HADOOP 3.X INSTALLATION ON UBUNTU

Running MapReduce Program

Abstract

In this tutorial on Installation of Hadoop 3.x on Ubuntu, we are going to learn steps for setting up a pseudo-distributed, single-node Hadoop 3.x cluster on Ubuntu. Then, we will also learn steps for running the Hadoop MapReduce program (WordCount example).

Dongchul Park

dpark@sm.ac.kr

1. OBJECTIVE

In this tutorial on Installation of Hadoop 3.x on Ubuntu, we are going to learn steps for setting up a pseudo-distributed, single-node Hadoop 3.x cluster on Ubuntu. We will learn steps like how to install java, how to install SSH and configure passwordless SSH, how to download Hadoop, how to setup Hadoop configurations like .bashrc file, hadoop-env.sh, core-site.xml, hdfs-site.xml, mapred-site.xml, yarn-site.xml, how to start the Hadoop cluster and how to stop the Hadoop services. Then, we will address how to run word count program (WordCount.java) on this Hadoop cluster together with prerequisite steps: preparing input data and putting them in HDFS, compiling java codes and creating jar file, running the jar file on the Hadoop cluster, and finally checking the HDFS output file.

2. INSTALLATION OF HADOOP 3.X ON UBUNTU

Throughout this tutorial, we assume we log in the Ubuntu machine with the account name: dpark. That is, its home directory is /home/dpark/ and dpark has a root privilege. So, you can replace dpark account with your own account on your Ubuntu machine. In addition, we recommend Ubuntu LTS (Long Term Support) version: currently the latest LTS version is Ubuntu 18.04 LTS as of 2019.

2.1. JAVA 8 INSTALLATION

Hadoop requires working java installation. Let us start with steps for installing java 8:

~~A. INSTALL PYTHON SOFTWARE PROPERTIES~~

```
~~sudo apt-get install python-software-properties~~
```

~~B. ADD REPOSITORY~~

```
~~sudo add-apt-repository ppa:webupd8team/java~~
```

C. UPDATE THE SOURCE LIST

```
sudo apt-get update
```

D. INSTALL JAVA 8

```
sudo apt-get install openjdk-8-jdk openjdk-8-jre
```

Note: please do not install java version 9 (oracle-java9-installer) at this moment. This causes an error in Yarn as of now (bug!).

E. CHECK IF JAVA IS CORRECTLY INSTALLED

```
java -version
```

2.2. CONFIGURE SSH

SSH is used for remote login. SSH is required in Hadoop to manage its nodes, i.e. remote machines and local machine if you want to use Hadoop on it. Let us now see SSH installation of Hadoop 3.x on Ubuntu:

A. INSTALLATION OF PASSWORDLESS SSH

```
sudo apt-get install openssh-server
```

```
sudo apt-get install pdsh
```

B. GENERATE KEY PAIRS

```
ssh-keygen -t rsa -P '' -f  ~/.ssh/id_rsa
```

C. CONFIGURE PASSWORDLESS SSH

```
cat  ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

E. CHANGE THE PERMISSION OF FILE THAT CONTAINS THE KEY

```
chmod  0600  ~/.ssh/authorized_keys
```

F. CHECK SSH TO THE LOCALHOST

```
ssh localhost
```

Note: now you should be able to log in the localhost machine without typing a password.

2.3. INSTALL HADOOP

A. DOWNLOAD HADOOP

Go to Apache Hadoop homepage (https://hadoop.apache.org) and please head to the "releases" page (http://hadoop.apache.org/releases.html) to download a release of Apache Hadoop. As of October 2019, the latest version of Hadoop is 3.1.3. Download this version. By default, the file will be downloaded in your Downloads directory.

http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-3.1.3/hadoop-3.1.3.tar.gz

B. UNTAR TARBALL

```
tar -xzvf hadoop-3.1.3.tar.gz
```

C. MOVE HADOOP DIRECTORY

```
mv hadoop-3.1.3 /home/dpark/hadoop
```

We assume we want to install hadoop in the following directory: /home/dpark/hadoop.

2.4. HADOOP SETUP CONFIGURATION

A. EDIT .BASHRC

Open .bashrc file with any editor such as *vi, vim, nano, emacs, gedit*, etc. This .bashrc file is located in user's home directory.

```
gedit  ~/.bashrc
```

And edit .bashrc file by adding the following parameters.

```
export HADOOP_HOME="/home/dpark/hadoop"
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_MAPRED_HOME=${HADOOP_HOME}
export HADOOP_COMMON_HOME=${HADOOP_HOME}
export HADOOP_HDFS_HOME=${HADOOP_HOME}
export YARN_HOME=${HADOOP_HOME}
export HADOOP_CLASSPATH=$(hadoop classpath)
```

Note: you have to use your own Hadoop home directory name (red part).

Then save and run this file to take them effect.

```
source ~/.bashrc
```

B. EDIT HADOOP-ENV.SH

Edit configuration file hadoop-env.sh (located in HADOOP_HOME/etc/hadoop) and set JAVA_HOME by adding the following line:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

C. EDIT CORE-SITE.XML

Edit configuration file core-site.xml (located in HADOOP_HOME/etc/hadoop) and add following entries:

```
<configuration>

<property>
<name>fs.defaultFS</name>
<value>hdfs://localhost:9000</value>
</property>

<property>
<name>hadoop.tmp.dir</name>
<value>/home/dpark/hadoop_tmp</value>
</property>

</configuration>
```

Note: **Assuming you already made your own Hadoop working directory (red part) in advance**.

D. EDIT HDFS-SITE.XML

Edit configuration file hdfs-site.xml (located in HADOOP_HOME/etc/hadoop) and add following entries:

```
<configuration>

<property>
<name>dfs.replication</name>
<value>1</value>
</property>

</configuration>
```

E. EDIT MAPRED-SITE.XML

Edit configuration file mapred-site.xml (located in HADOOP_HOME/etc/hadoop) and add following entries:

```
<configuration>

<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>

<property>
<name>yarn.app.mapreduce.am.env</name>
<value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
</property>

<property>
<name>mapreduce.map.env</name>
<value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
</property>

<property>
<name>mapreduce.reduce.env</name>
<value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
</property>

</configuration>
```

Note: If mapred-site.xml file does not exist, then copy and use template file as follows;

cp mapred-site.xml.template mapred-site.xml

F. EDIT YARN-SITE.XML

Edit configuration file yarn-site.xml (located in HADOOP_HOME/etc/hadoop) and add following entries:

```
<configuration>

<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>

<property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>

</configuration>
```

2.5. HOW TO START THE HADOOP SERVICES

Let us now see how to start the Hadoop cluster. The first step to starting up your Hadoop installation is formatting the Hadoop filesystem which is implemented on top of the local filesystem of your "cluster". This is done as follows. First, change to the Hadoop directory you installed (HADOOP_HOME) and follow the following directions.

A. FORMAT THE NAMENODE

Change to the HADOOP_HOME/bin directory and run

```
hdfs namenode -format
```

NOTE: This activity should be done once when you install Hadoop and not for running Hadoop filesystem, else it will delete all your data from HDFS.

B. START HDFS SERVICES

Change to the HADOOP_HOME/sbin directory and run

```
start-dfs.sh
```

Note: if it will give an error at the time of start HDFS services then use:

echo "ssh" | sudo tee /etc/pdsh/rcmd_default

C. START YARN SERVICES

Change to the HADOOP_HOME/sbin directory and run

```
start-yarn.sh
```

Note: You can run both HDFS and Yarn service all at once instead of running both scripts (start-dfs.sh and start-yarn.sh) separately.

```
start-all.sh
```

D. CHECK HOW MANY DAEMONS ARE RUNNING

Let us now see whether expected Hadoop processes are running or not:

```
jps
```

```
2961 ResourceManager
2482 DataNode
3077 NodeManager
2366 NameNode
2686 SecondaryNameNode
3199 Jps
```

Congratulations! your Hadoop installation is completed and you are ready to run MapReduce programs.

2.6. HOW TO STOP THE HADOOP SERVICES

Let us learn how to stop Hadoop services now:

A. STOP YARN SERVICES

Change to the HADOOP_HOME/sbin directory and run

```
stop-yarn.sh
```

B. STOP HDFS SERVICES

Change to the HADOOP_HOME/sbin directory and run

```
stop-dfs.sh
```

Note: You can stop both HDFS and Yarn service all at once instead of running both scripts (stop-dfs.sh and stop-yarn.sh) separately.

```
stop-all.sh
```

Note: Browse the web interface for the NameNode; by default, it is available at:

NameNode → **http://localhost:9870/**

Browse the web interface for the ResourceManager; by default, it is available at:

ResourceManager → **http://localhost:8088/**

3. RUNNING MAPREDUCE PROGRAMS

Now, we are going to run WordCount example on Hadoop MapReduce. WordCount example reads text files and counts how often words occur. The input is text files and the output is text files, each line of which contains a word and the count of how often it occurred, separated by a tab.

3.1. PREREQUISITE

Make sure Hadoop is correctly installed in your system and running:

```
hadoop version
```

```
jps
```

Make sure javac is running correctly:

```
javac -version
```

3.2. RUN WORDCOUNT PROGRAM

Now let's run WordCount program.

A. PREPARE WORDCOUNT PROGRAM

You need to prepare your own WordCount.java program or you may use the code example at the appendix chapter of this document.

Let's make a working directory for this tutorial and assume HADOOP_HOME is /home/dpark/hadoop. Say,

```
mkdir -p /home/dpark/hadoop/WordCountTutorial
```

Then, put the WordCount.java file in this directory.

B. PREPARE INPUT DATA

Create a new folder for input data. You can make your own input data directory at any place. For instance,

```
mkdir -p /home/dpark/hadoop/WordCountTutorial/input_data
```

Change to this input_data directory and add your own text files or create your own text files in this folder. For instance,

```
echo "Hadoop is an elephant." > file0
```

```
echo "Hadoop is as yellow as can be" > file1
```

```
echo "Oh what a yellow fellow is Hadoop" > file2
```

Now we have 3 input files (file0, file1, file2) containing each text statement as above in the input_data directory.

C. CREATE A FOLDER FOR JAVA CLASS FILES

Create a new folder to hold java class files. For instance,

```
mkdir -p /home/dpark/hadoop/WordCountTutorial/wordcount_class
```

D. SET HADOOP_CLASSPATH ENVIRONMENT VARIABLE

You can first check if HADOOP_CLASSPATH environment variable has been already set correctly.

```
echo $HADOOP_CLASSPATH
```

If you cannot see environment variables, you need to set up HADOOP_CLASSPATH environment variable as follows;

```
export HADOOP_CLASSPATH=$(hadoop classpath)
```

Or you can put this line into .bashrc file for your convenience (this file is located in your home directory. For instance, /home/dpark/.bashrc)

E. CREATE A DIRECTORY ON HDFS

Create your own directory on HDFS with the following HDFS command format to put input data we prepared into HDFS.

hadoop fs -mkdir <DIRECTORY_NAME>

For instance,

```
hadoop fs -mkdir -p /user/dpark/wordcount/input
```

F. CHECK HDFS

To check if the directory is created on HDFS correctly, you can browse web interface.

Go to website with the following URL:

http://localhost:9870

This is a default namenode website and you can check namenode status there.

Click *[Utilities] – [Browse the file system]* menu. Then you can find your HDFS directory. You can click the directory name to browse the directories further.

G. UPLOAD THE INPUT FILES TO THE HDFS DIRECTORY

Now you need to upload the input files we prepared before to the HDFS directory with the following HDFS command type:

hadoop fs -put <INPUT_FILES> <HDFS_DIRECTORY>

For instance,

```
hadoop fs -put /home/dpark/hadoop/WordCountTutorial/input_data/*
/user/dpark/wordcount/input
```

You can also check if they all are uploaded to HDFS input directory correctly via namenode web browser we opened before.

H. COMPILE THE WORDCOUNT JAVA CODE

Change back to the tutorial directory containing our WordCount.java and compile it.

```
cd /home/dpark/hadoop/WordCountTutorial
```

Compile the java code with the following format

javac -classpath ${HADOOP_CLASSPATH} -d <CLASS_DIRECTORY> < JAVA_CODE>

For instance,

```
javac -classpath ${HADOOP_CLASSPATH} -d
/home/dpark/hadoop/WordCountTutorial/wordcount_class
/home/dpark/hadoop/WordCountTutorial/WordCount.java
```

Or assuming you are already in WordCountTutorial directory, you can simply use a relative path as follows;

javac -classpath ${HADOOP_CLASSPATH} -d ./wordcount_class WordCount.java

Now you can see class files under /wordcount_class directory.

I. CREATE A JAR FILE

We need to put all the output class files into one jar file with the following command format:

jar -cvf <JAR_FILE_NAME> -C <CLASS_DIRECTORY> .

For example,

```
jar -cvf wordcount.jar -C /home/dpark/hadoop/WordCountTutorial/wordcount_class .
```

Or similarly, you can use a relative path of course if you are already in WordCountTutorial directory.

jar -cvf wordcount.jar -C ./wordcount_class .

Now you should see one jar file named wordcount.jar under your current directory (WordCountTutoria directory).

J. RUN WORDCOUNT PROGRAM ON HADOOP

Now, we are all set and ready to run the jar file on Hadoop with following command type:

hadoop jar <JAR_FILE> <CLASS_NAME> <HDFS_INPUT_DIRECTORY> <HDFS_OUTPUT_DIRECTORY>

As our example,

```
hadoop jar /home/dpark/hadoop/WordCountTutorial/wordcount.jar WordCount
/user/dpark/wordcount/input /user/dpark/wordcount/output
```

Hooray~!! It will work like a charm!!

Note: before you run hadoop program, the HDFS_OUTPUT_DIRECTORY should not exist. Just in case you already ran the same command before, or you designated already existing HDFS directory to the HDFS_OUTPUT_DIRECTORY, it will cause an error saying 'output directory already exists'. In this case, you may delete the existing HDFS directory or you can specify a different new output HDFS directory. If you want to delete HDFS directory, you can use the following command format:

hadoop fs -rm -r <HDFS_DIRECTORY>

e.g,) hadoop fs -rm -r /user/dpark/wordcount/output

K. CHECK THE HDFS OUTPUT FILE

After both map and reduce tasks complete, we can check the output file(s) with the following HDFS command format:

*hadoop fs -cat <HDFS_OUTPUT_DIRECTORY>**

e.g.,) `hadoop fs -cat /user/dpark/wordcount/output/*`

Note: you can also browse the output HDFS files and directory on the namenode web browser.

In addition, if you want to see output HDFS files, you can also type as follows;

```
hadoop fs -ls /user/dpark/wordcount/output/
```

Then, this will display the output HDFS files similar to the following examples.

```
-rw-r--r--   1 dpark supergroup          0  2018-04-10 13:54 output/_SUCCESS
-rw-r--r--   1 dpark supergroup         84 2018-04-10 13:54 output/part-r-00000
```

_SUCCESS file is an empty file indicating RUN was successful. To see output of run, view part-r-00000 file.

```
hdfs fs -cat output/part-r-00000
```

Note: if this Chapter 3 is not clear to you, I recommend you watch the following YouTube video clip for your information. This video tutorial addresses how to run wordcount codes on the Hadoop machine.

https://youtu.be/6sK3LDY7Pp4

APPENDIX

1. WORDCOUNT PROGRAM (SAMPLE CODE)

You may use the following sample codes for your word count program. Simply copy and paste these codes. Then save it named 'WordCount.java' for your convenience.

<WordCount.java>

```
import java.io.IOException;
import java.util.regex.Pattern;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

import org.apache.log4j.Logger;

public class WordCount extends Configured implements Tool {

 private static final Logger LOG = Logger.getLogger(WordCount.class);

 public static void main(String[] args) throws Exception {
  int res = ToolRunner.run(new WordCount(), args);
  System.exit(res);
 }

 public int run(String[] args) throws Exception {
  Job job = Job.getInstance(getConf(), "wordcount");
  job.setJarByClass(this.getClass());
  // Use TextInputFormat, the default unless job.setInputFormatClass is used
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  job.setMapperClass(Map.class);
  job.setReducerClass(Reduce.class);
```

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private long numRecords = 0;
    private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");

    public void map(LongWritable offset, Text lineText, Context context)
        throws IOException, InterruptedException {
      String line = lineText.toString();
      Text currentWord = new Text();
      for (String word : WORD_BOUNDARY.split(line)) {
        if (word.isEmpty()) {
            continue;
        }
            currentWord = new Text(word);
            context.write(currentWord,one);
        }
    }
  }

  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text word, Iterable<IntWritable> counts, Context context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable count : counts) {
        sum += count.get();
      }
      context.write(word, new IntWritable(sum));
    }
  }
}
```

2. SOURCE CODE ANALYSIS

The only standard Java classes you need to import are `IOException` and `regex.Pattern`. You use `regex.Pattern`to extract words from input files.

```
import java.io.IOException;
import java.util.regex.Pattern;
```

This application extends the class `Configured`, and implements the `Tool` utility class. You tell Hadoop what it needs to know to run your program in a configuration object. Then, you use `ToolRunner` to run your MapReduce application.

```
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

The `Logger` class sends debugging messages from inside the mapper and reducer classes. When you run the application, one of the standard `INFO` messages provides a URL you can use to track the job's success. Messages you pass to Logger are displayed in the map or reduce logs for the job on your Hadoop server.

```
import org.apache.log4j.Logger;
```

You need the `Job` class to create, configure, and run an instance of your MapReduce application. You extend the `Mapper` class with your own `Map`class and add your own processing instructions. The same is true for the `Reducer`: you extend it to create and customize your own `Reduce` class.

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
```

Use the `Path` class to access files in HDFS. In your job configuration instructions, you pass required paths using the `FileInputFormat` and `FileOutputFormat` classes.

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

Writable objects have convenience methods for writing, reading, and comparing values during map and reduce processing. You can think of the `Text` class as *StringWritable*, because it performs essentially the same functions as those for integer (`IntWritable`) and long integer (`LongWritable`) objects.

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
```

`WordCount` includes `main` and `run` methods, and the inner classes `Map` and`Reduce`. The class begins by initializing the logger.

```
public class WordCount extends Configured implements Tool {
   private static final Logger LOG = Logger.getLogger(WordCount.class);
```

The `main` method invokes `ToolRunner`, which creates and runs a new instance of `WordCount`, passing the command line arguments. When the application is finished, it returns an integer value for the status, which is passed to the `System` object on exit.

```
  public static void main(String[] args) throws Exception {
     int res = ToolRunner.run(new WordCount(), args);
     System.exit(res);
  }
```

The `run` method configures the job (which includes setting paths passed in at the command line), starts the job, waits for the job to complete, and then returns an integer value as the success flag.

```
public int run(String[] args) throws Exception {
```

Create a new instance of the `Job` object. This example uses the `Configured.getConf()` method to get the configuration object for this instance of `WordCount`, and names the job object *wordcount*.

```
Job job = Job.getInstance(getConf(), "wordcount");
```

Set the JAR to use, based on the class in use.

```
job.setJarByClass(this.getClass());
```

Set the input and output paths for your application. You store your input files in HDFS, and then pass the input and output paths as command-line arguments at runtime.

```
    FileInputFormat.addInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

Set the map class and reduce class for the job. In this case, use the `Map` and `Reduce` inner classes defined in this class.

```
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
```

Use a `Text` object to output the key (in this case, the word being counted) and the value (in this case, the number of times the word appears).

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
```

Launch the job and wait for it to finish. The method syntax is`waitForCompletion(boolean verbose)`. When true, the method reports its progress as the `Map` and `Reduce` classes run. When false, the method reports progress up to, but not including, the `Map` and `Reduce` processes.

In Unix, 0 indicates success, and anything other than 0 indicates a failure. When the job completes successfully, the method returns 0. When it fails, it returns 1.

```
    return job.waitForCompletion(true) ? 0 : 1;
  }
```

The `Map` class (an extension of `Mapper`) transforms key/value input into intermediate key/value pairs to be sent to the `Reducer`. The class defines several global variables, starting with an `IntWritable` for the value 1, and a `Text` object used to store each word as it is parsed from the input string.

```
  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
```

Create a regular expression pattern you can use to parse each line of input text on word boundaries ("\b"). Word boundaries include spaces, tabs, and punctuation.

```
  private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");
```

Hadoop invokes the `map` method once for every key/value pair from your input source. This does not necessarily correspond to the intermediate key/value pairs output to the `reducer`. In this case, the map method receives the offset of the first character in the current line of input as the key, and a `Text` object representing an entire line of text from the input file as the value. It further parses the words on the line to create the intermediate output.

```
  public void map(LongWritable offset, Text lineText, Context context)
        throws IOException, InterruptedException {
```

Convert the `Text` object to a string. Create the `currentWord` variable, which you use to capture individual words from each input string.

```
      String line = lineText.toString();
      Text currentWord  = new Text();
```

Use the regular expression pattern to split the line into individual words based on word boundaries. If the word object is empty (for example, consists of white space), go to the next parsed object. Otherwise, write a key/value pair to the context object for the job.

```
    for ( String word : WORD_BOUNDARY.split(line)) {
        if (word.isEmpty()) {
          continue;
        }
        currentWord = new Text(word);
        context.write(currentWord,one);
      }
  }
```

The mapper creates a key/value pair for each word, composed of the word and the `IntWritable` value *1*. The reducer processes each pair, adding one to the count for the current word in the key/value pair to the overall count of that word from all mappers. It then writes the result for that word to the reducer context object, and moves on to the next. When all of the intermediate key/value pairs are processed, the map/reduce task is complete. The application saves the results to the output location in HDFS.

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>
{
     @Override public void reduce(Text word, Iterable<IntWritable> counts,
Context context)
         throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable count : counts) {
```

```
        sum += count.get();
      }
      context.write(word, new IntWritable(sum));
    }
  }
}
```

Note: This source analysis chapter (Chapter 2) is copied from the following website:

https://www.cloudera.com/documentation/other/tutorial/CDH5/topics/ht_wordcount1_source.html