

An Introduction to MapReduce

Brett Walenz

University of Nebraska at Omaha

CSCI 8080 – Design and Analysis of Algorithms

Spring 2009

Abstract

MapReduce is a system architecture and software implementation designed for high volume data processing in a distributed computing environment. The original implementation was designed and developed by Google, and later implemented in the Hadoop library for the Java programming language, as well as the Ruby, .NET, C++, and other programming languages. This paper will discuss the key aspects of MapReduce, including the need and origins of the original concept, how it works, and an analysis of the system design.

Introduction

MapReduce is a system that was designed to solve high volume data processing in a distributed computing environment. Designed and developed by Google, the initial motivation for the development of the system came from the creation of many large scale textual processing tasks that required distribution across many machines to finish in a reasonable amount of time[1]. MapReduce was designed as an abstraction layer, allowing algorithms to be converted to the MapReduce framework without worrying about the details of the data distribution, load balancing, and other problems that come from sharing information across hundreds of machines.

A user implementing an algorithm with the MapReduce system is required to write two pieces of functionality: a *map* function, which takes an input and produces a set of key/value pairs that are used as intermediate objects and sent to the *reduce* function, which unifies equal keys that have been produced by the distributed *map* functions, resulting in a combined derivation according to the algorithms specification.

The Origins of MapReduce

At Google, there were many large scale data operations that were ideal for large scale parallel computation efforts. Three of the notable operations are web crawling, search logs, and click streams. Web crawling is easily parallelized as there is a large degree of independence from document to document. For example, a web crawler parses documents and adds them to an index, allowing ranked retrieval to occur. Before adding to the index, the document must be tokenized (broken apart into single term vectors), analyzed (removing HTML, javascript, etc), and so forth.

Current distributed systems are based on parallel database products, which provide a standard SQL query interface while hiding the implementation details of the parallel process. However, many of the solutions are extremely expensive and SQL can be difficult for programmers to code with as they are used to procedural languages. They can be even more difficult to maintain, especially if the original development team is no longer available. With that in mind, Google developed the MapReduce concept.

Implementation Details

There are two central themes in the MapReduce system. First is the concept of an algorithm requiring conversion to *map* and *reduce* functions. Next, there is the underlying architecture that handles load balancing, data distribution, fault tolerance, and more. To illustrate the first concept, a simple example is shown.

Map and Reduce

The traditional example to explain MapReduce is the problem of counting the number of occurrences of each word in a large collection of documents. It can be seen that the problem is easily parallelizable as each document can be analyzed separately, and then joined at the end of the process with the other documents. In this case, the *map* function takes a *key*, where the key is the document name, and a *value*, which in this example is the contents of the document.

```
/** The map function takes a key and value pair and produces a set of key and
value intermediate pairs. These are then used by the reduce function to
produce a final output. */
function map(String key, String value) {
    for(Word w : value) { //assumes there is tokenization
        putIntermediateValue(w, "1");
    }
}

/** The reduce function takes a key and a collection of values encoded as
Strings. In this simplistic case, the values will always be one, but a smart
map function could count and combine all instance of tokens together. */
function reduce(String key, List<String> values) {
    int count = 0;
    for (String number : values) {
        count += String.valueOf(number);
    }
    outputToMapReduce(count);
}
```

The example is straightforward. The *map* function takes a document name, the key, and its contents, the value, and outputs intermediate values where each intermediate value consists of a token, the key, and a number, the value. In the example above, the *map* function outputs a "1" for every token. A master server is responsible for bundling key and value pairs together, and submitting them to the *reduce* function. In the *reduce* function, the function loops through all the values contained in the collection and produces an aggregated result, which is then output to the MapReduce system.

System Architecture

A MapReduce instance is controlled by a single Master which controls spawning of map and reduce *workers*, which are instances of the map and reduce functions that may be on different machines. As the system progresses through its tasks, the Master is responsible for assigning new map workers, responding to faults (such as a machine not responding or taking a very long time to complete), ensuring that the loads are balanced between machines, and alerting the user program (or code that is executing the MapReduce system) that the operation is complete.

The MapReduce system also handles the concept of *task granularity*, or the allocation of map and reduce phases spread out in a reasonable fashion across the available machines. Google's MapReduce system creates M map pieces and R reduce phases, which ideally are much larger than the number of available machines. The Master requires an amount of overhead, both in the number of scheduling decisions and the amount of memory required for each map or reduce phase. Specifically, in *MapReduce: Simplified Data Processing on Large Clusters*, it is stated that the Master requires $O(M+R)$ scheduling decisions and $O(M*R)$ state storage.

Applications of MapReduce

At this point, there are many diverse implementations of MapReduce. Widespread through the Google network is their own implementation, a combination of C++ and Java, which is the source of the features listed in the original publication. Hadoop, an Apache project written in Java, is used by Yahoo, the Nutch open source search engine, and others, and replicates many of the features from the original publication. There are Ruby, Python, and .NET implementations of MapReduce, and there are commercial service companies dedicated to providing MapReduce support for data centers.

There are many third-party applications that build upon the Hadoop project to provide services. Pig is a platform for analyzing large data sets that consists of a high level language for expressing data analysis programs[3], Mahout is a library that implements many machine learning techniques using the Hadoop library, and HBase is a Hadoop database, which is modeled after Google's Bigtable[4]. Recently, Yahoo! launched the largest deployment of Hadoop with a 10,000 machine Linux cluster[5].

System Analysis

It is fairly intuitive to understand why MapReduce works, provided that the problem is suited to the framework. The framework itself is fairly rigid, providing a two-step process (the map and reduce phases), which takes identical problems and combines them into one final output. MapReduce is conceptually simple, it uses the same concepts and techniques that humans use every day to solve similar problems, such as counting and sorting. However, MapReduce isn't suited for all tasks, such as algorithms requiring multiple steps, as in an assembly line, nor will it handle system queries, such as relational databases, although MapReduce is commonly used to build the query index for information retrieval systems.

In a large scale distributed system, a primary concern is the efficiency and frequency of input/output. This is especially true in the case of MapReduce, as *map* workers submit intermediate key/value pairs to the reduce tasks, who also send output files when their task is complete. For this reason, Google's implementation will attempt to place *map* executions close to the original data set, or a portion of the data set, to conserve bandwidth. When the *map* phase is complete, it stores its intermediate values to the local disk, and local files are only transferred when necessary for the *reduce* phase. Completed reduce tasks are stored on the global file system, which uses the Google File System [6]. In addition to saving bandwidth, these details are a very efficient approach and have the potential to decrease *map* worker running times. Since *map* workers are placed nearby the original data set, the risk of data transfer time being large is lessened.

For the tasks it has been specifically developed for, MapReduce is clearly one of the best. In late 2008, Google announced that it had achieved the current fastest time for the standard terabyte sort benchmark [7], completing in 68 seconds using 1000 machines. Also interesting is that the previous record holder was the Apache Hadoop project, which completed the task in 209 seconds!

The entire system is extremely efficient. A distributed version of Grep, which scans through 10^{100} byte records for a rare three character sequence, completed in just under two minutes on an 1800 machine cluster, with the majority of the time spent setting up the map workers and transferring data [1]. This created 1764 workers spread across the network and one reduce task.

Disadvantages

As mentioned earlier, MapReduce is ill suited to operations requiring many sequential tasks, or requiring fast querying of relational data sets. While the business logic of fault tolerance, distributed systems, and load balancing are separated from the implementation details of the algorithm, many open source applications are still very complex to set up, see the Hadoop Clustering Configuration Page[5] for the complexities involved. Furthermore, across implementations reliability is variable. For instance, in the Hadoop implementation, both the Master and the Job Tracker are single points of failure, and if either one of them does fail, the entire operation is lost.

Another disadvantage is the requirement of the implementation system to have a fallback mechanism to handle "stragglers". Stragglers are machines or worker executions that are running behind the rest of the system, and can cause drastic slowdown in the overall execution time of the system. To counteract this, Google's MapReduce contains a backup monitor that can replace slowly executing tasks with a replica task on another machine. Google's research indicates that not having this feature for the terabyte sort benchmark produced a speed that was 44% slower than with the backup monitor enabled.

Conclusion

MapReduce is a system architecture that has been implemented by many companies and software groups, and has been used for many large scale distributed tasks, including web crawling, machine learning, and query logs. The goal of the system is to provide an abstraction layer between the fault tolerance, data distribution and other parallel systems tasks, and the implementation details of the

specific algorithm. MapReduce is not ideal for algorithms requiring many sequential tasks, nor is it meant for relational database approaches. Implementations using MapReduce currently hold the top two positions in the terabyte sorting benchmark, and have even been used to sort a petabyte of information.

References

[1] [Dean and Ghemawat 2004] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implementation, San Francisco, CA, Dec. 2004.

[2] <http://hadoop.apache.org/pig/>

[3] <http://hadoop.apache.org/hbase/>

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[5] http://hadoop.apache.org/core/docs/r0.19.1/cluster_setup.html

[6] Ghemawat, S., Gobiuff, H., and Leung, S. 2003. The Google file system. *SIGOPS Oper. Syst. Rev.* 37, 5 (Dec. 2003), 29-43.

[7] <http://www.hpl.hp.com/hosted/sortbenchmark/>