**Aims**

This exercise aims to get you to:

* Compile, run, and debug MapReduce tasks via Command Line
* Compile, run, and debug MapReduce tasks via Eclipse
* Apply the design pattern “in-mapper combining” you have learned in Chapter 2.2 to MapReduce programming

NOTE: PLEASE USE YOUR OWN PATHS IF NOT USING VIRTUAL MACHINE

## One Tip on Hadoop File System Shell

## Following are the three commands which appear same but have minute differences:

**1. hadoop fs {args}**

**2. hadoop dfs {args}**

**3. hdfs dfs {args}**

## The first command: fs relates to a generic file system which can point to any file systems like local, HDFS etc. So this can be used when you are dealing with different file systems such as Local FS, HFTP FS, S3 FS, and others.

## The second command: dfs is very specific to HDFS. It would work for operation relates to HDFS. This has been deprecated and we should use hdfs dfs instead.

## The third command: It is the same as the 2nd. It would work for all the operations related to HDFS and is the recommended command instead of hadoop dfs.

## Thus, in our labs, it is always recommended to use hdfs dfs {args}.

## Compile and Run “WordCount” via Command Line

This exercise aims to make you know how to compile your MapReduce java program and how to run it in Hadoop.

1. Download the sample code “WordCount.java”:

**$ wget http://www.cse.unsw.edu.au/~z3515164/WordCount.java**

2. Add the following environment variables to the end of file ~/.bashrc:

|  |
| --- |
| **export HADOOP\_CLASSPATH=${JAVA\_HOME}/lib/tools.jar** |

Save the file, and then run the following command to take these configurations into effect:

**$ source ~/.bashrc**

3. Compile WordCount.java and create a jar:

**$ hadoop com.sun.tools.javac.Main WordCount.java**

**$ jar cf wc.jar WordCount\*.class**

4. Generate two files, file1 and file2 in folder TestFiles at your home folder:

**$ mkdir ~/TestFiles**

**$ echo Hello World Bye World > ~/TestFiles/file1**

**$ echo Hello Hadoop Goodbye Hadoop > ~/TestFiles/file2**

5. Start HDFS and put the two files to HDFS:

**$ start-dfs.sh**

**$ hdfs dfs -mkdir input**

**$ hdfs dfs -put ~/TestFiles/\* input**

6. Run the application:

**$ hadoop jar wc.jar WordCount input output**

7. Check out the output:

**$ hdfs dfs -cat output/\***

If you are using Mac OS, please add ‘\’ before the ‘\*’ if the file/folder exists but you still see some errors. For example, in step 7 (check out the output)

**$ hdfs dfs -cat output/\\***

## Create a WordCount Project in Eclipse

0. Download Eclipse and the Hadoop plugin

Download the Eclipse IDE from <https://www.eclipse.org/downloads/packages/> corresponding to your working environment:

<https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2022-03/R/eclipse-java-2022-03-R-linux-gtk-x86_64.tar.gz> for the VM;

<https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2022-03/R/eclipse-java-2022-03-R-macosx-cocoa-x86_64.dmg> for Mac with Intel chips;

<https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2022-03/R/eclipse-java-2022-03-R-macosx-cocoa-aarch64.dmg> for Mac with M1 chips.

Download the Hadoop Eclipse plugin from **http://www.cse.unsw.edu.au/~z3515164/hadoop-eclipse-plugin-3.3.2.jar**

Next, move the jar file to the plugin folder of Eclipse, e.g.:

**$ mv hadoop-eclipse-plugin-3.3.2.jar ~/eclipse/plugins**

For Mac OS, you could open Finder -> Application, find Eclipse. Then you can right click on it, choose Show Package Contents and go to Contents -> Eclipse ->plugins. Finally copy the jar file into this folder.





The plugin makes it simple to create a new Hadoop project, to view files in HDFS, and to execute Hadoop jobs. In this exercise, you will learn how to use Eclipse to create a MapReduce project, configure the project, and run the program. You can also manage the files in HDFS by using Eclipse, instead of using commands to transfer files between local file systems and HDFS.

1. Configure the eclipse Hadoop plugin:

a) Open Eclipse and make the workspace folder at “/home/comp9313/workspace” by default. If you installed Hadoop locally, please create your workspace in one proper path.

b) In Eclipse Menu, select Window->Preferences, then a dialog will pop up like below:



Configure your Hadoop installation directory as shown in the figure.

Here, please also check the java compiler version: click Java->Compiler, and make sure that the compiler compliance level is 11.



c) Change to the Map/Reduce Perspective:

Select Window->Open Perspective->Other->Map/Reduce



d) Connect Eclipse with HDFS

Right click in tab Map/Reduce Locations, and select “New Hadoop location”



In the pop-up dialog, give a name for the Hadoop location, and change the port of DFS Master to “9000”



Next, click the “Advanced parameters” tab, and set “fs.defaultFS” to “hdfs://localhost:9000”, and set “hadoop.tmp.dir” to the “tmp” folder under the $HADOOP\_HOME.

2. Create your WordCount Project in Eclipse

a) In the left “Project Explorer” panel, click Create a project, and in the pop-up dialog select the Map/Reduce Project wizard.



Then, name the project as “WordCount”, and you can see it in “Project Explorer”.

b) Test the connection. If you have successfully connected Eclipse and Hadoop, you can now see the folders and files in HDFS under “DFS Locations”. You can click the files to view them, and you can also download files to local file system or upload files to HDFS.



c) Create a new class “WordCount”, in package “comp9313.lab2”



d) Replace the code of class WordCount by the content of “WordCount.java” in the first exercise.



e) Copy the file “log4j.properties” from $HADOOP\_CONF\_DIR to the src folder of project “WordCount”

**$ cp $HADOOP\_CONF\_DIR/log4j.properties ~/{your workspace}/WordCount/src**

Then right click the project in Eclipse and click “Refresh”.

This step is to configure the log4j system for Hadoop. Without doing this, you cannot see the Hadoop running message in Eclipse console.

## Run MapReduce Jobs in Eclipse

## Right click the new created file WordCount.java, and select Run as->Run Configurations->Java Application. After double clicking, in the dialog, click the tab “Arguments”, and configure the arguments for this project as: hdfs://localhost:9000/user/comp9313/input hdfs://localhost:9000/user/comp9313/output (Please use your own username if Hadoop is installed locally). Finally, click “Run”.

##

|  |
| --- |
| **Warning:** Note that if output already exists, you will meet an exception. Remember to delete **output** on HDFS before you run a Hadoop job:**$ $HADOOP\_HOME/bin/hdfs dfs –rm –r output** |

## If everything works normally, you will see the Hadoop running message in the Eclipse console.

|  |
| --- |
| **Note:** If you still see the following warnings after you run the program, you may need to restart eclipse. |

## Refresh (or maybe Reconnect) the location “hadoop”, you will see that a new folder “output” is listed, and you can click the file in the folder to see the results.

##

|  |
| --- |
| Quiz: Split the code into three files: one for mapper, one for reducer, and one for main (driver), and run the project again. Normally, in a MapReduce project, we will put the three classes into different files.Note that the mapper and reducer classes are not static in this case! |

## After you have set up the run configuration the first time, you can skip the step of configuring the arguments in subsequent runs, unless you need to change the arguments.

## Now you’ve made the MapReduce job run in Eclipse. Note that we did not configure Eclipse to use YARN for managing resources.

## Package MapReduce Jobs using Eclipse

## Once you've created your project and written the source code, to run the project in pseudo-distributed mode and let YARN manage resources, we need to export the project as a jar in Eclipse:

## NOTE: You need to use YARN in the following task. Please check whether you have mapred-site.xml in $HADOOP\_CONF\_DIR folder. If you renamed it (add .bak) at Lab 1, please rename it. After that, use $start-yarn.sh.

## 1. Right-click on the project and select Export.

## 2. In the pop-up dialog, expand the Java node and select JAR file. Click Next.

##

## 3. Enter a path in the JAR file field (e.g., /home/comp9313/wc.jar), and click Finish.

## 4. Open a terminal and run the following command:

**$ hadoop jar ~/wc.jar comp9313.lab2.WordCount hdfs://localhost:9000/user/comp9313/input hdfs://localhost:9000/user/comp9313/output**

## Remember to delete the output folder in HDFS first!

## You can also simply run the following command:

**$ hadoop jar ~/wc.jar comp9313.lab2.WordCount input output**

## By using the “hadoop” command, I/O is based on the distributed file system by default, and /user/comp9313 is the default working folder.

## Debugging Hadoop Jobs

## To debug an issue with a job, the easiest approach is to run the job in Eclipse and use a debugger. To debug your job, do the following step.

## 1. Set a watch point in TokenizerMapper in the while loop:

|  |
| --- |
| **while (itr.hasMoreTokens()) {** **word.set(itr.nextToken());** **context.write(word, one);** **System.out.println(word.toString());****}** |

## Double click the line number of the red line in Eclipse to set the watch point.

## 2. Right-click on the project and select Debug As -> Java Application, and open the debug perspective.

## 3. The program will run, and stop at the watch point:

##

## Now you can use the Eclipse debugging features to debug your job execution.

## 4. Logs are also very useful for you to debug your MapReduce program.

## You can either print the debug information in stdout or write the debug information in the Hadoop system log.

## Import the relevant log classes in the java file:

|  |
| --- |
| **import org.apache.commons.logging.Log;****import org.apache.commons.logging.LogFactory;** |

## In TokenizerMapper, add the following two lines after “System.out.println(word.toString());”:

|  |
| --- |
| **Log log = LogFactory.getLog(TokenizerMapper.class);****log.info("MyLog@Mapper: " + word.toString());** |

## In the reducer class IntSumReducer, add the following lines at the end of the reduce function:

|  |
| --- |
| **System.out.println(key.toString()+ " " + result.toString());****Log log = LogFactory.getLog(IntSumReducer.class);****log.info("MyLog@Reducer: " + key.toString() + " " + result.toString());** |

## Export the project as a jar file and run it in the terminal again.

## You will find your log messages in logs through different ways:

## a) Through <http://localhost:9870>

## Select Utilities->Logs, then click “userlogs/”, the log folder of your recent job is shown at the bottom. Go into the folder, and you will see another four log folders.

## Text  Description automatically generated with medium confidence

## Each map and reduce will record their own log. Enter the folder ending with “000002”, and then click syslog, you can find:

##

## If you click stdout, you can find:

## Graphical user interface, text, application, chat or text message  Description automatically generated

## As you can see, System.out.println() prints the information to stdout, while, the Log class writes the information to syslog.

## Enter the folder ending with “000003”, and then click syslog, you can find:

## Text  Description automatically generated

## Enter the folder ending with “000004”, and then click syslog, you can find:

##

## If you click stdout, you will see:

## Text, chat or text message  Description automatically generated

## b) Through your local machine.

## Open terminal, cd to the Hadoop log folder to check the logs for your job:

## $ cd $HADOOP\_HOME/logs/userlogs

## For large MapReduce project, using logs is the best way to debug your code.

## Try to Write Your First Hadoop Job

## 1. Download the test file, and put it to HDFS:

**$ wget** [**http://www.cse.unsw.edu.au/~z3515164/pg100.txt**](http://www.cse.unsw.edu.au/~z3515164/pg100.txt)

**$ hdfs dfs –rm input/\***

**$ hdfs dfs –put ~/pg100.txt input**

## 2. Now please write your first MapReduce job to accomplish the following task:

## Output the number of words that start with each letter. This means that for every letter we want to count the total number of words that start with that letter. In your implementation, please first all words to lower case. You can ignore all non-alphabetic characters. Create a class “LetterCount.java” in package “comp9313.lab2” to finish this task.

## Hint: In the (key, value) output, each letter is the key, and its count is the value.

## How to write a mapper properly?

## How to write a combiner? Is it necessary?

## How to write a reducer properly?

## Compare your results with the answer provided at:

<https://webcms3.cse.unsw.edu.au/COMP9313/22T3/resources/81107>

## Improve WordCount by In-Mapper Combining

Use the following codes to tokenize a line of document:

**StringTokenizer itr = new StringTokenizer(value.toString(),
 " \*$&#/\t\n\f\"'\\,.:;?![](){}<>~-\_");**

Documents will be split according to all characters specified (i.e., **" \*$&#/\t \n\f"'\,.:;?![](){}<>~-\_"**), and higher quality terms will be generated.

Convert all terms to lower case as well (by using toLowerCase() method).

Apply this to the mapper class of WordCount we have used.

1. Put the input file to HDFS by:

**$ hdfs dfs –rm input/\***

**$ hdfs dfs –put ~/pg100.txt input**

1. Use the new method to tokenize each line of document
2. Run the code in Eclipse: Right click the class->Run As->Run Configuration->Right click Java Application and select New. Then specify the arguments: “hdfs://localhost:9000/user/comp9313/input hdfs://localhost:9000/user/comp9313/output”
3. Remember to delete to output folder if it exists

Type the following command in the terminal:

**$ hdfs dfs -cat output/\* | head**

You should see results:



Use the following command:

**$ hdfs dfs -cat output/\* | tail**

You should see:



Based on the WordCount.java we used in Lab2, you are required to write an improved version using “in-mapper combining”. The input file is still the one we used in Lab 2.

Create a new class “WordCount2.java” in the package “comp9313.lab3” and solve this problem. Your results should be the same as generated by WordCount.java.

Hints:

1. Refer to the pseudo-code shown in slide 23 of Chapter 2.2.
2. You need to use a HashMap in the mapper class to buffer the partial results for different calls of the map() function. You can define an object of type HashMap<String, Integer> for this problem.

If you are not familiar with HashMap, the following pages may be helpful:

<https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html>

<http://www.java67.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html>

<http://www.java2novice.com/java-collections-and-util/hashmap/>

For example, in order to iterate the key-value pairs in a HashMap, you can do like below:

|  |
| --- |
| **Set<Entry<String, Integer> > sets = map.entrySet();****for(Entry<String, Integer> entry: sets){****//entry.getKey() to get the key****//entry.getValue() to get the value****}** |

1. The results are not emit in map() now. Rather, you will need to override the cleanup() function in the mapper class to generate the key-value pairs. The cleanup() function is defined like:

|  |
| --- |
| **public void cleanup(Context context) throws IOException, InterruptedException {** **//generate the output of the mapper** **… …****}** |

This function will be called once at the end of the map task. You do not need to call this function, and Hadoop will do it (just like the function map()). More usage of cleanup() please refer to: <https://hadoop.apache.org/docs/r3.3.2/api/org/apache/hadoop/mapreduce/Mapper.html>

You mapper class will be like:

|  |
| --- |
| **public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {** **… …** **private static HashMap<String, Integer> map = new HashMap<String, Integer>();** **public void map(Object key, Text value, Context context) throws IOException, InterruptedException {** **//Your map function** **}** **public void cleanup(Context context) throws IOException, InterruptedException {** **//iterate the HashMap object, and send the results to the reducer** **}****}** |

1. Do you need to change the reducer?
2. How to specify the configuration in the main function?
3. **Note:** All Java primitives are wrapped by the Writable class in Hadoop. Your key-in, key-out must be an instance of Writable! You cannot do below:

 **context.write(a String object, an Integer object)**

Instead, you need to wrap the String object by Text, and the Integer object by IntWritable, and then:

**context.write(a Text object, an IntWritable object)**

## Solutions of these Problems

## I hope that you can finish all problems by yourself, since the hints are already given. All the source codes will be published in the course homepage on Friday.