**Term Project Final Report**

**Parallel Processing with MapReduce in Big Data**

**Qian Mai, Bogki Yun**

**CSCI 5551 Parallel and Distributed Systems**

**Computer Science and Engineering**

**University of Colorado at Denver**

1. **Abstract**

With the fast development of data analysis, data structure and data processing, a popular term is widely used these days to describe the significant growth of data - Big data. It involves any collection of data sets which are large in size and complex in structure, thus becoming difficult for traditional data processing applications to handle. However, using a MapReduce programming can help simplify big data analysis [1][4].

MapReduce is a programming model and an associated implementation for processing and generating large data set with a parallel, distributed algorithm on a cluster. It is composed of two procedures: Map() and Reduce(). Map() procedure that performs filtering and sorting while Reduce() procedure performs a summary operation [3].

To storage and process large-scale data-set on clusters of commodity hardware, we can use an open-source software framework called Hadoop [6], which is an Apache top-level project being built and used by a global community of contributors and users. Hadoop Distributed File System (HDFS) is a distributed file system which stores data on commodity machines, providing very high aggregate bandwidth across the cluster [2][7].

Our goal, in this term project, is the comparison the MapReduce program performance at each node which is single, 2, 3, and 4 of Hadoop system. We will install a Hadoop on 5 machine with VirtualBox and run the MapReduce program which is counting word at a bunch of web crawl dataset (approximately 3.1GB) by Amazon S3 [5]. And we will evaluate and analyze the results.

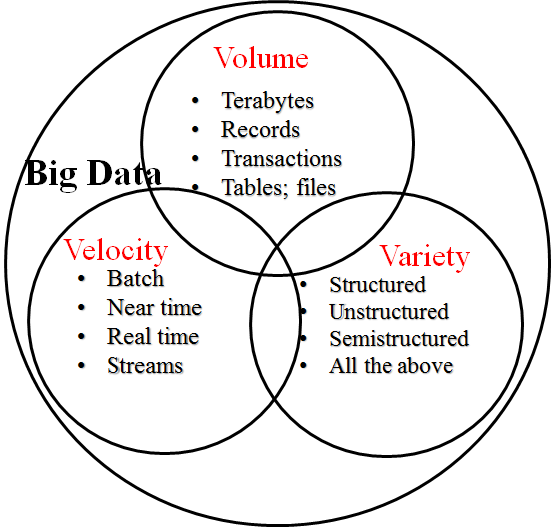
1. **Motivation and Problem**

Currently, big data is popular and maybe as important to business and society in many aspects- as the Internet has become an indispensable part of our daily life. However, the challenges of handling big data usually include analysis, capture, curation, search, sharing, storage and transfer, visualization, and privacy violations. A MapReduce is known for a prominent parallel data processing tool and is gaining significant momentum from both industry and academia as the volume of data to analyze grows rapidly. As is known to all, MapReduce performance is highly related to Big Data system environment, for example, the number of nodes in Hadoop, some configuration such as Shuffle, Combiner class. That’s why we choose this topic as our term project.

1. **Background knowledge and Introduction**

**3.1 Big Data**

Big data is an all-encompassing term for any collection of data sets so large and complex that it becomes difficult to process using traditional data processing applications. Big data is composed of three parts: Volume (terabytes, petabytes), Variety (social networks, blog posts, logs, sensors, etc.) and Velocity (real-time or near-real-time).

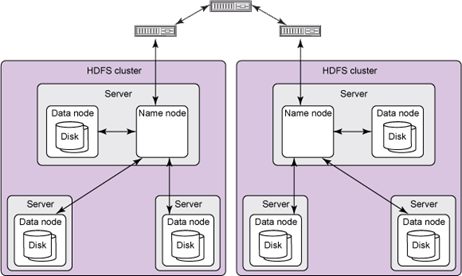


**Figure 3.1** 3Vs of Big Data

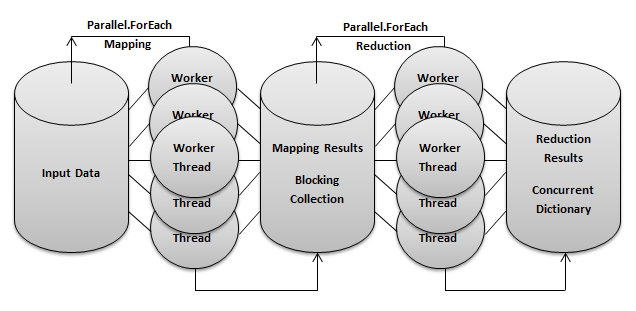
**3.2 Hadoop**

**3.2.1 Hadoop Overview**

Apache Hadoop is open source, and pioneered a fundamentally new way of storing and processing data. Big data is composed of two key components: Distributed File System and MapReduce Engine. Distributed File System provides high boundary storage while MapReduce is data processing framework.



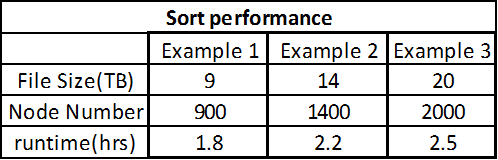
**Figure 3.2.1a** Hadoop Distributed File System (HDFS)



**Figure 3.2.1b** Hadoop MapReduce Engine

**3.2.2 Hadoop Scale**

Hadoop has been demonstrated on clusters of up to 4000 nodes.



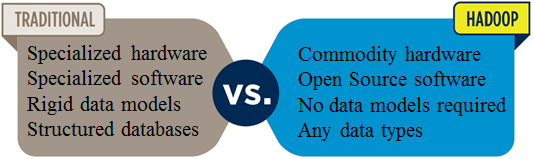
**Figure 3.2.2** Sort performance of Hadoop Examples

**3.2.3 Cost of Distribution**

The HDFS reading performance is much lower than the local File System for the small data set, because each node on the testing cluster has 1 GB Ram and the small data set (512 MB) is fit within the Ram. HDFS is designed for huge data sets, so for the small data set the HDFS writing/reading performance is lower than the local File System. HDFS is a distributed file system above all over local file system on each node because of the HDFS management and maybe Java IO overhead.

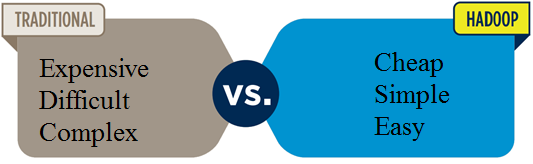
**3.2.5 Hadoop vs. Traditional Database**

Hadoop overcomes the traditional limitations of storage and compute.



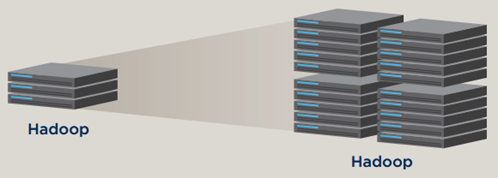
**Figure 3.2.5a**

It leverage inexpensive, commodity hardware as the platform.



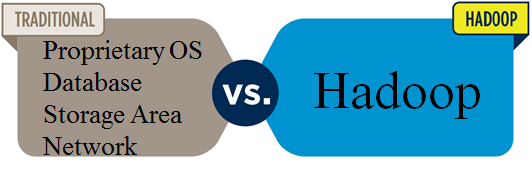
**Figure 3.2.5b**

Hadoop also provides linear scalability 3 from 1 to 4000 servers.



**Figure 3.2.5c**

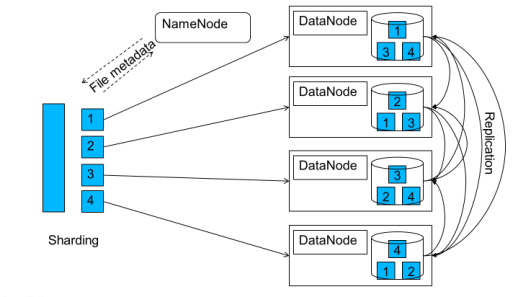
It is low cost, open source software.



**Figure 3.2.5d**

**3.2.5 Fault Tolerance**

If a node fails, the master will detect that failure and re-assign the work to a different node on the system. Restarting a task does not require communication with nodes working on other portions of the data. If a failed node restarts, it is automatically added back to the system and assigned new tasks. If a node appears to be running slowly, the master can redundantly execute another instance of the same task.



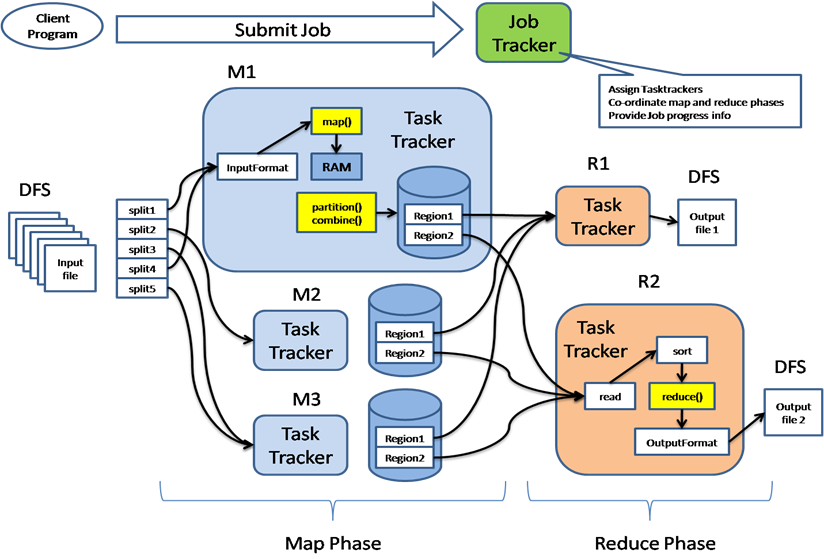
**Figure 3.2.5** Data Replication

**3.2.6 Hadoop Eco Systems**

* *Hadoop Streaming*: A utility to enable MapReduce code in any language: C, Perl, Python, C++, Bash, etc.
* *Hive and Hue*: get 4000 notes and multi-Petabyte scalability. Hue gives a browser-based graphical interface to do your Hive work.
* *Pig*: A higher-level programming environment to do MapReduce coding which provides incredible price-performance and high availability.
* *Sqoop*: Provides bi-directional data transfer between Hadoop and your favorite relational database.
* *Oozie*: Manages Hadoop workflow. It provide if-then-else branching and control within your Hadoop jobs.
* *Hbase*: A super-scalable key-value store. It works very much like a persistent hash-map.
* *FlumeNG*: A real time loader for streaming your data into Hadoop. It stores data in HDFS and HBase.
* *Whirr*: Cloud provisioning for Hadoop. You can start up a cluster in just a few minutes with a very short configuration file.
* *Mahout*: Machine learning for Hadoop. Used for predictive analytics and other advanced analysis.
* *Fuse*: Makes the HDFS system look like a regular file system so you can use *ls, rm, cd,* and others on HDFS data.
* *Zookeeper*: Used to manage synchronization for the cluster.

**3.3 MapReduce**

MapReduce is a programing model for processing large data sets with a parallel, distributed algorithm on a cluster. It was pioneered by Google in 2004. It intended to facilitate and simplify the processing of vast amounts of data in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner.



**Figure 3.3** Architecture of MapReduce

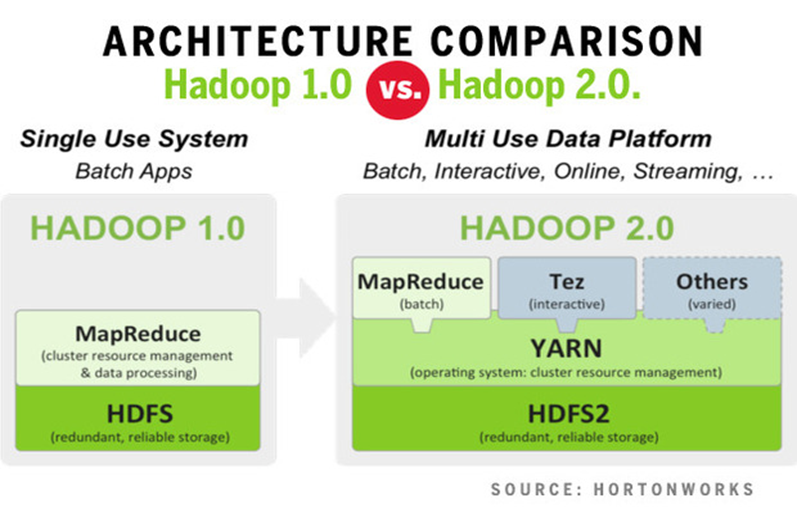
MapReduce is a large scale, open source software framework dedicated to scalable, distributed, data-intensive computing and there are fault tolerant, reliable, and supports thousands of nodes and petabytes of data. MapReduce is not suitable for all problems, but when it works, it may save a lot of application running time.

**3.4 Hadoop 1.0 vs. 2.0**

**3.4.1 Hadoop 1.0 issues and 2.0**

Initially, the Hadoop 1.0 has many issues such as 1) there is only one NameNode, which managed the whole cluster. So there is a scalability limit, 4,000 nodes and 4,000 tasks, 2) it was impossible to update Hadoop components on some of the nodes, 3) the MapReduce paradigm can be applied to only a limited tasks, 4) no other models (other than MapReduce) of data processing, and 5) resources are not utilized in the most efficient way. The Hadoop 2.0 is a breakthrough, when it was published in 2013. They addressed a new concept, YARN (Yet Another Resource Negotiater). YARN extends the number of tasks that can be successfully solved with Hadoop. Hadoop 2.0 eliminate vulnerability of a system with a SingleNode and the possible number of nodes was greatly increased.

**3.4.2 Architecture Comparison**



**Figure 3.4.2** Architecture Comparison

**3.4.3 YARN**

YARN is a Resource Manager that was separating the processing engine and resource management capabilities. It called OS on Hadoop, because it has a function of managing, monitoring workloads, maintaining a multi-tenant environment, implementing security controls, and managing High Availability features of Hadoop. Also it allows multiple, diverse user applications and supports multiple processing models in addition to MapReduce.

**3.4.4 Comparison**

|  |  |
| --- | --- |
| **Hadoop 1.0** | **Hadoop 2.0** |
| Limited up to 4,000 nodes per cluster | Potentially up to 10,000 nodes  per cluster |
| JobTracker bottleneck  – Resource management  - Job scheduling  - Monitoring | Efficient cluster utilization  (YARN) |
| Only has one namespace  for managing HDFS | Support multiple namespace  for managing HDFS |
| Map and Reduce slots are  static |  |
| Only job to run is MapReduce | Any apps can integrate with  Hadoop |
|  | Beyond Java |

**Table 3.4.4** Comparison

1. **Solutions**

4.1 Install Big Data System (Hadoop) in Linux

4.2 Collecting Source Data (3.1GB) at Amazon S3

4.3 Making MapReduce programming (in Java) for analyzing the Big Data

4.4 Execute the MapReduce Program on different nodes (single, 2, 3, 4)

4.5 Making sequential file reading programming (in Java) for comparison

1. **Implementation**

**5.1 Getting Big Dataset**

In our proposal, we intended to get the Big Dataset from Twitter using Twitter API[8]. But we realized the Twitter policy for developer is changed to have a limitation to get a Twitter data. That’s why we were looking for another dataset for our project. Eventually, we found the common web crawling data set supported Amazon Web Services[5]. There are various way to download the data with specific url, but we made a program to get it automatically.

**5.1.1 Data URL and Download Programming**

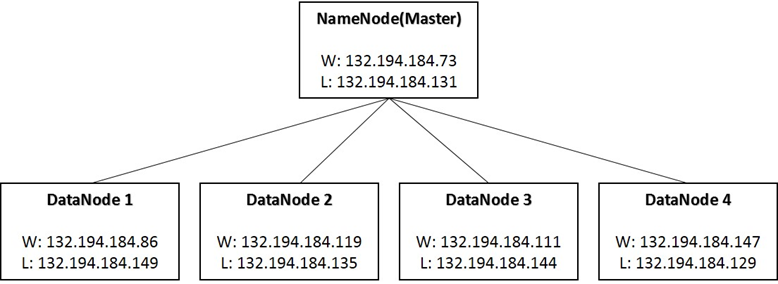
We’ve got 48 address of dataset.

|  |
| --- |
| https://aws-publicdatasets.s3.amazonaws.com/common-crawl/crawl-data/CC-MAIN-2014-35/segments/1408500800168.29/warc/CC-MAIN-20140820021320-00203-ip-10-180-136-8.ec2.internal.warc.gz  https://aws-publicdatasets.s3.amazonaws.com/common-crawl/crawl-data/CC-MAIN-2014-35/segments/1408500800168.29/warc/CC-MAIN-20140820021320-00204-ip-10-180-136-8.ec2.internal.warc.gz  https://aws-publicdatasets.s3.amazonaws.com/common-crawl/crawl-data/CC-MAIN-2014-35/segments/1408500800168.29/warc/CC-MAIN-20140820021320-00205-ip-10-180-136-8.ec2.internal.warc.gz  <https://aws-publicdatasets.s3.amazonaws.com/common-crawl/crawl-data/CC-MAIN-2014-35/segments/1408500800168.29/warc/CC-MAIN-20140820021320-00206-ip-10-180-136-8.ec2.internal.warc.gz> |

With upper address, we downloaded dataset as much as we can get up to 150GB to make a big data. Below psuedo code is getting data with URL information.

|  |
| --- |
| public class URLGetData {  public URLGetData(); // constructor    public void getURLData() { // call two functions  readURLFile();  saveURLData();  }  private void readURLFile(); // read the url from config file  private void saveURLData(); // connect to WEB site, download the data, and save it.  } |

**5.2 Architecture of Hadoop Cluster in our project**



**Figure 5.2** Architecture of Hadoop cluster

**5.2.1 Namenode**

The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself.

**5.2.2 Datanode**

A DataNode stores data in the Hadoop Distributed File System (HDFS). A functional file system has more than one DataNode, with data replicated across them.

**5.3. Installation and configuration**

**5.3.1 Install Hadoop on Hydra**

Hydra is a cluster, which is it has a master node which can manage all other node’s computing availability. But Hadoop is a distributed computing system, so each node can be executed independently. And also, Hadoop system make a Hadoop Distributed File System (HDFS) with each node’s disk storage, but Hydra already share the disk to all nodes with Network File System (NFS). So, we changed a Hydra to 5 machine in lab.

**5.3.2 Pre-requisite for Hadoop in our environment**

Hadoop supports GNU/Linux as a development and production platform and only Win32 as a development platform.

To install the Hadoop, there need some required software.

* Java VM 1.5.x or newer version (We used Java 1.7.0\_71)
* ssh and sshd (Hadoop needs sshd to enable remote access)
* VirtualBox to install Hadoop on Windows 7
* Some port should be accessed (8020, 50070, 10080, etc)

**5.3.3 Installation to 5 machines with VitualBox**

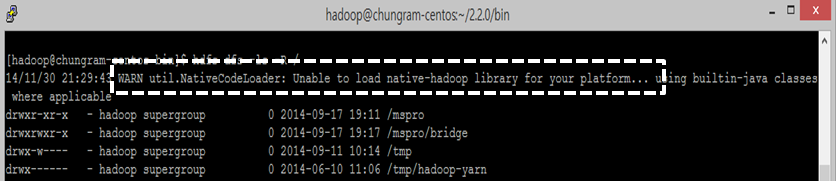
Hadoop Installation is a big challenge in our project. I had to find some solutions on the internet by myself when I met various errors. It’s a kind of potential problem in Open Source.

**5.3.3.1 1st Installation with Archive File**

We’re downloaded the tarball which is the archived version Hadoop 2.5.2 from the hadoop.apache.org.

|  |
| --- |
| [hadoop-m]$wget<http://www.gtlib.gatech.edu/pub/apache/hadoop/common/hadoop-2.5.2/hadoop-2.5.2.tar.gz>  [hadoop-m]$ tar -xvf hadoop-2.5.2.tar.gz  [hadoop-m]$ sudo -u hdfs hadoop fs -ls -R / |

After distract the tarball, we execute the file list command in HDFS. But there is a warning.



That error is the reason why the archived version is compiled on 32 bit Linux system. But our Linux version is Linux 2.6.32 64 bit.

**5.3.3.2 2nd Installation with Hadoop source file**

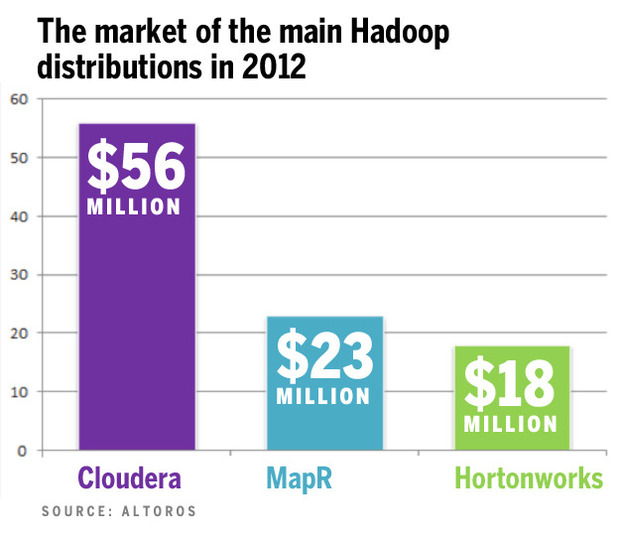
Second, we tried to compile Hadoop 2.5.2 source file. When we compiled the source, many tools need to be installed in the Linux system such as cmake, gcc-c++, ant, maven, protobuf.

When the Hadoop source is compiled, there were still many errors. So, we should find another way to install the Hadoop.

* [ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.16:test (default-test) on project hadoop-common: There are test failures.
* [ERROR] Please refer to /home/hadoop/tools/hadoop-2.5.2-src/hadoop-common-project/hadoop-common/target/surefire-reports for the individual test results.
* [ERROR] Exit code: 1 - /home/hadoop/tools/hadoop-2.5.2-src/hadoop-common-project/hadoop-annotations/src/main/java/org/apache/hadoop/classification/InterfaceStability.java:27: error: unexpected end tag: </ul>

**5.3.3.3 3rd Installation with Hadoop Distributions Company**

When we looked for another way, we found that there is major Hadoop Distributions, Cloudera, MapR, and Hortonworks[9]. A standard open source Hadoop distribution (Apache Hadoop) includes: 1) The Hadoop MapReduce framework for running computations in parallel, 2) The Hadoop Distributed File System (HDFS), and 3) Hadoop common, a set of libraries and utilities used by other Hadoop modules. This is only basic set, but there are other solutions for speed up computations, optimize routine tasks, etc. On the other hand, vendor distributions are designed to overcome issues with the open source edition and provide additional value to customers. So that system give us various features, 1) reliability - the vendors are react faster when bugs are detected, 2) support - the vendors provide technical assistance, and 3) completeness - very often vendors are supplemented with other tools to address specific tasks. We decided to select one of the vendor’s product. Below figure is a comparing the revenue of these major Hadoop vendors in 2012. As the figure, we choose the Cloudera product.



**Figure 5.3** Comparing the Top Hadoop Distributions

**5.3.3.3.1 Install Cloudera product**

To install Cloudera Distribution Hadoop (CDH) 5.2, we just followed the manual supported by Cloudera home page.

Steps to install CDH 5 manually

Step 1: add or build the CDH 5 repository



Step 2: add a repository key

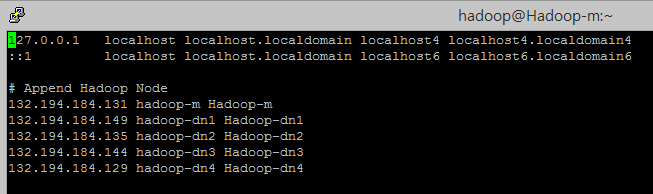


Step 3: Install CDH 5 with YARN

|  |  |
| --- | --- |
| **Where to install** | **Install commands** |
| Resource Manger Host | **sudo yum clean all; sudo yum install hadoop-yarn-resourcemanager** |
| Name Node Host | **sudo yum clean all; sudo yum install hadoop-hdfs-namenode** |
| Secondary Namenode Host | **sudo yum clean all; sudo yum install hadoop-hdfs-secondarynamenode** |
| All cluster hosts except the Resource Manager | **sudo yum clean all; sudo yum install hadoop-yarn-nodemanager hadoop-hdfs-datanode hadoop-mapreduce** |
| One host in cluster | **sudo yum clean all; sudo yum install hadoop-mapreduce-historyserver hadoop-yarn-proxyserver** |
| All client hosts | **sudo yum clean all; sudo yum install hadoop-client** |

Step 4: Deploying CDH 5 on a cluster

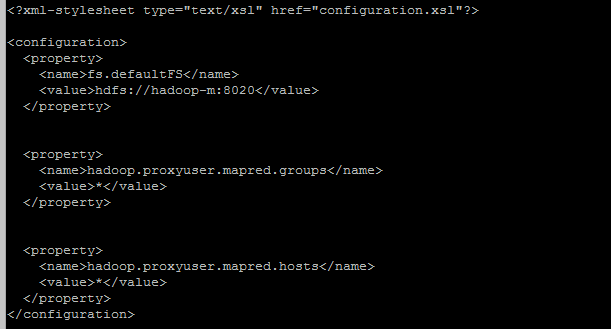
1) Configure Network Names



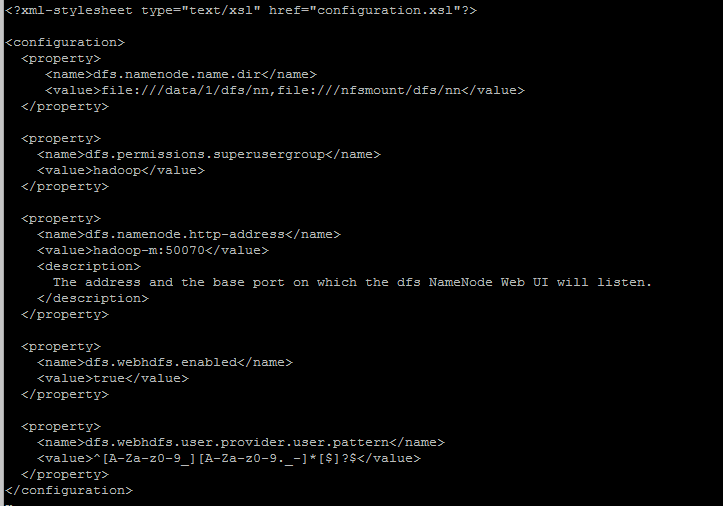
**Figure 5.3.3.3.1a** Hostnames of Namenode

2) Customizing Configuration Files on HDFS

There are two main configuration files: core-site.xml and hdfs-site.xml in */etc/hadoop/conf*.



**Figure 5.3.3.3.1b** core-site.xml



**Figure 5.3.3.3.c** hdfs-site.xml

To configure local storage directories for using by HDFS:

On Namenode hosts:



On all Datanode hosts:



Configure the owner of the HDFS:



After setting the all configuration files, format the HDFS on namenode hosts:



3) Start HDFS, deploy the configuration to all datanodes:

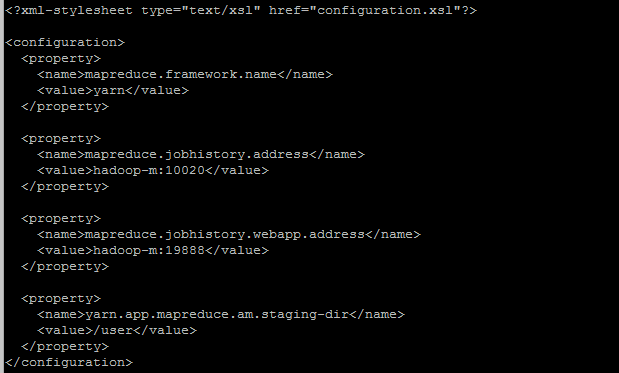


And then, finally start HDFS:

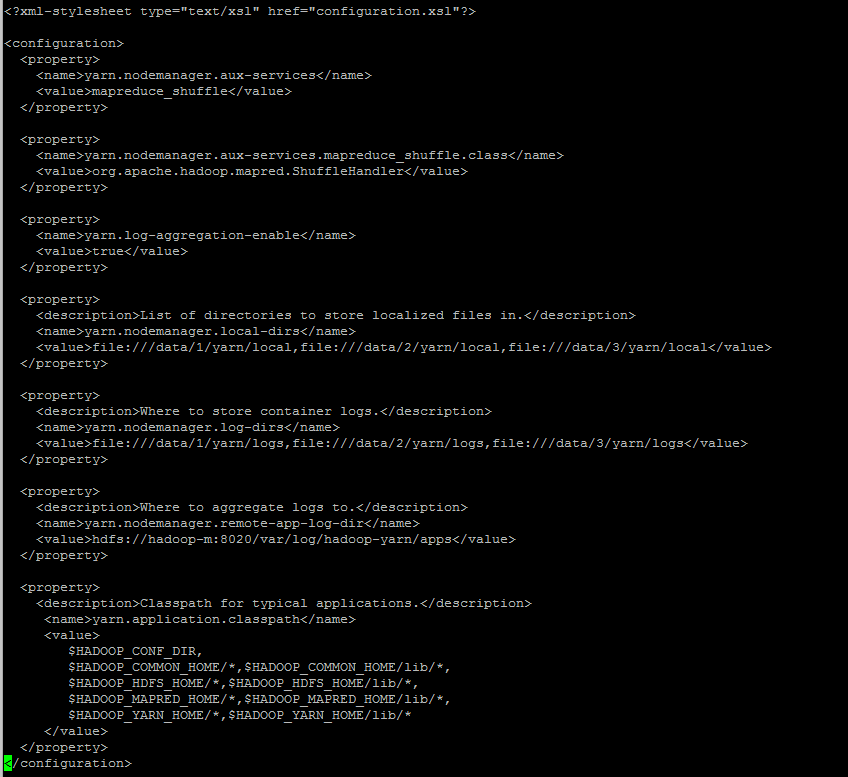


Step 5. Customizing configuration on YARN

1) Configures Properties for YARN clusters: mapred-site.xml and yarn-site.xml



**Figure 5.3.3.3.1d** mapred-site.xml



**Figure 5.3.3.3.1**e yarn-site.xml

2) To configure local storage directories for use by YARN

Create the yarn local and log directories:



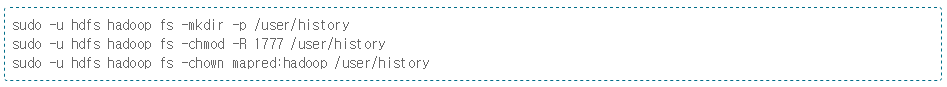


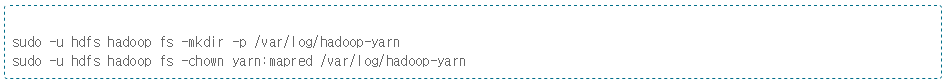
Configure yarn local and log directories:





3) Create the history and log directory and Set permissions and owner

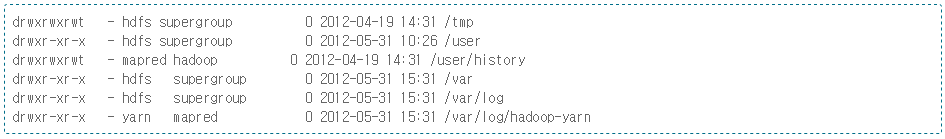




4) Verify the HDFS File Structure



Then we should see,



Step 6: Start YARN and the MapReduce JobHistory Server

On the ResourceManager System:



On each Nodemanager(DataNode):



On MapReduce JobHistory Server:



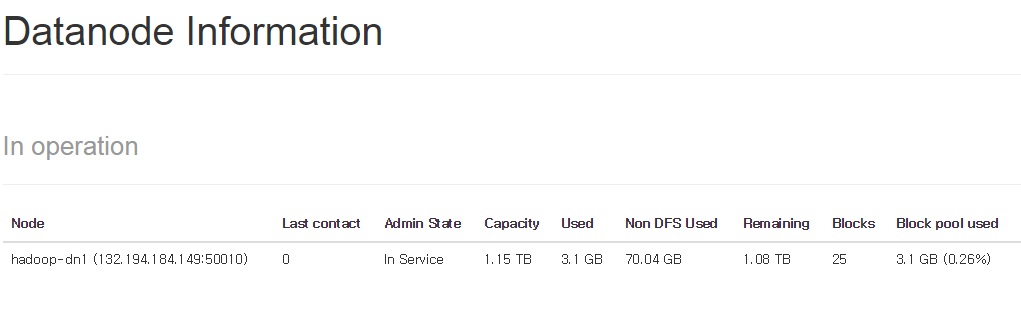
**5.4. Execution Plan**

We tested our MapReduce Program (WordCount) from 1 node up to 4 nodes. So we started from 1 node, 2 node , 3 node, and 4 node in order.

**5.4.1 1 DataNode**

1) Save the data from local to HDFS:

# sudo -u hdfs hadoop fs -put /data/cdata-330m /user/cdata-330m-1node

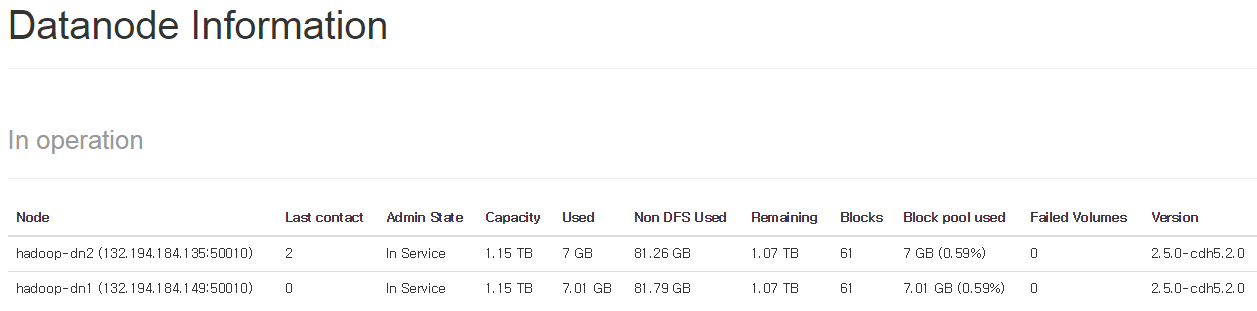


2) Word Cound Execution with Jar: # sudo -u hdfs hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount /user/cdata-330m-1node/cdata-330m/ /user/pa/cdata-330m-1node/wc-out &

**5.4.2 2 DataNode**

1) Save the data from local :

# sudo -u hdfs hadoop fs -put /data/cdata-330m /user/cdata-330m-2node



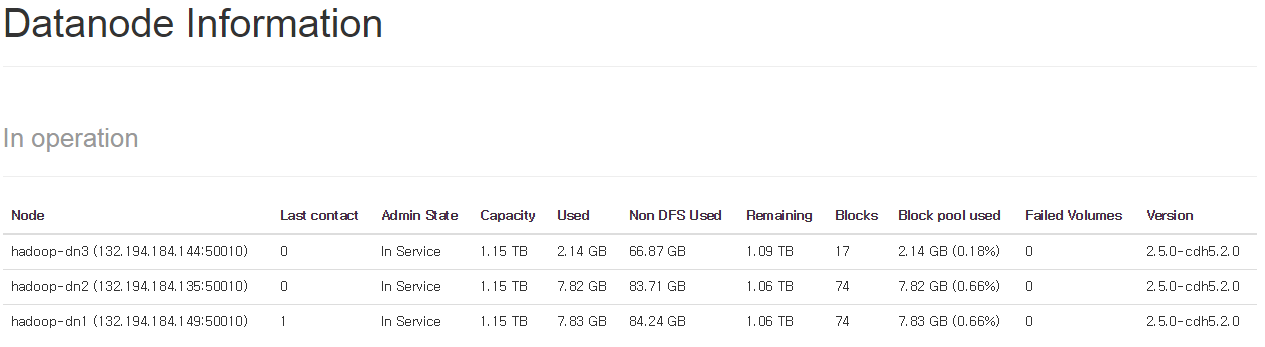
2) word count:

# sudo -u hdfs hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount /user/cdata-330m-2node/cdata-330m/ /user/pa/cdata-330m-2node/wc-out &

**5.4.2.3 3 DataNode**

1) Save the data from local:

# sudo -u hdfs hadoop fs -put /data/cdata-330m /user/cdata-330m-3node



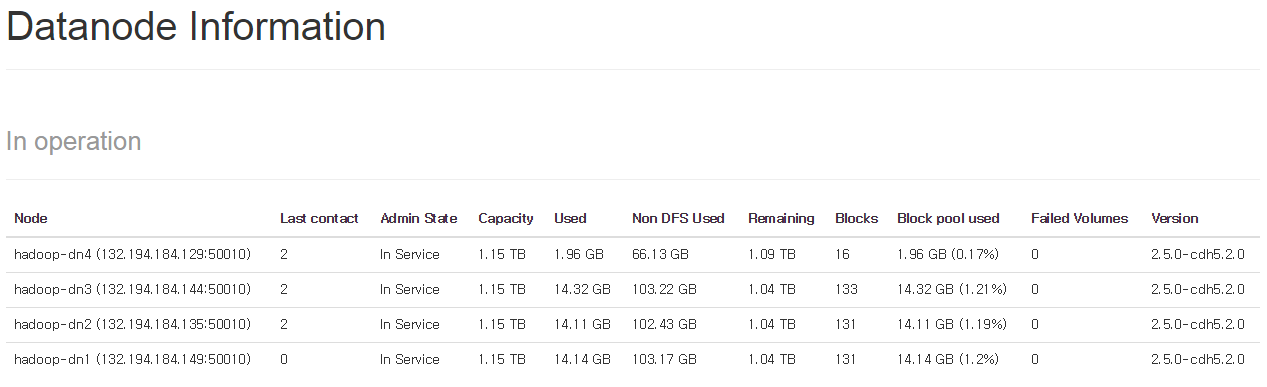
2) Word Count: # sudo -u hdfs hadoop jar

/usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount /user/cdata-330m-3node/ /user/pa/cdata-330m-3node/wc-out &

**5.4.2.4 4 DataNode**

1) Save data to local:

sudo -u hdfs hadoop fs -put /data/cdata-330m /user/cdata-330m-4node



2) Word Count:

# sudo -u hdfs hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount /user/cdata-330m-4node/ /user/pa/cdata-330m-4node/wc-out &

**5.5 Word Counting Programming**

**5.5.1 Hadoop Wordcount Program**

|  |
| --- |
| public class WordCount {  public static class TokenizerMapper  extends Mapper<Object, Text, Text, IntWritable>{  private final static IntWritable one = new IntWritable(1);  private Text word = new Text();  public void map(Object key, Text value, Context context  ) throws IOException, InterruptedException {  StringTokenizer itr = new StringTokenizer(value.toString());  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one);  }  }  }  public static class IntSumReducer  extends Reducer<Text,IntWritable,Text,IntWritable> {  private IntWritable result = new IntWritable();  public void reduce(Text key, Iterable<IntWritable> values,  Context context  ) throws IOException, InterruptedException {  int sum = 0;  for (IntWritable val : values) {  sum += val.get();  }  result.set(sum);  context.write(key, result);  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf, "word count");  job.setJarByClass(WordCount.class);  job.setMapperClass(TokenizerMapper.class);  job.setCombinerClass(IntSumReducer.class);  job.setReducerClass(IntSumReducer.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.addInputPath(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  System.exit(job.waitForCompletion(true) ? 0 : 1);  }  } |

**Figure 5.5.1** Hadoop Wordcount Program

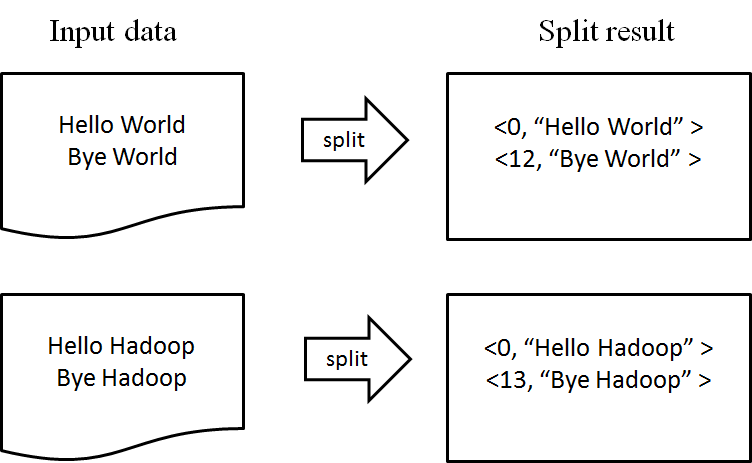
**5.5.2 WordCount Logic View**

The *Map* and *Reduce* functions of *MapReduce* are both defined with respect to data structured in (key, value) pairs. *Map* takes one pair of data with a type in one [data domain](https://en.wikipedia.org/wiki/Data_domain), and returns a list of pairs in a different domain: Map (k1,v1) → list(k2,v2). The *Map* function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key.

The *Reduce* function is then applied in parallel to each group, which in turn produces a collection of values in the same domain: Reduce (k2, list (v2)) → list (v3). Each *Reduce* call typically produces either one value v3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired result list. Thus the MapReduce framework transforms a list of (key, value) pairs into a list of values. This behavior is different from the typical functional programming map and reduce combination, which accepts a list of arbitrary values and returns one single value that combines *all* the values returned by map.

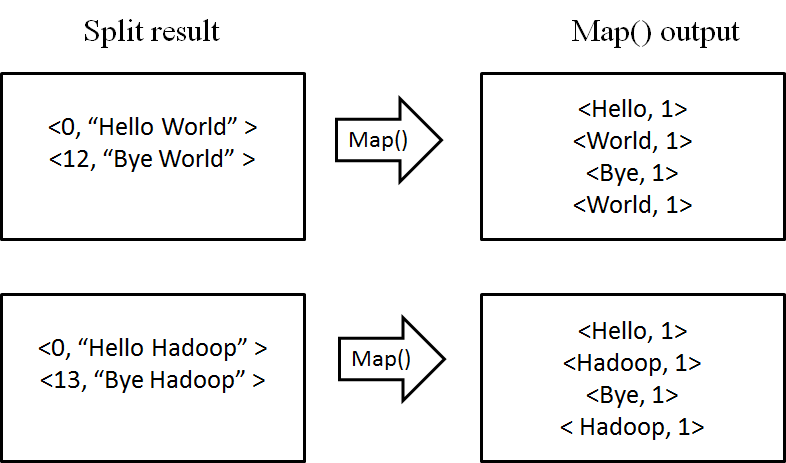
The following example demonstrates how word count program execute a text file in MapReduce. It can be divide by four processes: Split, Map, Sort and combine, Reduce.

5.5.2.1 Process 1: Split



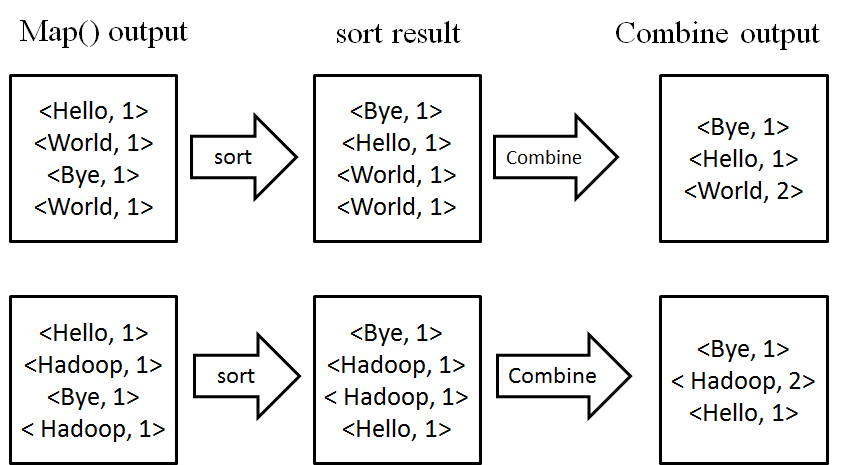
**Figure 5.5.2.1** Split

5.5.2.2 Process 2: Map()



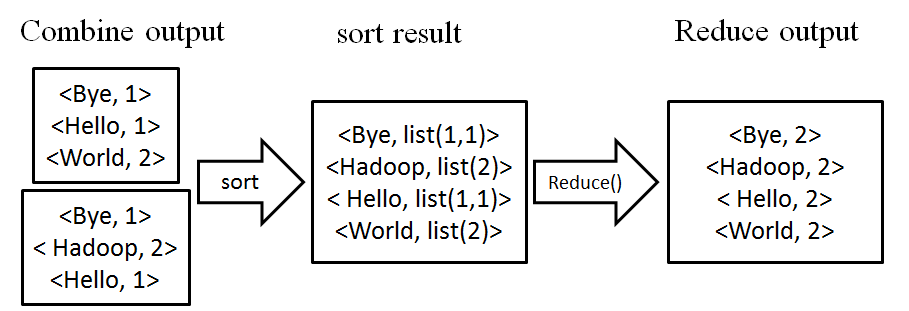
**Figure 5.5.2.2** Map()

5.5.2.3 Process 3: Sort and Combine



**Figure 5.5.2.3** Sort and Combine

**5.5.2.4 Process 4: Reduce()**



**Figure 5.5.2.4** Reduce()

**5.5.3 Sequential Wordcount Program**

|  |
| --- |
| public class ReadFile {  public static void main(String[] args) {  long startTime = System.nanoTime();  long endTime = 0;  TreeMap<String, Integer> frequencyData = new TreeMap<String, Integer>();  readWordFile(frequencyData);  endTime = System.nanoTime();  System.out.println("totalTime = " + (endTime - startTime) / 1000000  + " ms");  }  public static void readWordFile(TreeMap<String, Integer> frequencyData) {  // Scanner wordFile;  Integer count; // The number of occurrences of the word  File folder = new File("TestFolder");  File[] listOfFiles = folder.listFiles();  for (int i = 0; i < listOfFiles.length; i++) {  File file = listOfFiles[i];  if (file.isFile() && file.getName().endsWith(".warc")) {  InputStream in = null;  BufferedReader bfReader = null;  try {  in = new FileInputStream(file);  bfReader = new BufferedReader(new InputStreamReader(in));  char[] buffer = new char[128];  StringBuilder temp = new StringBuilder();  while (bfReader.read(buffer) > 0) {  StringBuilder word = null;  if (temp.length() > 0) {  word = new StringBuilder(temp);  } else {  word = new StringBuilder();  }  boolean isWordBoundary = false;  for (char c : buffer) {  if (c == ' ') {  isWordBoundary = true;  }  if (!isWordBoundary) {  word.append(c);  } else {  temp.append(c);  }  }  count = getCount(word.toString(), frequencyData) + 1;  frequencyData.put(word.toString(), count);  }  } catch (FileNotFoundException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } finally {  if (null != bfReader) {  try {  bfReader.close();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  }  } |

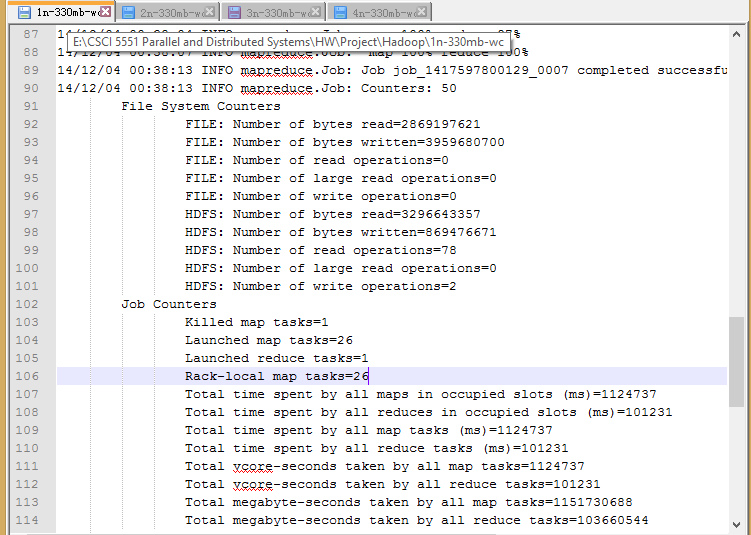
**Figure 5.5.3** Sequential Wordcount Program

1. **Results and Evaluation**

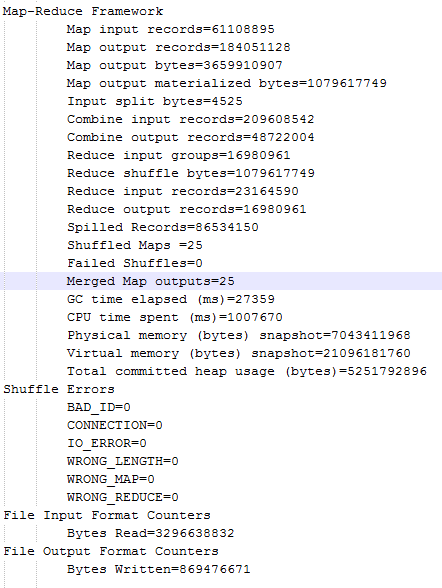
**6.1 WordCount Generated Result**

There are four text files (1n-330mb-wc.txt, 2n-330mb-wc.txt, 3n-330mb-wc.txt, 4n-330mb-wc.txt) which of each is generated by 1 node, 2 nodes, 3 nodes and 4 nodes respectively.

The section of “File system Counters”, “Map-Reduce Framework”, “Shuffle Errors”, “File Input Format Counters” and “File Output Format Counters”. That means all four scenarios have executed successfully. The only different section is “Job Counter” section where we will collect the data.

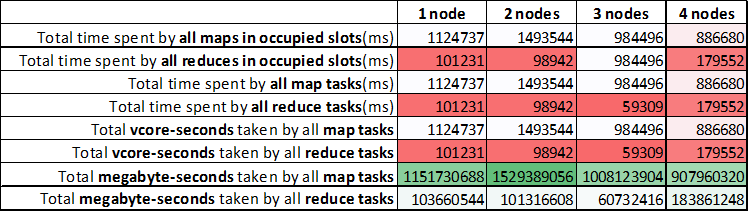


**Figure 6.1a** WordCount Generated Result (Part1)



**Figure 6.1b** WordCount Generated Result (Part2)

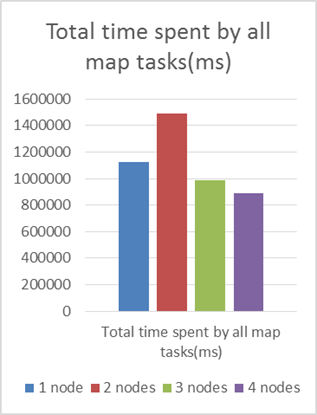
**6.2 JobCounters Data Collection**



**Table 6.2** Job Counters Data Collection



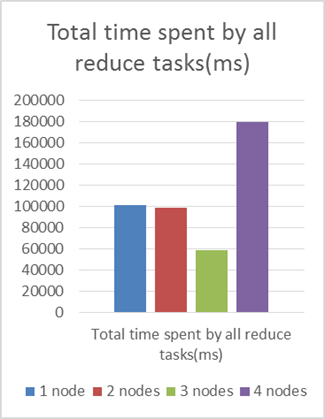
**Table 6.2a** Total time spent by all map task



**Figure 6.2a** Total time spent by all map task

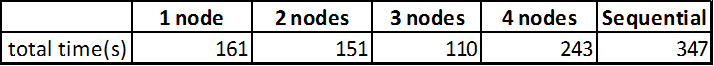


**Table 6.2b** Total time spent by all reduce task

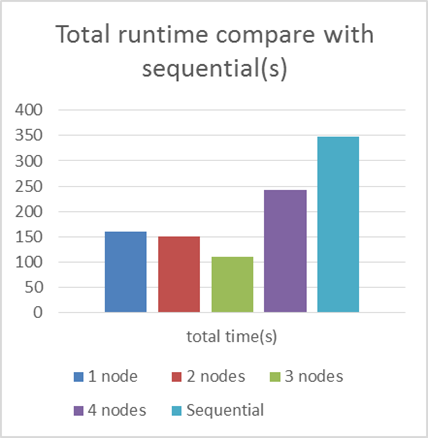


**Figure 6.2b** Total time spent by all reduce task

**6.3 Word Counters Total Runtime**



**Table 6.3** Word Counters Total Runtime



**Figure 6.3** Word Counters Total Runtime

1. **Conclusion**

Hadoop allows writing/reading parallel on all data nodes like other distributed file system.

MapReduce perform parallel operations both in Map() and Reduce() depending on user’s purposes. Compared with sequential wordcount, MapReduce wordcount has better runtime in single node as well as multiple nodes.

We learned a lot through this project, and big data system is a real big system. And while using parallel methods for Big Data, it needs a lot of resources to big data processing. It needs to observe the MapReduce application for tuning, where bottleneck happens. Our future work is tuning the MapReduce applications with deep understanding. And while using a lot of eco-system, we’ll compare which is better, and finally we will compare the performance between MapReduce and No-MR.

**Reference**

[1] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, Bongki Moon, “Parallel Data Processing with MapReduce: A Survey”, *In SIGMOD Record*, December 2011 (Vol. 40, No. 4)

[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, “The Hadoop Distributed File System”, *In Proc, IEEE* 2010,

[3] MapReduce Wikipedia, http://en.wikipedia.org/wiki/MapReduce

[4] J. Dean, S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *In Proc. of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco CA, Dec. 2004.

[5] Amazon Public data sets, common crawl corpus, http://aws.amazon.com/datasets/41740

[6] Apache Hadoop, [http://hadoop.apapche.org](http://hadoop.apapche.org/)

[7] HDFS (Hadoop distributed file system) architecture.

http://hadoop.apache.org/common/docs/current/hdfs design.html, 2009

[8] Twitter API, <https://dev.twitter.com/>

[9] Comparing the top Hadoop Distributions, http://www.networkworld.com/article/2369327/software/comparing-the-top-hadoop-distributions.html