code starts with importing necessary Java libraries and classes

import java.io.IOException;

import java.util.StringTokenizer;

2 library for java

9 library for hadoop

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

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.output.FileOutputFormat;

1. The **WordCount** class serves as the main class that configures and runs the MapReduce job.

public class WordCount {

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

2. In the **main** method, a new Hadoop configuration and job instance are created. The job is named "word count"

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");

the **WordCount** class is set as the jar for the job.

job.setJarByClass(WordCount.class);

3. The mapper and reducer classes are specified using the **setMapperClass** and **setReducerClass** methods.

job.setMapperClass(WordCountMapper.class);

job.setCombinerClass(WordCountReducer.class);

job.setReducerClass(WordCountReducer.class);

4. The output key and value classes are set to **Text** and **IntWritable** respectively.

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

1. The input and output file paths are provided through command-line arguments (**args[0]** and **args[1]**) using the **FileInputFormat** and **FileOutputFormat** classes.

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

1. The **job.waitForCompletion(true)** method starts the job and waits for it to complete. It returns **true** if the job is successful, and **false** otherwise.

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

1. The **WordCountMapper** class extends the **Mapper** class provided by Hadoop. It specifies the input key-value types (**Object**, **Text**) and the output key-value types (**Text**, **IntWritable**) for the mapper.

public static class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {

2. It declares two private instance variables: **one** of type **IntWritable** and **word** of type **Text**.

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

private Text word = new Text();

The **map** method is overridden from the **Mapper** class and is responsible for processing each input key-value pair.

1. The **map** method takes four parameters:
   * **Object key**: represents the input key. In this case, it is not used.
   * **Text value**: represents the input value, which is a line of text.
   * **Context context**: provides access to Hadoop's facilities for interacting with the Hadoop framework.

public void map(Object key, Text value, Context context) throws IOException,

InterruptedException {

1. Inside the **map** method, a **StringTokenizer** named **itr** is created, which tokenizes the input line of text into individual words. The **value.toString()** converts the **Text** value to a string before tokenizing.

StringTokenizer itr = new StringTokenizer(value.toString());

1. Using a **while** loop, the **hasMoreTokens** method of **itr** is checked to see if there are more words to process.
2. Inside the loop, the next token (word) is obtained using **itr.nextToken()**, and it is set as the value of the **word** instance variable.
3. The **context.write(word, one)** statement writes the word as the output key and the value **one** (which is set to **1**) as the output value. This emits a key-value pair from the mapper.

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

}

}

}

public static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

}