**Apache Hadoop and Map Reduce**

Independent Study

kumar

**IntroDUCTION**

Documented the learning outcomes of Hadoop and Map Reduce.

**Software Versions:**

**Operating System: Ubuntu Server 12.0**

**Hadoop: 1.03**

**Java: 6**

Table of Contents

INTRODUCTION3

**What is Hadoop**3

**Big Data** 3

**Map Reduce ………………………………………………………………………………… 4**

The Hadoop Distributed File System (HDFS)6

**Configuring HDFS** 9

**HDFS Commands**11

HADOOP MAP REDUCE JOBS …………………………………………………………….13

**Example 1(Word Count) ……………………………………………………………. ……………………………………..14**

**Example 2 (Max Temperature) ……………………………………………………………………………………………17**

**REFERENCES ……………………………………………………………………………….**18

**1.Introduction:**

**What is Hadoop?**

Hadoop is a software framework from Apache Software Foundation to build application processing big data in distributed environment. Hadoop implements MapReduce computational paradigm to process the distributed data. Hadoop also provides following components to perform a reliable and scalable distributed computing.

HDFS: Hadoop distributed file system named HDFS to process the big data in distributed fashion.

Hadoop Yarn: A framework for MapReduce job scheduling and cluster resource management.

**Big Data**

“Big Data can be very small and not all large datasets are big.”( Rindler & Mike 2.0, 2012)

Volume alone won’t define characteristic of big data. IBM defines big data characteristics as below

1. Volume – a single jet engine can generate 10TB of data in 30 minutes, likewise smart meters and industry equipment’s will generate similar volume of data.
2. Velocity – Social media data streams, Twitter generates 8TB per day and Facebook generates 10TB per day.
3. Variety – 80% world’s data is unstructured. The variety of data is increasing with new business and marketing strategies.

Additionally, value could also be defining character of big data.

1. Value – The value of smaller data hidden inside bigger data is might be valuable, extracting and identifying the valuable data from bigger superset will be challenging.

**MapReduce**

1. Usually big data is distributed in various systems, to process the distributed data we need parallel process the data. MapReduce programming model is a way to parallel process huge amount of data. MapReduce does the computation in two-step map and reduce. Map function process a key/value pair and generate intermediate key/value pairs and reduce function will merge all the intermediate values associated with the respective key.

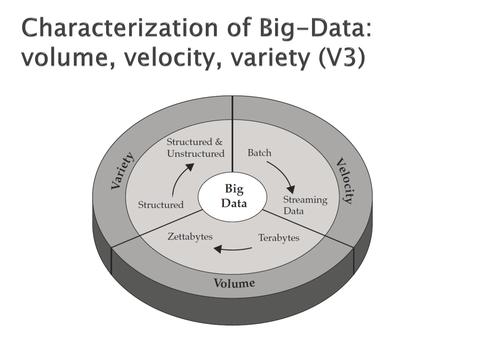


Figure 1.1 Big Data ( Zikopoulos, P. C., Eaton, C., DeRoos, D., Deutsch, T., & Lapis, G. ,2012 )

**Programming Model:**

In MapReduce, the input file is sliced into multiple key value pairs and then mapped to the mapper function, which can run in different domains (CPU). Mapper function creates intermediate key pairs as specified in the map function. Reduce function will collect the intermediate keys from the different domains/machines and produces a final result.

The figure 1.2 will explain the working of MapReduce Model (Dean, J., & Ghemawat, S., 2008).

**MapReduce Operation Details:**

Below steps are going to explain the process of MapReduce programming model.

1. At First, the input file to the user program is fragmented into 16 megabytes to 64 megabytes (MB) per piece by the MapReduce library.
2. MapReduce library then starts many copies of the user program in cluster of machines.
3. Among various copies of the program one will act as a master and others as workers. The master assigns tasks to the workers, there will be M map tasks and R reduces tasks to assign.

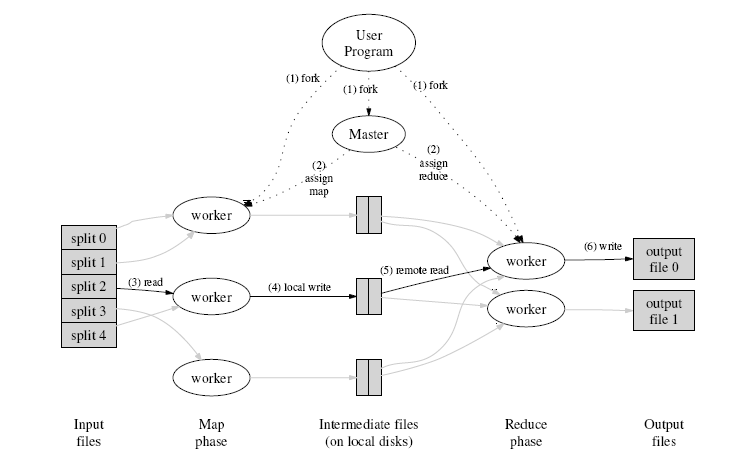


Figure 1.2 Map Reduce

1. The worker with a map task reads its corresponding fragment of input file. It then parses its key/value pairs out of input data and passes each pair to user defined map function for processing. The processed data is buffered into memory with an intermediate key/value pair.
2. Periodically, the buffered intermediate key/value pairs are written to local disk and partitioned into R regions by the partitioning function.
3. The location of buffered pairs in local disk is passed back to the master task. Master task then forwards the location to the reduce workers/tasks.
4. When Master notifies the reduce worker, the reduce worker gets the data from the location of map function using remote procedure calls. When all the intermediate data is received, it is sorted and grouped together. If the intermediate data is too large to fit in memory, an external sort is used.
5. After sorting intermediate data is processed by the reduce function. Each unique intermediate key and corresponding value is processed and append to the final output file for this reduced partition.
6. After the entire map and reduce tasks are completed, the master wakes up the user program. At the same time MapReduce call in the user program returns back to the user code.

After all the above steps, the output will be available in R output files.

The master keeps track of all worker process, if one fails it will reschedule to an available worker. If a map task fails, it is task is re-executed because the data will be in the local disk of failed machine and therefore in accessible.

The failed reduce task need not be re-executed because their output is stored in global file system.

**2. The Hadoop Distributed File System (HDFS):**

HDFS is a special type of file system used to process data in distributed fashion. Moving computation is easier than moving data (Borthakur, 2008). HDFS enables that by distributing and replicating the files in blocks, to various machines to process. Distributing and replicating the data in multiple machines, ensure the availability and durability of failure in very parallel applications. HDFS is designed to store large volume of data in multiple machines. Unlike NFS, the Network File System that is ubiquitous in distributed file system, which stores files on a single machine.

The design of HDFS is based on the design of Google File System (GFS) described in this paper <http://research.google.com/archive/gfs.html>.

HDFS is a block-structured file system: the files are broken into fixed size of 64 MB unlike other file system splits in order of 4 or 8 KB. This large magnitude block size allows HDFS to decrease the amount of metadata and fast streaming reads of data. The blocks of data are stored across a cluster of one or more machines to ensure the availability of file on failure of machines.

HDFS has two major parts for its operation Name Node and Data Node. Name Node is a single machine, which has the metadata information about all the blocks. HDFS can have more than single Data Node. Data Node is where actual block of data is stored. Name Node is a single point of failure. Loss of single Data Node doesn’t not fatal to the cluster since the data is replicated among various Data Nodes where as Name Node failure is fatal for Hadoop Jobs.

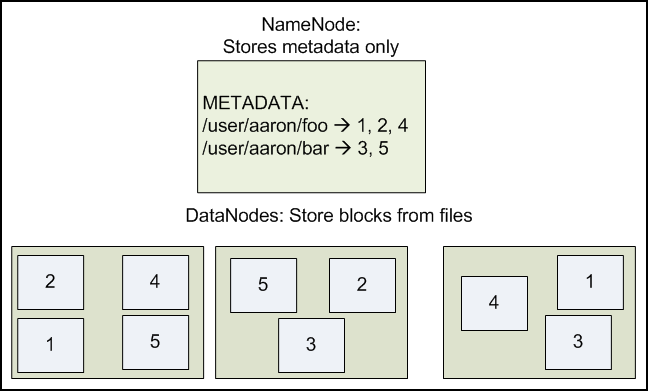
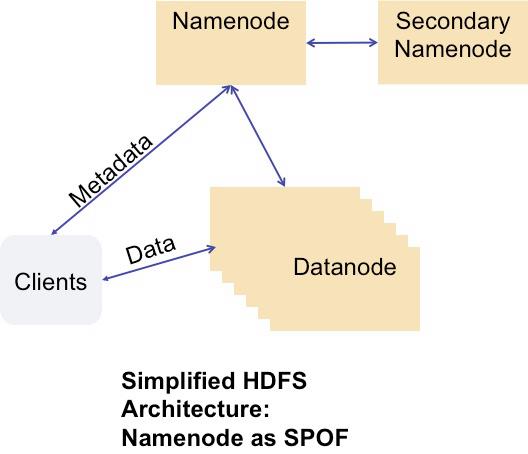


Figure 2.1: Data Nodes holding blocks of multiple files with a replication factor of 2. The Name Node maps the filenames onto the block ids.

Facebook and few other companies have redesigned the HDFS Name Node architecture to handle single point of failure. A secondary stand by Name Node is used to replace the primary Name Node incases of failure. Figure 2.3 shows the Facebook architecture of Name Node (Avatar Node) for High Availability.

**Figure 2.2 HDFS Architecture**

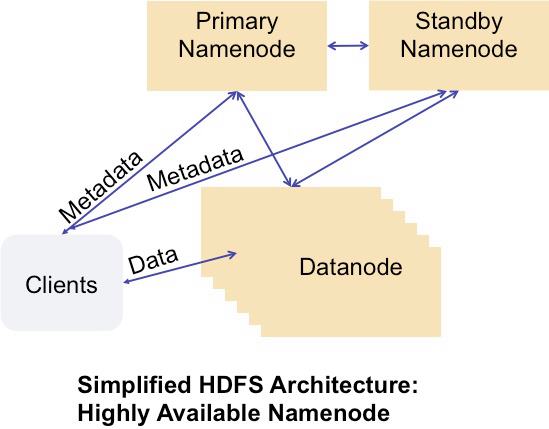


Figure 2.3 Facebook Architecture

HDFS is designed to overcome the problems in other distributed file systems and NFS. HDFS has following advantages than other distributed file system.

* HDFS scales horizontally based upon number of Data Nodes and its storage capacity HDFS can store large amounts of data (terabytes or petabytes).
* HDFS stores data reliably. Based upon the replication factor (Default 3) data reliability increases.
* HDFS provides fast and scalable access to the information. Name Node doesn’t involve in transfer of data. Data Nodes are responsible for data transfer, so increase in number of machines enables HDFS to serve more clients.
* HDFS well integrated with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

Advantages of HDFS such as fast data transfer and scalability comes with few trade offs.

* Due to bigger Block size of data in HDFS random seek times at an arbitrary position in files comes with the cost.
* Data is stored in write once and read many fashion. Earlier version of Hadoop doesn’t support file append. From Hadoop 0.20 data appends to end of file is available.
* Large block size of data and sequential read nature, local cache is not possible. Data has to re-read from HDFS source incases of failure.

**2.1 Configuring HDFS**

Assumed Hadoop environment setup is done, if not please refers to Hadoop Installation guide document.

The following settings are necessary to configure HDFS:

|  |  |  |
| --- | --- | --- |
| **key** | **value** | **example** |
| fs.default.name | *protocol*://*servername*:*port* | hdfs://alpha.milkman.org:9000 |
| dfs.data.dir | *pathname* | /home/*username*/hdfs/data |
| dfs.name.dir | *pathname* | /home/*username*/hdfs/name |

fs.default.name -- Name Node Machine URL, Data Node will register with it.

dfs.data.dir – Data Node directory path

dfs.name.dir – Local Name Node directory

Then make the directories necessary:

|  |
| --- |
| hduser@ip-10-31-220-173:$ mkdir -p $HOME/hdfs/data  hduser@ip-10-31-220-173:$ mkdir -p $HOME/hdfs/name |

Go to /usr/local/hadoop/conf / folder and override the hdfs-site.xml settings with the below settings

|  |
| --- |
| <configuration>  <property>  <name>fs.default.name</name>  <value>hdfs://localhost:9000</value>  </property>  <property>  <name>dfs.data.dir</name>  <value>/home/hduser/hdfs/data</value>  </property>  <property>  <name>dfs.name.dir</name>  <value>/home/hduser/hdfs/name</value>  </property>  </configuration> |

**2.2 STARTING HDFS**

For first time setup, we have to format the HDFS using below comment.

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./hadoop namenode -format |

Start dfs with the following command

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./start-dfs.sh |

**2.3 HDFS Web Interface**

HDFS exposes a web server to monitor and file browsing. Name Node can be accessed by <http://namenodeurl:50070/>. 50070 is the default port. Data Node are exposed http://datanodeurl:50075/ .

Example:

<http://ec2-54-234-165-21.compute-1.amazonaws.com:50070/> - Name Node

<http://ec2-54-234-165-21.compute-1.amazonaws.com:50075>/ - Data Node

You can get the same report in command prompt using below command.

|  |
| --- |
| ./hadoop dfsadmin -report |

**2.4 HDFS Commands**

Interaction with HDFS can be done using command line interface and programmatically. Command Line interface is done through modules ‘dfs’ and ‘dfs-admin’.

Hadoop command line interaction mostly happen through below syntax

ubuntu@ip-10:hadoop$ ./hadoop *moduleName* *-cmd* *args...*

**Listing Files:**

|  |
| --- |
| ubuntu@ip-10-31-220-173:/usr/local/hadoop/bin$ ./hadoop dfs -ls |

**Make Directory:**

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./hadoop dfs -mkdir user |

**Upload a File:**

Transferring single file to HDFS

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./hadoop dfs -put test.txt /home/hduser/ |

Transferring Multiple Files

Create a new directory and copy all the files you need to transfer in it. I created directory named ‘myFiles’ and have few files in it. To move all the files just execute below command

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./hadoop dfs -put myFiles /home/hduser/ |

**Retrieving data from HDFS:**

**Cat command**

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./hadoop dfs -cat test.txt  sdasdasdasdasdas |

Get Command – To copy the file from HDFS to local machine

|  |
| --- |
| hduser@ip-10-31-220-173:bin$ ./hadoop dfs -get test.txt /home/hduser/myFiles/input1.txt |

**Shutting Down HDFS:**

If you want to shut down the HDFS functionality of your cluster, you should login into the Name Node machine and run the below command.

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./stop-dfs.sh |

Above are few basic and ubiquitously used commands, apart from this dfs module support many commands which can referenced from here <http://hadoop.apache.org/docs/r0.18.2/hdfs_shell.html>

**Using HDFS Programmatically**

HDFS can be manipulated programmatically coupling with Hadoop map reduce job. See sample code for Java-based HDFS access.

**3.0 Hadoop Map Reduce Job:**

For running Hadoop MapReduce Job we have start Task tracker and Job Tracker, assuming HDFS running already.

Instead we can start all the process at once by running the below script.

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop/bin$ ./start-all.sh |

**Running an example program:**

|  |
| --- |
| hduser@ip-10-31-220-173:/usr/local/hadoop$ hadoop jar hadoop-examples-1.0.3.jar pi 10 1000 |

**Output:**

Wrote input for Map #1

Wrote input for Map #2

Wrote input for Map #3

...

Wrote input for Map #10

Starting Job

INFO mapred.FileInputFormat: Total input paths to process: 10

INFO mapred.JobClient: Running job: job\_200806230804\_0001

INFO mapred.JobClient: map 0% reduce 0%

INFO mapred.JobClient: map 10% reduce 0%

...

INFO mapred.JobClient: map 100% reduce 100%

INFO mapred.JobClient: Job complete: job\_200806230804\_0001

...

Job Finished in 34.679 seconds

Estimated value of PI is 3.14080000000000000000

**Writing a MapReduce Job:**

Hadoop Map Reduce Jobs are split into three major parts.

Mapper – Maps the input file into key value pairs to process

Reducer - Reducer summarizes the mapped key value pair to desired result

Driver – This tells Hadoop when and what MapReduce job process to run.

Combiner (optional) - Performs the reduction step on the output of each mapper before sending it to the reducer. Can reduce load on the reducer if there are many mappers

**Word Count Example:**

**Mapper Code:**

|  |
| --- |
| import java.io.IOException;  import java.util.StringTokenizer;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.io.Writable;  import org.apache.hadoop.io.WritableComparable;  import org.apache.hadoop.mapred.MapReduceBase;  import org.apache.hadoop.mapred.Mapper;  import org.apache.hadoop.mapred.OutputCollector;  import org.apache.hadoop.mapred.Reporter;  public class WordCountMapper extends MapReduceBase  implements Mapper<LongWritable, Text, Text, IntWritable> {  private final IntWritable one = new IntWritable(1);  private Text word = new Text();  //overrides the base mapper  public void map(WritableComparable key, Writable value,  OutputCollector output, Reporter reporter) throws IOException {  String line = value.toString();  StringTokenizer itr = new StringTokenizer(line.toLowerCase());  while(itr.hasMoreTokens()) {  word.set(itr.nextToken());  output.collect(word, one);  }  }  } |

Reducer:

|  |
| --- |
| import java.io.IOException;  import java.util.Iterator;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.io.WritableComparable;  import org.apache.hadoop.mapred.MapReduceBase;  import org.apache.hadoop.mapred.OutputCollector;  import org.apache.hadoop.mapred.Reducer;  import org.apache.hadoop.mapred.Reporter;  public class WordCountReducer extends MapReduceBase  implements Reducer<Text, IntWritable, Text, IntWritable> {  public void reduce(Text key, Iterator values,  OutputCollector output, Reporter reporter) throws IOException {  int sum = 0;  while (values.hasNext()) {  IntWritable value = (IntWritable) values.next();  sum += value.get(); // process value  }  output.collect(key, new IntWritable(sum));  }  } |

**Driver Class:**

|  |
| --- |
| import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapred.FileInputFormat;  import org.apache.hadoop.mapred.FileOutputFormat;  import org.apache.hadoop.mapred.JobClient;  import org.apache.hadoop.mapred.JobConf;  public class WordCount {  public static void main(String[] args) {  JobClient client = new JobClient();  JobConf conf = new JobConf(WordCount.class);  // specify output types  conf.setOutputKeyClass(Text.class);  conf.setOutputValueClass(IntWritable.class);  // specify input and output dirs  FileInputPath.addInputPath(conf, new Path("input"));  FileOutputPath.addOutputPath(conf, new Path("output"));  // specify a mapper  conf.setMapperClass(WordCountMapper.class);  // specify a reducer  conf.setReducerClass(WordCountReducer.class);  conf.setCombinerClass(WordCountReducer.class);  client.setConf(conf);  try {  JobClient.runJob(conf);  } catch (Exception e) {  e.printStackTrace();  }  }  } |

Compiling Java Files:

|  |
| --- |
| javac -verbose -classpath /usr/local/hadoop/hadoop-core-1.0.3.jar:/usr/local/hadoop/lib/commons-logging-1.1.1.jar -d class WordCount.java |

Creating Jar Files:

|  |
| --- |
| jar cf wordcount.jar \*.class |

Running the job:

Run below command to run word count example

|  |
| --- |
| $ hadoop jar /usr/local/hadoop/hadoop-examples-1.0.3.jar wordcount text.txt output |

Example 2:

Determine the maximum temperature for each year from National Climatic Data Center weather data. Referred From Lecture Notes Prof. Alan Kaminsky RIT – Department of computer science.

<http://www.cs.rit.edu/~ark/730/>

See the sample code folder for code and sample output.

Monitoring job:

Using below commands job monitoring and tracking could be done.

**$ hadoop job -help**

Other Tools:

Zookeeper – A high-performance coordination service for distributed applications. If you are running 100 of clusters and 1000 machines it helps in co-ordination.

HBASE – A distributed database that supports structured data storage for large tables. It is highly scalable horizontally

Hadoop YARN - A framework for job scheduling and cluster resource management. It is introduced from Hadoop version 2.0

References

Dean, J., & Ghemawat, S. (2008, January). MapReduce: Simplified data processing on large clusters. *Communications of the ACM - 50th Anniversary Issue: 1958 - 2008, 51*(1), 107-113. doi:10.1145/1327452.1327492

Ghemawat, S. Gobioff, H. and Leung, S.-T. [The Google File System](http://research.google.com/archive/gfs.html). Proceedings of the 19th ACM Symposium on Operating Systems Principles. pp 29--43. Bolton Landing, NY, USA. 2003. © 2003, ACM.

Borthakur, Dhruba. [The Hadoop Distributed File System: Architecture and Design](http://hadoop.apache.org/common/docs/r0.18.3/hdfs_design.html). © 2007, The Apache Software Foundation.

Oracle (Ed.). (2012, January). *Big Data for the Enterprise*. Retrieved from http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf

Rindler, A., & MIKE2.0. (2012, August 9). *Big data definition* [Wiki article]. Retrieved September 13, 2012, from MIKE2.0 website: http://mike2.openmethodology.org/w/index.php?title=Big\_Data\_Definition&oldid=18517

Zikopoulos, P. C., Eaton, C., DeRoos, D., Deutsch, T., & Lapis, G. (2012). *Understanding big data*. Retrieved from <http://www-01.ibm.com/software/info/rte/bdig/bdwa-7-post.html>

Kaminsky, A. Distributed computing course winter quarter 2012. Retrieved from <http://www.cs.rit.edu/~ark/730/>

Yahoo, "Hadoop Tutorial from Yahoo!" by [Yahoo! Inc.](http://developer.yahoo.com/hadoop/)