This documentation provides a comprehensive overview of the Hadoop **WordCount** MapReduce example, explaining the key concepts, code structure, and clarifying the topics you've asked about multiple times.

### **1. Introduction to MapReduce in Hadoop**

MapReduce is a programming model used for processing large datasets in parallel across a distributed cluster. It involves two primary functions:

* **Map Function:** Processes input data into key-value pairs.
* **Reduce Function:** Aggregates the results of the Map phase based on the keys.

The **WordCount** program is often used as a starting point in Hadoop to demonstrate how MapReduce works.

### **2. Overview of the WordCount Program**

The WordCount program counts the occurrences of each word in a given text file. It involves three main components:

1. **Mapper Class**: Reads input and generates key-value pairs.
2. **Reducer Class**: Aggregates the counts for each word.
3. **Main Class**: Configures and runs the MapReduce job.

### **3. Code Explanation**

| 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;  import java.io.IOException;  import java.util.StringTokenizer;  public class WordCount {  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {  private final static IntWritable one = new IntWritable(1);  private Text word = new Text();  public void map(Object key, Text value, Context context) throws IOException, InterruptedException {  StringTokenizer itr = new StringTokenizer(value.toString());  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one);  }  }  }  public static class IntSumReducer 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);  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf, "word count");  job.setJarByClass(WordCount.class);  job.setMapperClass(TokenizerMapper.class);  job.setCombinerClass(IntSumReducer.class);  job.setReducerClass(IntSumReducer.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.addInputPath(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  System.exit(job.waitForCompletion(true) ? 0 : 1);  }  } |
| --- |

#### **3.1. The WordCount Class**

This is the main class containing the MapReduce job configuration, including the Mapper, Reducer, and the job setup in the main() method.

| public class WordCount { |
| --- |

#### **3.2. The Mapper Class**

The **Mapper** class processes the input data (text) and generates intermediate key-value pairs. In this case, the key is a word, and the value is the count (always 1).

| public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {  private final static IntWritable one = new IntWritable(1);  private Text word = new Text();  public void map(Object key, Text value, Context context) throws IOException, InterruptedException {  StringTokenizer itr = new StringTokenizer(value.toString());  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one); // Emit word and count (1)  }  }  } |
| --- |

* **StringTokenizer:** Splits each line of text into words.
* **context.write(word, one):** Emits a word paired with a count of 1.
* **IntWritable one = new IntWritable(1):** Instead of using primitive int, we use the IntWritable class (a Hadoop-specific class) to wrap the integer 1. This is required because Hadoop works with **Writable** types, not primitive types.

#### **3.3. The Reducer Class**

The **Reducer** class processes the intermediate key-value pairs, groups them by key (word), and sums the counts.

| public static class IntSumReducer 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(); // Sum up the counts  }  result.set(sum); // Set the total count  context.write(key, result); // Emit the word and total count  }  } |
| --- |

* The **reduce() method** sums up the counts of each word and writes the final result as a key-value pair (word, total count).

#### **3.4. The Main Method**

The **main()** method configures and runs the MapReduce job. It sets the Mapper and Reducer classes, specifies input and output paths, and executes the job.

| public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf, "word count");  job.setJarByClass(WordCount.class);  job.setMapperClass(TokenizerMapper.class);  job.setCombinerClass(IntSumReducer.class); // Optional: Combiner for local aggregation  job.setReducerClass(IntSumReducer.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.addInputPath(job, new Path(args[0])); // Input path from args[0]  FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path from args[1]  System.exit(job.waitForCompletion(true) ? 0 : 1); // Run the job and wait for completion  } |
| --- |

* **FileInputFormat.addInputPath(job, new Path(args[0]));**: Specifies the input directory passed via the first command-line argument (args[0]).
* **FileOutputFormat.setOutputPath(job, new Path(args[1]));**: Specifies the output directory passed via the second command-line argument (args[1]).

### **4. Key Concepts Clarified**

#### **4.1. Why context.write(word, one)?**

You asked why one (an IntWritable object) is used instead of just the number 1. The answer is that Hadoop expects data to be in **Writable** types, not primitive data types. **IntWritable** is Hadoop’s way of wrapping an int so it can be serialized for storage and processing in the distributed system.

#### **4.2. What is args[0] and Why Use It?**

The args[] array holds **command-line arguments** passed when running the job.

* **args[0]**: This is the first command-line argument, which represents the **input path** (location of the data to be processed).
* **args[1]**: This is the second argument, which represents the **output path** (where the results should be stored).

Using args[] allows you to run the program on different datasets without modifying the code.

### **5. Running the Job**

To run the MapReduce job, you need to compile the Java code, create a JAR file, upload the input file to HDFS, and then execute the job.

**Compile the Java Code:**

| javac -classpath `hadoop classpath` -d . WordCount.java  jar cf wc.jar \*.class |
| --- |

*Make sure WoedCount.java is the path of your java file!!!*

**Upload the Input Data to HDFS:**

| sh  echo "Hadoop is fun. Hadoop is powerful. Hadoop is scalable." > input.txt  hdfs dfs -mkdir /wordcount  hdfs dfs -put input.txt /wordcount/ |
| --- |

**Run the MapReduce Job:**

| hadoop jar wc.jar WordCount /wordcount/input.txt /wordcount/output |
| --- |

Here while running the WordCount class the first argument which is args[0] will be input file location which is /wordcount/input.txt and args[1] will be output fill location. We use args[0] and args[1] in our code as file paths.

**Check the Output:**

| hdfs dfs -cat /wordcount/output/part-r-00000 |
| --- |

hdfs is not traditional file system. Here we can’t use cd. But we can use ls, cat, nano etc. so to check all the files in output folder we can use hdfs dfs -ls /wordcount/output

### **6. File Output Explanation (part-r-00000)**

The output file part-r-00000 is generated by the **Reducer**. Hadoop names output files in the format part-r-<index> to distinguish them when there are multiple reducers. Since you're using a single reducer, the output is in part-r-00000.

* **part**: Indicates the file is a part of the final output.
* **r**: Indicates the file comes from the Reducer.
* **00000**: The index of the file. If there were multiple reducers, you'd see files like part-r-00000, part-r-00001, etc.

### **7. Conclusion**

This document covered the entire process of running a **WordCount** MapReduce job in Hadoop. We discussed:

* The MapReduce programming model.
* The structure and functionality of the **WordCount** Java code.
* Specific clarifications regarding the use of IntWritable instead of int and why args[0] is used for the input path.

If you have any further questions or need more clarification on any part, feel free to ask!

**Practice:**

* Identify the top N most frequent words in a large text dataset.
* Calculate the average word length in a text file.
* Count the frequency of each character (instead of words) in a large text file
* etc…