Word Count Code using MR2 Classes and API

A Guide to Understand the Execution of Word Count

edureka!

edureka!

A guide to understand the execution and flow of word count

WRITE YOU FIRST MRV2 PROGRAM AND EXECUTE IT ON YARN IN HADOOP 2.0

As discussed in the previous blog, Hadoop 2.0, MRv2 and YARN are here to make Hadoop enterprise ready and take it to the next level. If you haven't, create a single Node Hadoop 2.0 cluster to start your hadoop 2.0 journey. This blog helps you to write and execute the first MRv2 Program.

Jars Used As Reference Libraries

hadoop-mapreduce-client-core-2.2.0.jar

hadoop-common-2.2.0.jar

These are the jars present in Hadoop-2.2.0 distribution. For your help the jars are zipped along with this document.

edureka!

Packages Imported

```
import java.io.IOException;
import java.util.*;
```

Java Libraries remains same for MR1 and MR2

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
```

All these packages are present Hadoop-common-2.2.0.jar

```
import org.apache.hadoop.mapreduce.*;
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 org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
```

All these packages are present in hadoop-mapreduce-client-core-2.2.0.jar

Main Class

Name of the Main Class or Driver Class within which we have our Mapper and Reducer Class and the main Method

```
public class WordCountNew {
```

MAPPER CLASS

Name of the Mapper Class which inherits Super Class Mapper

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

edureka!

Mapper Class takes 4 Arguments i.e.

Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

```
//Defining a local variable one of type IntWritable
        private final static IntWritable one = new     IntWritable(1);
//Defining a local variable word of type Text
        private Text word = new Text();
```

We override the map method which is defined in the Parent (Mapper) Class. It takes 3 arguments as Inputs

map (KEYIN key, VALUEIN value, Context context)

```
public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
```

```
//Converting the record (single line) to String and storing it
        in a String variable line
        String line = value.toString();

//StringTokenizer is breaking the record (line) into words
        StringTokenizer tokenizer = new StringTokenizer(line);

        //Running while loop to get each token(word) one by one from
        StringTokenizer
                        while (tokenizer.hasMoreTokens()) {
                        //Saving the token(word) in a variable word
                                word.set(tokenizer.nextToken());
                        //Writing the output as (word, one), the value
                                of word will be equal to token and value
                                of one is 1
                                context.write(word, one);

                }
        }
}
```

edureka!

In the map method, we receive a **record** (single line). It is stored in a string variable line. Using **StringTokenizer**, we are breaking the line into individual words called **tokens**, on the basis of space as **delimiter**. If the line was Hello There, StringTokenizer will give two tokens Hello and There. Finally using the context object we are dumping the **Mapper** output. So as per our example the Output from the Mapper will be

Hello 1 & There 1 and so on.

The Output from the Mapper is taken as Input by the Reducer

Name of the **Reducer Class which inherits Super Class Reducer**

```
public static class Reduce extends
                Reducer<Text, IntWritable, Text, IntWritable> {
```

Reducer Class takes 4 Arguments i.e.

Reducer <KEYIN, VALUEIN, KEYOUT, VALUEOUT>

We override the reduce method which is defined in the Parent (Reduce) Class. It takes 3 arguments as Inputs

reduce (KEYIN key, VALUEIN value, Context context)

```
public void reduce(Text key, Iterable<IntWritable> values,
                                Context context) throws IOException,
InterruptedException {

//Defining a local variable sum of type int
                        int sum = 0;
//Running for loop to iterate over the values present in
                Iterator
                        for (IntWritable val : values) {
                        //We are adding the value to the variable over
                                every iteration
                                sum = sum + val.get();

//Finally writing the key and the value of sum(number of times
        the word occurred in the input file) to the output file
                        context.write(key, new IntWritable(sum));

                }
        }
```

edureka!

In the reduce method, we receive a key as word and a list of values as input.

For eg: Hello<1,1,1,1>

So to find out the occurrence of the word Hello in the input file then we simply have to sum all the values of the list. Hence we run a for loop to iterate over the values one by one and adding it to variable sum.

Finally we will write the output i.e key (word) & value (sum) using the context object.

So as per the above example the output will be :

Hello 4

main method known as entry point of the application. This is the method which is called as soon as jar is executed

```
public static void main(String[] args) throws Exception {

//Creating an object of Configuration class, which loads the
        configuration parameters
                Configuration conf = new Configuration();

//Creating the object of Job class and passing the conf object
        and Job name as arguments. The Job class allows the user
        to configure the job, submit it and control its execution.

                Job job = new Job(conf, "wordcount");

//Setting the jar by finding where a given class came from
                job.setJarByClass(WordCountNew.class);

//Setting the key class for job output data
                job.setOutputKeyClass(Text.class);

//Setting the value class for job output data
                job.setOutputValueClass(IntWritable.class);

//Setting the mapper for the job
                job.setMapperClass(Map.class);

//Setting the reducer for the job
                job.setReducerClass(Reduce.class);

//Setting the Input Format for the job
                job.setInputFormatClass(TextInputFormat.class);

//Setting the Output Format for the job
                job.setOutputFormatClass(TextOutputFormat.class);

//Adding a path which will act as a input for MR job. args[0]
        means it will use the first argument written on terminal
        as input path
        FileInputFormat.addInputPath(job, new Path(args[0]));
```

edureka!

```
//Setting the path to a directory where MR job will dump the
        output. args[1] means it will use the second argument written on terminal as
        output path
        FileOutputFormat.setOutputPath(job,new
                Path(args[1]));


//Submitting the job to the cluster and waiting for its
        completion
                job.waitForCompletion(true);
        }

}
```

**********NOW EXPORT THE JAR AND RUN THE CODE**********

edureka!