Lab 3

Objectives

- Write a simple MapReduce program.
- Customize the MapReduce program by accepting user input.
- Run Hadoop MapReduce programs in standalone and pseudo-distributed modes.

Prerequisites

- Setup the development environment as explained in Lab1.
- Download these two sample files sample file 1, sample file 2. Decompress the second file after download.
- For Windows users, install the Ubuntu app from Microsoft Store and set it up. We will need it to run YARN easily.

Lab Work

I. Setup (10 minutes)

1. Create a new empty project using Maven for Lab 3. See Lab 1 for more details.
2. Import your project into IntelliJ IDEA.
3. Copy the file `$HADOOP_HOME/etc/hadoop/log4j.properites` to your project directory under `src/main/resources`. This allows you to see internal Hadoop log messages when you run in IntelliJ IDEA.
4. Place the two sample files in your project home directory.
5. In `pom.xml` add the following dependencies.

```
<properties>
  <hadoop.version>2.10.0</hadoop.version>
</properties>

<dependencies>
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-common</artifactId>
      <version>${hadoop.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-hdfs</artifactId>
      <version>${hadoop.version}</version>
    </dependency>
```

```
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-mapreduce-client-common</artifactId>
      <version>${hadoop.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-mapreduce-client-core</artifactId>
      <version>${hadoop.version}</version>
    </dependency>

</dependencies>
```

II. Simple Filter Program (30 minutes)

In this part, you will need to write a MapReduce program that produces the lines that have a specific response code in them.

1. Take a few minutes to look into the sample file and understand its format. You can import the file into Excel or other spreadsheet program to make it easier to understand.
2. Create a new class named `Filter` in package `edu.ucr.cs.cs167.<UCRNetID>` and add a main function to it.
3. Add the following code stub in your filter class.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;


/**
 * Filter log file by response code
 */
public class Filter {
    public static class TokenizerMapper
            extends Mapper<LongWritable, Text, NullWritable, Text> {

        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            super.setup(context);
            // TODO add additional setup to your map task, if needed.
        }

        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            if (key.get() == 0)
                return; // Skip header line
            String[] parts = value.toString().split("\t");
            String responseCode = parts[5];
            // TODO Filter by response code
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "filter");
        job.setJarByClass(Filter.class);
        job.setMapperClass(TokenizerMapper.class);
```

```
        job.setNumReduceTasks(0);
        job.setInputFormatClass(TextInputFormat.class);
        Path input = new Path(args[0]);
        FileInputFormat.addInputPath(job, input);
        Path output = new Path(args[1]);
        FileOutputFormat.setOutputPath(job, output);
        // String desiredResponse = args[2];
        // TODO pass the desiredResponse code to the MapReduce program
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4. Take some time to understand the code and answer the following questions.

- (Q1) What do you think the line `job.setJarByClass(Filter.class);` does?
- (Q2) What is the effect of the line `job.setNumReduceTasks(0);` ?
- (Q3) Where does the `main` function run? (Driver node, Master node, or a slave node).

5. Initially, to make the code easy to run and test, we want the program to filter all the lines with response code `200` . We will hard code the value `200` instead of accepting it from the user input. To do so, replace the `// TODO Filter by response code` with the following code snippet and make any required adjustments to the code.

```
if (responseCode.equals("200"))
    context.write(NullWritable.get(), value);
```

Notice that we use `String#equals` rather than the operator `==` since `String` is not a primitive value.

6. Run the code with the following command line arguments `nasa_19950801.tsv filter_output.tsv` . Make sure to delete the output directory before you run if you execute your program several times.
7. Check the output. (Q4) How many lines do you see in the output?
8. Compile and run your program from the command line using the `hadoop jar` command.

```
hadoop jar target/<*.jar> edu.ucr.cs.cs167.[NetID].Filter nasa_19950801.tsv filter_output.tsv
```

III. Take User Input For the Filter (20 minutes)

In this part, we will customize our program by taking the desired response code from the user as a command line argument.

1. Uncomment the line `// String desiredResponse = args[2];` in the `main` function.
2. Add the desired respose code to the job configuration using the method `Configuration#set` .
3. In the setup function, add a code that will read the desired response code from the job configuration and store it in an instance variable in the class `TokenizerMapper` .
4. Modify the `map` function to use the user given response code rather than the hard-coded response code that we used in Part II.
5. Run your program again to filter the lines with response code `200` . This time, you will need to pass it as a command-line argument.
6. Run it from IntelliJ IDEA on a file in your local file system. (Q5) How many files are produced in the output? (Q6) Explain this number based on the input file size and default block size.
7. Run it from the command line on a file in HDFS. (Q7) How many files are produced in the output? (Q8) Explain this number based on the input file size and default block size.

Note: Make sure that you run the namenode and datanode from the command line to access HDFS as explained in Lab 2.

Note: If you run your program from the command-line without setting up YARN (see next section), then it runs in standalone mode.

IV. Run in Pseudo-distributed Mode (45 minutes)

To run your MapReduce program in pseudo-distributed mode, we will need to configure Hadoop to use YARN and start YARN instances.

1. Configure Hadoop to run MapReduce programs with YARN. Edit the file `$HADOOP_HOME/etc/hadoop/mapred-site.xml` and add the following part.

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

Note: If you do not have a `mapred-site.xml` file, maky a copy of `mapred-site.xml.template` and name it `mapredi-site.xml` .

2. Edit the file `$HADOOP_HOME/etc/hadoop/yarn-site.xml` and add the following part.

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

3. Start the resource manager (Master). In a new command line window, run `yarn resourcemanager` . Leave the process running on that window.
4. Start the node manager (Slave). In a new command line window, run `yarn nodemanager` . Leave the process running on that window.

Note: For Windows users, run the above two commands in an Ubuntu window rather than a regular command-line or PowerShell windows. For compatibility, run all the five processes in Ubuntu windows, that is, Resource Manager, Node Manager, Name Node, Data Node, and Driver command.

5. Generate a JAR file for your program and run it using the command `yarn jar <*.jar> <input> <output> <code>` .

```
yarn jar target/<*.jar> nasa_19950801.tsv filter_output.tsv 200
```

6. If you did not do already, start HDFS as described in Lab 2 and run your program on an input file that is stored in HDFS and produce the output in HDFS.

V. Write an Aggregate Program (30 minutes)

In this part, we will create another MapReduce program that computes the total bytes for each response code. That is the sum of the column `bytes` grouped by the column `response` .

1. Create a new class `Aggregation` based on the following stub code.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;
```

```
public class Aggregation {
  public static class TokenizerMapper extends Mapper<LongWritable, Text, IntWritable, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private IntWritable outKey = new IntWritable();
    private IntWritable outVal = new IntWritable();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
      if (key.get() == 0)
        return;
      String[] parts = value.toString().split("\t");
      int responseCode = Integer.parseInt(parts[5]);
      int bytes = Integer.parseInt(parts[6]);
      // TODO write <responseCode, bytes> to the output
    }
  }

  public static class IntSumReducer
      extends Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(IntWritable key, Iterable<IntWritable> values,
                       Context context) throws IOException, InterruptedException {
      // TODO write <key, sum(values)> to the output
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "aggregation");
    job.setJarByClass(Aggregation.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setMapOutputKeyClass(IntWritable.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(IntWritable.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setNumReduceTasks(2);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

2. Implement the TODO items to make the desired logic. Hint: look at the WordCount example.
3. Run your program on the file `nasa_19950801.tsv` and check the output directory. (Q9) How many files are produced in the output directory and how many lines are there in each file? (Q10) Explain these numbers based on the number of reducers and number of response codes in the input file.
4. Run your program on the file `nasa_19950630.22-19950728.12.tsv`. (Q11) How many files are produced in the output directory and how many lines are there in each file? (Q12) Explain these numbers based on the number of reducers and number of response codes in the input file.
5. Run your program on the output of the `Filter` operation with response code `200`. (Q13) How many files are produced in the output directory and how many lines are there in each file? (Q14) Explain these numbers based on the number of reducers and number of response codes in the input file.

VI. Submission (15 minutes)

1. Add a `README` file with all your answers.
2. Add a `run` script that runs compiles and runs your filter operation on the sample input file with response code 200. Then, it should run the aggregation method on the same input file. The output files should be named `filter_output` and `aggregation_output` accordingly.

Common Errors

- Error: When I run my program on YARN, I see an error message similar to the following.

```
Failing this attempt.Diagnostics: [...]Container [pid=xxx,containerID=xxx] is running beyond virtual memory
limits. Current usage: xxx MB of yyy GB physical memory used; xxx TB of yyy GB virtual memory used. Killing
container.
```

- Fix: Add the following configuration to your `$HADOOP_HOME/etc/yarn-site.xml` .

```
<property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>
```

See also