Hadoop Reference Guide

Contents

Introduction

This reference guide presents the basic instructions and commands to log onto the cloud server, the structure of a Map Reduce program, and instructions on how to run a program on the cloud server. Refer to this guide whenever you need a reminder of commands, syntax, and when trying to find common errors. This reference guide assumes you have basic programming knowledge and will not show you any syntax in regards to a specific programming language unless otherwise stated.

Tools to Use

This section will give a list of recommended tools to use to work with Hadoop. You may use any tools you wish, these are just suggestions.

1. Windows users may want to use PuTTY. This program allows easy SSH into linux based machines to issue command line instructions
2. An Integrated Development Environment (IDE). Eclipse is particularly good for Java and has add-ons for other languages. However any IDE you're comfortable with will work
3. A folder with Hadoop libraries. You can connect your IDE to it and you will eliminate issues with the IDE considering Hadoop libraries as syntax errors.

How to Connect to the Cluster

This section assumes you have been assigned a specific machine number that you will connect to. If you do not have a machine number, see an instructor for assistance as you cannot get access to run your programs without it.

1. Connect to Instance
   - SSH into reu-acad-XX.ecloud.mst.edu as “hadoop”, password: stu-pass
     Note: To SSH -> Windows: use PuTTY, Linux/Mac: Use command line
     Note 2: As before, XX is the machine number you were assigned
     Example: ssh hadoop@reu-acad-01.ecloud.mst.edu

Note: Password is: stu-pass
Note 2: The OS on the system is CentOS. If you are used to Ubuntu, the commands are mostly the same but there are some slight changes. For example, if you want to install vim, you would use: “sudo apt-get install vim” on Ubuntu, but on CentOS you use “yum install vim” If a command is not working, check the equivalent command for CentOS

   - You will be asked a few yes/no questions, enter yes

2. Launching Hadoop
   - Type the following command: start-dfs.sh
   - Type the following command: start-yarn.sh

You should see the following information displayed:
Namenode

Datanode
ResourceManager
NodeManager

- If any of these four items are missing, Hadoop did not launch correctly
- Once you see this, you are logged into the cluster and Hadoop is now running

Note: While you do have root access to your cloud machine, it is not recommended to do anything to the root directory. Damage to cluster files will result in IT having to completely reimage your machine, which will leave you without access for possibly several days. If it happens repeatedly, it will result in permanent loss of access.

Running Your Own Program

Getting Your Files to the Cloud Server

Once logged into your instance and Hadoop is running, you will need to move your program and possibly input files onto the instance. Since we do not have physical access to the machines, we'll use SFTP.

1. Move your files to your Missouri S&T account (FTP, USB, etc...)
2. Log into your cloud instance.

   Note: If you already ran the commands above to create the instance, you do not need to format hdfs, or start dfs and yarn again.
3. SFTP to your university account
   - sftp user@minersftp.mst.edu where user is your username

   Note: Sometimes you will get a message saying it cannot find minersftp.mst.edu. If this happens, change it to: sftp user@131.151.247.27   This is the actual IP address and will resolve this issue
4. Navigate to your files using linux commands
5. Type the following command: get "file" where "file" is the file you want to copy to the instances
6. Once you have all your files, type: "exit"

Getting Files into HDFS

When running Hadoop, your program (EX. JAR file for Java) will stay in your regular folder. Any input files must be in the Hadoop File System (HDFS). To get files into HDFS, do the following:

1. Navigate to your input files
2. Create a directory in HDFS if you have not already: hadoop fs -mkdir /directoryName

   Note: It is recommended that you create a general output directory as well
3. Type the following command: hadoop fs -copyFromLocal fileName /HDFS_Directory
   - Input fileName = File you want to move into HDFS
   - HDFS_Directory = Directory you created in step 2 or existing directory you want to use

Your file is now in HDFS. To check, type: hadoop fs -ls /HDFS_Directory where HDFS_Directory is your created directory in HDFS from step 2.

Note: HDFS directories ALWAYS start with /

EX: Hadoop fs –copyFromLocal input.txt /input

Running a Program with Hadoop

Now that you have your program on the instance and your input files in HDFS you can run your program. To run the program, simply type: hadoop jar jar_file Main_Class Program_Arguments

- This command is specific to Java files. Hadoop is fastest and easiest with Java, however there are libraries for other languages such as Python and C++. For specific commands for other languages, see the documentation for the libraries for these languages
- jar_file = JAR file created in Java with all of your code
- Main_Class = Name of the main class in your program. If you declared the main class in the MANIFEST file, you will leave this out.
- Program_Arguments = List program arguments like you would with any other program run in command line. At minimum, you should have one to tell the program where to dump output. Remember output will be in HDFS and the directory you choose needs to NOT already exist

The program will start and give you status updates. Once complete, you will get a summary of Map Reduce specific information.

Retrieving Results

Once your program is complete, you will have to get your output from HDFS. Do not leave your output in HDFS. If for any reason the machine your instance is on restarts or shuts down, you will lose all of your data.

1. Extract output from HDFS to the instance using: hadoop fs -copyToLocal HDFS_Directory/file localDirectory
    - HDFS_Directory = the directory path to your output. You can specify the file you want as shown above or you can use * to indicate you want all files. EX: Hadoop fs -copyToLocal /output/test1/* localDirectory
    - localDirectory = the directory on the instance you want the results moved to. If you are already in the directory you want, you can leave this out and it will default to your current directory. It should NOT be preceded by a /
2. Using SFTP as above, move your files to your Missouri S&T account using the “put” command instead of “get”

Map Reduce

This section will provide the basic structure of the various parts of a Map Reduce program. While the code and function of the program is entirely up to you, this section will give you a guideline of things that must be included for Map Reduce to work properly. Code examples are given in Java. See Hadoop library documentation for other languages.

Mapper

Example Mapper Function:

```
public static class MyMapper extends Mapper<k1, v1, k2, v2>{
   public void map(key, value, context){
     Your code here
     context.write(k2, v2);
   }
}
```

In the above piece of code, we start by declaring our own mapper class call MyMapper which extends the Hadoop Mapper class. The types k1 and v1 declare what types the mapper is expecting for input for the key and value. The input key value type is always LongWritable for Mappers. The types k2 and v2 are the output key and value types. The input value, and the output key and value types can be anything the developer desires. At the end, the mapper outputs a key and a value of the type declared at the top, which is sent to the reducers.

Combiner

Many Map Reduce programs can be made more efficient with the use of a combiner function. A combiner function groups values coming from a mapper. For example, if we have a program counting the number of times each word appears in a book, then for each mapper, the combiner would count the number of times each word appeared in that mapper. There are some important points to remember about combiners:

1. There is one combiner for each mapper. Meaning that the combiner does not aggregate all values, just the ones from an individual mapper
2. Map Reduce makes the decision if the combiner is used based on input size and available resources. It is not guaranteed the combiner will run. Therefore:
   - Combiner output types must be identical to reducer input types
   - The reducer must be written such that even if the combiner never ran, the program would still function properly

Example Combiner Function

```
Public static class MyCombiner extends Reducer<k1, v1, k2, v2>{
   public void reduce(key, values, context){
      your code here
      context.write(key, value);
   }
}
```

Notice that the combiner function uses the Map Reduce “Reduce” function. It functions in much the same way and reducers can be used as combiners in some applications.

Partitioner

By default, Map Reduce uses a hash function to make sure output from the mappers is split evenly among the reducers. However, if you have a specific way you wish the data to be partitioned, you may write your own partitioning function.

Example Partitioner Function

```
Public static class MyPartitioner extends Partitioner<key, value>{
   public int getPartition(key, value, int){
      your code here
      return int;
   }
}
```

The partitioner takes the key and value input, their types denoted in the class declaration, runs your code over them, and outputs a number denoting which reducer the key, value pair will be sent to. The integer in the function call refers to the number of reducers you are using. Reducers start at 0, therefore if you declare 3 reducers, you will be able to assign data to reducers 0, 1, and 2.

Reducer

The reducer does any final aggregation and final calculations, knowledge discovery, and/or completes whatever it is your program is supposed to do. The output from the reducer goes to one or more files, each file from a single reducer.

Example Reducer Function

```
Public static class MyReducer extends Reducer<k1, v1, k2, v2>{
   public void reduce(key, values, context){
      your code here
      context.write(key, value);
   }
}
```

Main Function

The main function of the program will set up the mappers, reducers, output files, and other information needed for Map Reduce to do a job.

Example Main Function

```
Public static void main(String args[]){
   Configuration conf = new Configuration();
   conf.set("constant", args[2]);
   Job job = new Job(conf, "MyJob");
   job.setJarByClass(Main.class);
   job.setMapperClass(MyMapper.class);
   job.setCombinerClass(MyCombiner.class);
   job.setPartitionerClass(MyPartitioner);
   job.setReducerClass(MyReducer);
   job.setNumReduceTasks(3);
   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(Text.class);
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(Text.class);
   FileInputFormat.addInputPath(job, new
Path(args[0]));
   FileOutputFormat.setOutputPath(job, new
Path(args[1]));
}
```

In the above main function, we first declare a new configuration. This configures Map Reduce for specifics to your program. Here is where you would put constants like other files it may need to look at. In the above example, we used an argument passing into the program. It then declares a new Job using the configurations you just created. We then declare the main class for the job to start working from. Afterward, it declares the mapper, combiner, partitioner, and reducer classes. It is possible to have multiple functions of each type, so Map Reduce requires that each job have the functions explicitly declared for it. We then declare the number of reducers we want to use. In this example, we will have 3 reducers. We then declare what the output key and value types are from the mapper and from the reducers. In this example, all types were declared as Text. More information about Hadoop types will be given in the next section. Finally, we declare the input and output directories. In this example, they are given as program arguments.

Hadoop Types

In order to maintain efficiency, Hadoop uses its own variable types, however they are convertible to primitive types for whichever language you use. For example, a Text is the same as a String. Therefore, we can convert them back and forth as needed:

```
String MyString = MyText.toString();
Text MyText = new Text(MyString);
```

Custom types are possible, but significantly reduce efficiency. It's recommended to stay with the built in types. Below is a table of Hadoop types and their primitive twins.

Primitive Type	Hadoop Type
Int	IntWritable
Long	LongWritable
Boolean	BooleanWritable
Float	FloatWritable
Byte	ByteWritable
Array	ArrayWritable*

* This can only be used as a type for values. It cannot be used as a key type

You can use whatever types you want in your own code, including Hadoop and the built in types for the language you are using. However, all input into mapper, combiner, partitioner, and reducer functions and output from these functions must be in one of the Hadoop built in types.

Linux Commands Reference

This section is a quick reference on common Linux commands you are likely to use. Please see the Useful Links section for a link to more commands.

Command	Example	Description
cat	cat example.txt	Sends the contents of the file to the screen to view. Can also be sent to a file.
cd	cd .. cd /home cd myDirectory	changes your current directory to the specified one. cd.. sends you to the parent directory of your current one
cp	cp myFile yourFile	Make a copy of a file. myFile is the original. yourFile is the name of the copy
ls	ls ls -l ls directory	lists all files and directories in the current directory. You can also see everything in a subdirectory by adding the subdirectory name at the end. -l includes the size of the files
mv	mv file dir/file	Moves a file to a specific location
rm	rm file rm -r directory	Delete a file or directory. -r forces a delete of a directory and all of its contents
chmod	chmod 777 file	Changes permissions on a file to a specified setting
diff	diff file1 file2	Compares two files and shows where they differ
man	man command	Displays the user manual on a specific linux command

Hadoop Commands Reference

Many of the commands available to Hadoop are slightly modified Linux commands. Below is a table of common Hadoop commands you will use. Additional commands will be found in the useful links section.

Command	Example	Description
copyFromLocal	hadoop fs -copyFromLocal localSrc Dest	Copies a file from a local folder to HDFS
copyToLocal	hadoop fs -copyToLocal HDFSSrc Dest	Copy a file from HDFS to local directory. Local destination argument is optional
Most linux commands	hadoop fs -ls /dir hadoop fs -mkdir /dir hadoop fs -rm /dir/file	Most basic linux commands can be used in HDFS in this manner
jar	hadoop jar myProgram.jar MainClass Args	Runs a Java JAR file with Map Reduce

Useful Links

Below is a list of other useful links that may help when writing Map Reduce programs.

Linux Commands
http://www.linuxdevcenter.com/cmd/

SFTP Commands
https://www.digitalocean.com/community/tutorials/how-to-use-sftp-to-securely-transfer-files-with-a-remote-server

Hadoop Shell Commands
http://hadoop.apache.org/docs/r0.18.3/hdfs_shell.html

Hadoop Tutorial
http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html