Advanced Data Management (CMM524)

Laboratory #6: Running a MapReduce Job

1. Aims

- To run a MapReduce job in the Cloudera Hadoop environment.

2. Outcomes

In completing this exercise, you should be able to:

- Create a MapReduce job from Java source code.
- Submit a MapReduce job to Hadoop.
- Stop a running MapReduce job.

3. Creating the `WordCount` Job

The `WordCount` job counts the number of occurrences of words in text files. (e.g. the Shakespeare collection).

3.1. The Job's Java Source Code

The job is written in the MapReduce Java API. To examine the source code:

- Open a Terminal window.
- Change into the Unix directory "`~/training_materials/developer/exercises/wordcount/src`" which contains the Java source code of the WordCount job.
    - What is the Unix command to use?
    - What is the Unix command to show your current working directory?
- List the content in the `stubs` sub-directory in Unix.
    - What is the Unix command to use?
    - What files/sub-directories do you find?
- Note:
    - You may find a `wordcount.jar` file is already in the folder, together with 3 `*.class` files inside `stubs`. In this case you don't need to compile the source, as they are already compiled. However, feel free to overwrite them as an exercise to compile and create a MapReduce job.

A MapReduce job consists of at least 3 Java classes: a driver class that sets up the job, a mapper class, and a reducer class.

- If you are interested, you can examine the classes using a text editor. However, this is not compulsory.
    - You can use a text editor to open the files.
    - **Make sure that you do not change the file content unless you know what you are doing.**

3.2. Compiling the Java Source Code

The next step is to compile the Java source (`*.java` files) into classes (`*.class` files).

- Make sure that you are in directory just above "`stubs`".
- At the Unix prompt, run the Java compiler to compile all `*.java` file in the `stubs` sub-directory:

```
javac -classpath `hadoop classpath` stubs/*.java
```

  - Notes:
    - **The "`" above are backward single quotes.** This makes Unix to execute the "`hadoop classpath`" command and puts the returned value into the above line. The reason for this interesting command is:
      - The "`hadoop classpath`" command returns a list of paths where the Hadoop Java libraries are located. Classes in these libraries are used by our `WordCount` program.
      - To compile `WordCount` successfully, we need to tell the Java compiler (i.e. `javac`) where to find 3rd-party classes (i.e. those defined in the Hadoop library) via its "`-classpath`" parameter.
      - If you are curious, you can execute the "`hadoop classpath`" command alone to see the paths of the Hadoop Java libraries.
    - The last part of the command "`stubs/*.java`" tells the Java compiler to look into the "`stubs`" sub-directory and compile all files with an extension of "`.java`".

If the compilation is successful, it should return no error.

- Check the content of the `stubs` sub-directory again.
  - Do you see any new file created?
    - How many new files are created by the compiler?
    - What file extension do the new files have?

3.3. Creating the Job Archive

The compiled Java classes (`*.class` files) must be archived into a JAR to create a MapReduce job.

- Make sure that you are still in the directory just above "`stubs`". i.e. "`~/training_materials/developer/exercises/wordcount/src`"
- At the Unix prompt, type the following command to create a `wordCount.jar` from the `*.classes` files in the stubs sub-directory:

```
jar cvf wordCount.jar stubs/*.class
```

- Note:
  - The options "`cvf`" tells the archiver (i.e. `jar`) to "*create*" with a "*verbose*" output to the screen and the "*filename*" is provided as follows.
  - This command collects all `*.class` files in the `stubs` sub-directory and archive them into the `wordCount.jar` file.
- Check to see you have the `wordCount.jar` file created.

4. Submitting a MapReduce Job

A MapReduce job reads input from HDFS files. We will use the Shakespeare collection we uploaded to HDFS in Lab#04.

- Make sure that you have the Shakespeare collection uploaded to HDFS.
  - What is the command to check the existence of the `shakespeare` directory in HDFS?
  - If you do not have the `shakespeare` directory in HDFS, upload it (refer to Lab#5).

Output of a MapReduce job is written to a non-exist directory in HDFS. **Hadoop will not write into an existing HDFS directory as it risks over-writing results that took a long time to compute.**

- Assuming that we will write the job output to a `shakeWordCount` HDFS directory. Make sure that this directory DOES NOT exist in HDFS.
  - What is the command to check the non-existence of this HDFS directory?

You are now ready to submit your `WordCount` job to Hadoop.

- At the Unix prompt, enter the following command to submit your `WordCount` job:

```
hadoop jar wordCount.jar stubs.WordCount shakespeare shakeWordCount
```

- Notes:
  - **The above is in 1 single line!**
  - The job driver class is `stubs.WordCount`.
  - The input are all files in the HDFS `shakespeare` directory.
  - The output will be written into the HDFS `shakeWordCount` directory.
- If you try to run the job a second time, Hadoop will refuse. Why?

5. Checking the Job Output

It is time to check out your job output.

- Check the content of the `shakeWordCount` HDFS directory.
  - What is the command to use?
  - What are inside the folder?
- Job results are in the `part-r-*` HDFS files.
  - What is the command to show the content of these result files?

6. The `AverageWordLength` Job

The `AverageWordLength` job counts the average length of all words that start with a character/alphabet while being case-sensitive. For example, for the input:

```
No now is definitely not the time
```

The output will be:

```
N    2.0
n    3.0
d    10.0
i    2.0
t    3.5
```

Java source files of the *AverageWordLength* job is in the `~/training_materials/developer/exercises/averagewordlength` folder.

6.1. The `AvgWordLength` Driver Class

- If you use the file browser, you may want to go into the `~/training_materials/developer/exercises/averagewordlength/src` folder first.
- Open `AvgWordLength.java` in a text editor. The file looks like this:

```
package stubs;
import org.apache.hadoop.mapreduce.Job;

public class AvgWordLength {
...
```

- Add the following lines **in blue** so that you can use classes defined in other libraries:

```
package stubs;
import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;

public class AvgWordLength {
…
```

- Scroll to the main body of the class, you will see the following:

```
    /*
     * TODO implement
     */
```

- Add the following lines **in blue** to complete the class:

```
/*
 * TODO implement
 */

if (args.length != 2)
       {
       System.out.printf("Usage: AvgWordLength <input dir> <output dir>\n");
       System.exit(-1);
       }

FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setMapperClass(LetterMapper.class);
job.setReducerClass(AverageReducer.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);
```

6.2. The `LetterMapper` Mapper Class

The `map` method of the Mapper receives a line of text from the `TextInputFormat` reader as a *(key,value)* pair, where the *key* is the offset of the line within the file and *value* is a single line of text. For example:

```
0    "No now is definitely not the time"
```

The Mapper ignores the key but splits the line into multiple words using spaces as delimiter. For each word, it emits the first letter of the word as a key, and the length of the word as value. For example:

```
N    2
n    3
I    2
d    10
n    3
t    3
t    4
```

- Open the `LetterMapper.java` file in a text editor.
- Scroll to the main body of the class, you will see the following:

```
/*
 * TODO implement
 */
```

- Add the following lines **in blue** to complete the class:

```
/*
 * TODO implement
 */

String line = value.toString();

for (String word : line.split("\\W+"))
{
if (word.length() > 0)
    {
    String letter = word.substring(0,1);
    context.write(new Text(letter), new IntWritable(word.length()));
    }
}
```

6.3. The `AverageReducer` Reducer Class

After shuffle-and-sort, the `reduce` method of the Reducer receives intermediate data in *(key, iterable of values…)* pairs:

```
N     (2)
d     (10)
i     (2)
n     (3,3)
t     (3,4)
```

Then it calculates the average word length for each key and outputs:

```
N     2.0
d     10.0
i     2.0
n     3.0
t     3.5
```

- Open the `AverageReducer.java` file in a text editor. You fill find the following section:

```
/*
 * TODO implement
 */
```

- **Add the following lines in blue** to complete the class:

```
/*
* TODO implement
*/

long sum = 0, count = 0;

for (IntWritable value : values)
    {
    sum += value.get();
    count++;
    }
if (count != 0)
    {
    double result = (double)sum / (double)count;
    context.write(key, new DoubleWritable(result));
    }
```

6.4. Running the `AvgWordLength` Job

To run the `AvgWordLength` job:

- Compile the 3 Java class by adapting the command in section 3.2.
    - Note: You may want to change into the folder just above "`stubs`" before compiling your Java program.
- Create a job archive by adapting the command in section 3.3.
    - Note: Again, make sure that you are in the correct folder before you do the archiving, and know where the JAR file will be saved.
- Finally, submit your `AvgWordLength` job to work on the `shakespeare` dataset.