the number of unique keys from the output of all intermediate keys, however a user can specify in a partitioning function how keys should be hashed to define how many *R* reduce workers are needed[3]. The master informs the *R* reduce workers of all the memory locations of the different sets of intermediate keys and the unique key they will process. Each worker reads in their specified key-value tuple from all the memory locations used to store the intermediate keys. The reduce workers are effectively shuffling the data into organized groups.

The final main step, reduce, begins after the after the reduce worker has read-in all the intermediate key data that has been assigned to them. The reducer worker aggregates the intermediate data into a final key-value tuple specified by the user’s reduce function. The result of each reduce worker’s aggregation is then returned to the user as the single final output. The workflow of MapReduce can be seen in Figure 1 below:

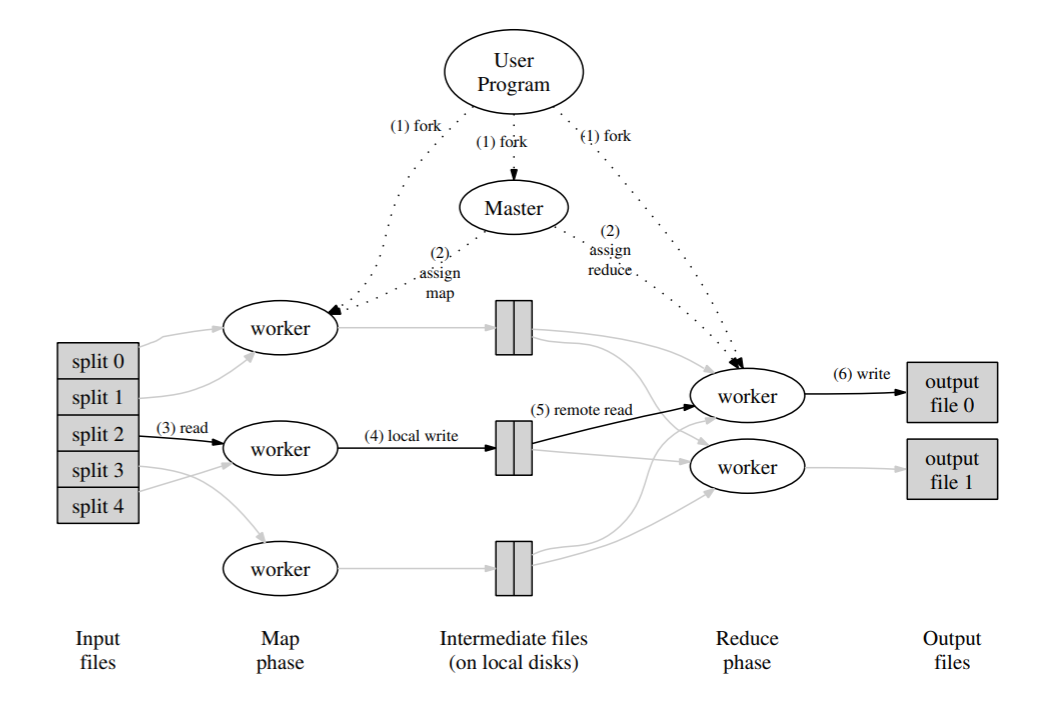


Figure 1: MapReduce Execution[1]

In real-world scenarios, machines can fail and processes could be easily disrupted or impacted by a down worker. MapReduce handles these potential machine failures by having the master periodically check the status of each process worker. If everything is running smoothly, the process worker would send an acknowledgement back to the master. However, if no response is received from the process worker the master assumes that the machine has failed. In this case, the task is re-assigned to a different process worker. Rather than figure out how much of the process the failed worker had completed, the new worker simply starts the task from the beginning.

A simple example of the MapReduce algorithm is finding the word count for a set of documents. While this example would not require MapReduce for only a few documents, a scaled-up version of this example may be the mapping word counts to build a TF-IDF model for some document search query function. In the mapping phase, the user would input the set of documents, and define the <word, occurrence> pairs to return as the intermediate keys. Since some words are often repeated many times in a single document, the user might specify that a combiner function be used as a first round grouping process. The inputted set of documents would be broken into chunks by the master and assigned to different map workers. Those workers would then simultaneously process their chunk of data and store into memory the set of intermediate keys. For example, if the word “Boston” occurred twice in a map worker’s chunk of documents, one of the intermediate keys it would return would be <”Boston”, 2>. As the map workers all finished their tasks, the master assigns reduce workers to remote read in the data from each memory location. The reduce workers would iterate through the intermediate keys and read in their assigned word whenever it appeared. Finally, the reduce worker would sum together the total number of occurrences of the word in the entire set of inputted documents and return to the user the final < word, count > tuples for all the documents. For example, if a second map worker found three instances of the word “Boston” , the reduce worker would read in <”Boston”, 2>, <”Boston”, 3> from the intermediate keys and sum them to <”Boston”, 5>.

The MapReduce model works in a way that solves several computational issues including parallelizing computations, distributing data, and potential machine failures. First, MapReduce parallelizes a process by breaking down data into smaller, more manageable chunks and processing each chunk simultaneously, rather than on the single massive dataset. This solves the issue of having a single task require an extensive runtime that could create a bottleneck in a user’s system. This also solves the issue of scalability, rather than having to upgrade the processing capabilities of your machine to handle the growing data inputs, MapReduce allows a user to scale horizontally by allowing a machine clusters to process more requests simultaneously. MapReduce also solves potential disruptions due to machine failure, which are common real-world issues that can occur when relying on multiple machines to process your data. It does this by duplicating data chunks onto separate machines to reduce the risk of data loss due to machine failure, and by periodically checking the status of each process worker.

MapReduce was incredibly successful at Google and a useful stepping-stone for moving into the age of big data as the platform had to index a rapidly growing network of web pages. However, as Google has grown, they have worked to improve the framework to allow for more complex pipelines in the MapReduce model (FlumeJava)[5], low-latency data processing applications allowing for real-time information to be processed (MillWheel)[6], and systems for incrementally updating the datasets rather than requiring updates to be done in batches (Perculator)[7]. Google’s introduction of MapReduce was also beneficial to the development of other distributed processing and data storage systems.

One popular open source software implementation of MapReduce is Apache Hadoop, hosted by the Apache Software Foundation. At the core of Hadoop’s framework is the Hadoop Distributed File System (HDFS) and Hadoop MapReduce[8]. Both HDFS and Hadoop MapReduce take inspiration from the MapReduce model by distributing data storage and processors across large machine clusters. The two naturally work together as the input source of Hadoop MapReduce is data hosted on the HDFS. Hadoop has been utilized by many big-data companies to store and process petabytes of data. Like Google, Apache has also made significant improvements to the basic MapReduce algorithm in the Hadoop system. For example, while Hadoop was successful at large-scale, batch processes, the development of Apache Spark benefited low-latency machine learning applications through the concept of RDD (resilient distributed datasets). RDD’s are beneficial when data is often re-used for a single process and computations are iterative by letting users perform computations on large clusters of data in-memory[9][10].

The MapReduce distributed algorithm was a successful tool for processing ever-growing big-data processes and analysis. The ability to distribute computations horizontally across multiple machines was an effective utilization of processing power and an easily scalable solution for companies needing to process large amounts of data quickly. Since Google introduced the MapReduce algorithm in 2003, it has been used as the foundation for many distributed file systems and analysis programs that meet the different and evolving needs of the Big Data community.

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." (2004).

[2] Lin, Jimmy, and Chris Dyer. "Data-intensive text processing with MapReduce." *Synthesis Lectures on Human Language Technologies* 3.1 (2010): 1-177.

[3] Leskovec, Jure, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[4] Lin, Jimmy, and Dyer, Chris. *Data-Intensive Text Processing with MapReduce*. United States, Morgan & Claypool Publishers, 2010.

[5] Chambers, Craig, et al. "FlumeJava: easy, efficient data-parallel pipelines." *ACM Sigplan Notices* 45.6 (2010): 363-375.

[6] Akidau, Tyler, et al. "MillWheel: fault-tolerant stream processing at internet scale." *Proceedings of the VLDB Endowment* 6.11 (2013): 1033-1044.

[7] Peng, Daniel, and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications." (2010).

[8] Ghazi, Mohd Rehan, and Durgaprasad Gangodkar. "Hadoop, MapReduce and HDFS: a developers perspective." *Procedia Computer Science* 48.C (2015): 45-50.

[9] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012.

[10] Zaharia, Matei, et al. "Spark: Cluster computing with working sets." *HotCloud* 10.10-10 (2010): 95.