

Hadoop

Shawn Jansepar

301041752

Hadoop Overview

Hadoop is a framework that allows you to write software that easily scales to multiple computers, splitting up computations amongst multiple machines. The idea is that there needs to be a way to query extremely large sets of data, and since distributed computing is hard, Hadoop eases the pain for developers. Analyzing massive amounts of data is a huge issue with computing today, and with billions of people using the internet, there is a large need for computations to be made within a matter of seconds. The solution that people have come up with is to divide up the problem into multiple sub-problems, then solve them in parallel across multiple computers. Hadoop uses multiple nodes in a particular way to effectively solve the problem as efficiently as possible. The idea is simple - you write your software following the framework of Hadoop, and the code you write will be as efficient on one computer as it is on thousands.

Hadoop Common Library

The Hadoop common library is a Java library that allows you to work with the subsystems of Hadoop. Hadoop has a set of libraries for the core features, such as MapReduce and the Hadoop Distributed File System, although there are many other projects that have been created to extend Hadoop's functionality. Let's talk about some of these core features!

Hadoop Distributed File System

Hadoop is not a typical piece of software that runs on your operating system that reads and writes to the computer's file system. Hadoop has its own file system - a distributed one that spans across multiple computers at the same time. While file systems typically communicate with the kernel for the read/write of files, the Hadoop Distributed File system uses a network protocol to read and write files to it, although not in a way that typical Network File Systems have been implemented. The difference is that in a typical distributed file system, it is not clear that the disks you are reading/writing to are over a network. When you execute commands to read and write files to your home folder, it may seem like everything you have done has happened on your computer, but if this folder is mounted on a network disk as opposed to a disk on your computer, these commands are not doing what you may think. The commands are actually calling a library created for the network protocol, and these libraries implement

remote call procedures. This has been the most common way to implement distributed file systems, such as NFS (Network File System), but Hadoop's DFS takes a slightly different approach.

The Hadoop Common Library has a set of libraries that allow you to access the HDFS, and while it may seem no different from NFS, it most definitely is! You are unable to transparently place the file system into the Unix commands like NFS does. You use the Hadoop Common Library to access the file system through an API written in Java. The downside here is that you have to use an actual Java program to access your file system through Unix commands (which try to mimic the Unix file system the best way they can). Let me make this clearer with some simple commands:

Listing the contents of the root folder:

Unix

ls

HDFS

/bin/hadoop dfs -ls /

At this point, I think you understand that there are differences on the surface, but the real question is, what is the difference underneath? To state it simply, NFS is basically a disk on a computer somewhere on the network, and any computer can put this network disk onto its own, thus the disk is "distributed" to multiple computers. HDFS is an entirely different beast. The downfall of NFS is that while it is convenient that multiple computers can have access to the same hard disk, it still does not solve the problem of reliability, and it does not solve the constant problem of scaling to meet the need for more and more data. It will be best to explain how the file system works first, and then explain how it solves these problems afterwards.

HDFS does not have one hard disk that stores information; it actually copies the data across each node (which is just a computer running somewhere). It also splits the data into chunks (much like block size on a typical hard disk) across each node, and there is a central location that records where each part of each file is. HDFS has two separate types of nodes. The type of node that contains the information about which nodes contain which chunks of data is called a *NameNode*. There is always only one NameNode in an HDFS setup, and without going into too much detail, it is simply too difficult to scale a NameNode to multiple nodes that work together. Having only one NameNode has a large downside - the NameNode becomes extremely important. If you lost the NameNode then your entire file system would be rendered useless. The good thing is that it's easy to replicate a NameNode, so if your main NameNode went down, you could switch to the one that is backing it up with some sort of switching mechanism. The nodes which contain the actual chunks of data are called *DataNodes*. You can add

as many of these DataNodes to your HDFS setup as you please, because the performance of the HDFS will not be affected – HDFS is designed to add scale linearly. This amount of text may have you a little bit confused, let me present you with a diagram that clears up any confusion with NameNodes and DataNodes:

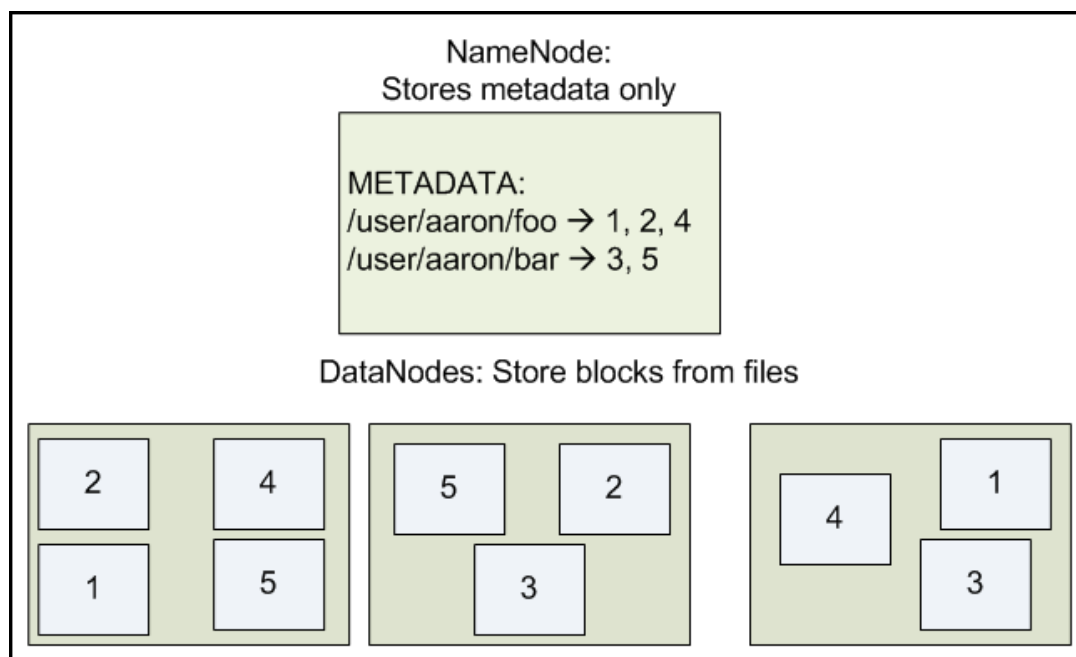


Figure 2.1, Module 2 from the Yahoo Hadoop Tutorial (reference at bottom)

Now that you understand the difference between NameNodes and DataNodes, let's go back to the problems that NFS has, and why the HDFS solves them. The first problem I mentioned is the issue of reliability. A NFS simply has a disk on a computer which is backed up by some mechanism, and while backups are a good thing, they are still not optimal. With HDFS, you have the option of data replication, which is set in the Hadoop configuration. What this allows you to do is have your file chunks replicated across multiple nodes. So for example, if you had three DataNodes, and a replication factor of 3, then each chunk you have for your file will be on every DataNode in your setup. This means that if you lose a server, that's okay because the data is still on two other DataNodes. A fantastic way to ensure reliability! Another issue with NFS is that it does not scale well. It is hard to add more space to a currently existing NFS setup because you would have to add more hard drives to the physical box where the NFS resides, or you need to setup a new NFS instance all together. This requires your application that uses NFS to be configured to use a newly created NFS instance, which is not a very dynamic solution. With HDFS, DataNodes can be added at any time, making the solution very dynamic and easy to maintain.

There are a lot more advantages to the HDFS than what I have talked about, and those advantages stem from what I am going to talk about next: the MapReduce

framework.

MapReduce

The technology that makes Hadoop a distributed system is called MapReduce. It works by defining a set of tasks, and performing those tasks on multiple computers in parallel. What you first do is “Map” your problem, which is the process of writing some functions that allow you to arrange the data that you’re looking for. After you’ve mapped that data, you “Reduce” that set of data to be merged into the results that you want. Now, this may still sound confusing, but just make sure that you note that the step “Map” and the step “Reduce” are two very different, but equally important tasks. If you are familiar with SQL, the “GROUP_BY” query has very similar intentions as MapReduce, although it is not nearly as efficient or scalable. The goal here is to give you a high level understanding of how MapReduce works conceptually, and how it works so well over a large cluster of computers.

Let’s talk about the mapping portion first. When writing any MapReduce code, you first have to define the mapping portion of your problem. This is a bit hard to describe, so let’s go through a great example from the Google MapReduce tutorial¹. Say you wanted to look through some logs on Apache, and you wanted to see how many times a particular URL was accessed. First, you would write a map function, which simply takes all of the occurrences of a URL, and maps the occurrence to the output.

```
mapper(log):  
    for line in log.contents():  
        if line.hasURL():  
            output (line.parseURL(), 1)
```

The output of this would be based on a list of (key, value):

```
/homepage/ 1  
/homepage/ 1  
/homepage/ 1  
/about/    1  
/about/    1  
/helloworld/ 1
```

Basically, the map function finds the data that you want and outputs it for the next task

¹(sudo code and output made by me, idea for example comes from the Google MapReduce tutorials list of examples).

to handle (the reducer). The reducer is what actually does the calculation of what you want to know, which in our example, is the number of occurrences of each URL. Here is some code for our reducer in this example:

```
reducer (url, values):  
    sum = 0  
    for value in values:  
        sum = sum + value  
    output (url, sum)
```

Notice that in the reducer, it only takes the input of a URL. That's because the task sent to the reducer is expected to handle the occurrences of only that URL, and the reducer function will be run for each individual URL. This means that there is one task per unique URL. This will be explained in more detail soon.

Based on the mapper output, the reducer output would look like this:

```
/homepage/    3  
/about/       2  
/helloworld/  1
```

As you can see, using the combination of these two functions, we were able to successfully find the number of occurrences of each URL. Now, this might all seem simple, especially since we have not taken multiple nodes into consideration here. But I will try to explain how this still works with a distributed set of nodes processing all of this data, and I will continue with our example to do this.

One important detail about MapReduce on multiple computers is the concept of a JobTracker and a TaskTracker. A JobTracker is what distributes and manages all of the tasks on each node, and the TaskTracker is actually monitoring each individual task, and in order to make sure each task is correctly being processed, it is in constant communication with the JobTracker (you can just envision the JobTracker as a manager, and the TaskTracker as the employees). Also, if you remember from the HDFS, there was a NameNode and a DataNode. These match up to our trackers here, where there is one JobTracker, which is on the same box as the NameNode (both are like central managers), and TaskTrackers are on the same box as each DataNode. There are a lot of reasons for having MapReduce implemented in this way, mostly for reliability reasons, but this is unfortunately out of the scope of our overview.

So, let's start off assuming we have four nodes in our Hadoop setup: one NameNode/JobTracker, and three DataNodes/TaskTrackers. Now let's say we had

multiple Apache log files across our HDFS (hopefully you remember this from earlier, the Hadoop Distributed File System!). First, we would run our mapper on every log file in the file system that we want to inspect. A JobTracker would send these tasks to the TaskTrackers, and we would run the mapper function on each log file. Afterwards, we would want to perform our reduce step, and this is where it gets interesting. As I mentioned before, the reduce function has an argument of a URL. Therefore, whichever box that is processing a particular URL (say for example, /homepage/), then that means the node processing this has to have all of the output from all three nodes onto this one node. And this is exactly what Hadoop does – it exchanges data for processing on each individual node. For example, let's say in the output of each logfile on each node, there was the following:

```
/homepage/    1
/homepage/    1
...
...
(1000 of these on each box, so 3000 total)
```

Therefore, whichever box that is in charge of running the “Reduce” step on the URL “/homepage/” will need all of that output, and during the switch from Map to Reduce, this data is exchanged in all of the proper ways for correct processing. All of the output from the map step ends up on the node doing the processing for that particular URL. It is a bit hard to grasp, here is an example which may help you understand what is going on:

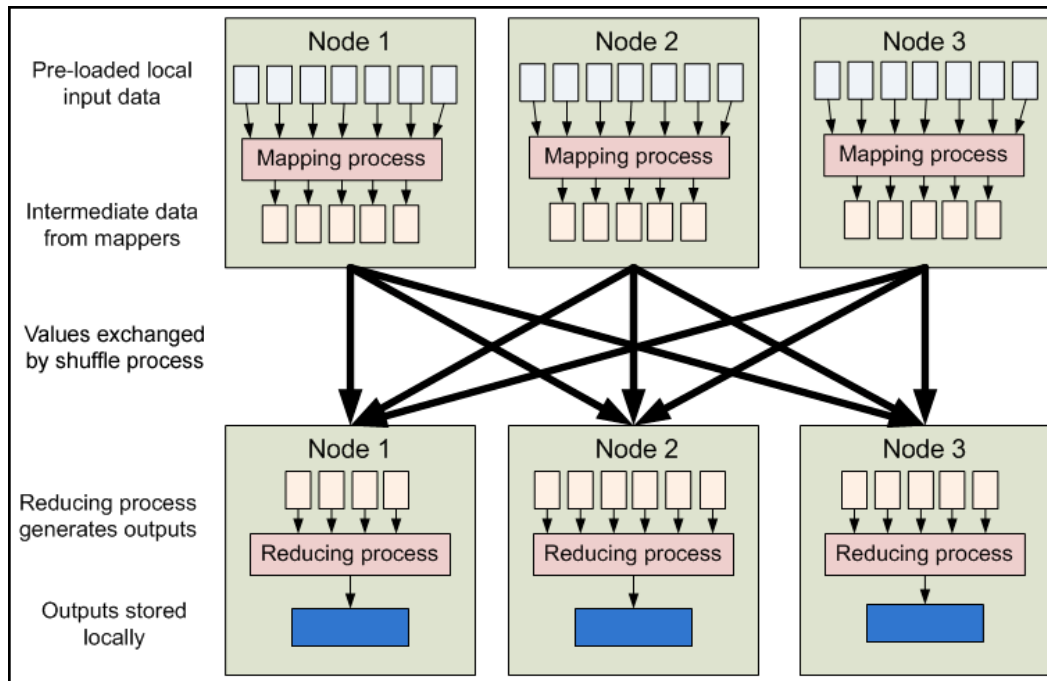


Figure 4.4, Module 4 from the Yahoo Hadoop Tutorial (reference at bottom)

Summary

Hadoop combines a lot of different sets of technologies which allow the platform to be very powerful. The great thing is that you are not completely limited by what you are provided with. If you decide that you don't want to use HDFS as your file system solution, then you can just as easily use some sort of NoSQL database to meet your needs. There is a lot that wasn't covered in this tutorial, such as any actual implementation, or exactly how Hadoop can survive crashing nodes and other potential disaster scenarios! Of course, if you are only working on small datasets, then Hadoop may be overkill, but if you intend to have to do computations over terabytes of information, Hadoop is surely the way to go.

Bibliography

Yahoo! Hadoop Tutorial

<http://developer.yahoo.com/hadoop/tutorial/>

Google MapReduce tutorial

Introduction to Parallel Programming and MapReduce

<http://code.google.com/edu/parallel/mapreduce-tutorial.html>