BIG DATA ANALYTICS - VIVA PREPARATION GUIDE

Most Probable Questions for MapReduce and Apache Spark

==

TOPIC 1: WORDCOUNT APPLICATION USING HADOOP MAPREDUCE

Sample Code Implementation:

The WordCount application is considered the "Hello World" of MapReduce. It counts occurrences of each word in a given input set.

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

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;

public class WordCount {

// Mapper Class
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().toLowerCase());
      context.write(word, one);
    }
  }

}
```

// Reducer Class
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);
  }
```

}

// Main Method
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);
}
}

LIKELY VIVA QUESTIONS ON WORDCOUNT MAPREDUCE APPLICATION

1. What is the purpose of the WordCount application in Hadoop? Answer: WordCount is a basic MapReduce application that counts the occurrence of each word in a set of input files. It demonstrates the fundamental concepts of MapReduce by splitting the task into a map phase (which tokenizes text and emits word-count pairs) and a reduce phase (which aggregates all counts for each unique word).

2. Explain the role of Mapper and Reducer in your WordCount implementation. Answer: The TokenizerMapper class splits input text into words and emits a key-value pair (word, 1) for each word. The IntSumReducer class receives all values for a given key (word) and sums these values to get the total count for each word.
3. What are IntWritable and Text classes in Hadoop, and why are they used instead of standard Java types? Answer: IntWritable and Text are Hadoop's serializable counterparts to Java's Integer and String. They implement Hadoop's Writable interface, which provides efficient serialization/deserialization methods required for data transfer between nodes in a distributed environment. This makes them more efficient for network transfer than standard Java types.
4. Explain the importance of the Combiner in your WordCount application. Answer: The Combiner acts as a mini-reducer that runs on the mapper's output before data is transferred across the network to reducers. In WordCount, it performs local aggregation of word counts, which significantly reduces the amount of data transferred over the network, improving overall job performance.
5. How would you modify this WordCount application to ignore common words like "the", "and", "a"? Answer: I would create a set of stop words in the Mapper class and check each tokenized word against this set. If the word is found in the stop words set, the mapper would skip emitting that word. This can be implemented by adding: private Set<String> stopWords = new HashSet<>(Arrays.asList("the", "and", "a", "to", "of", "in")); // In map method: if (!stopWords.contains(word.toString())) { context.write(word, one); }
6. How would you make the WordCount application case-insensitive? Answer: In the map method, I would convert each token to lowercase before emitting it as a key: word.set(itr.nextToken().toLowerCase());
7. If your input data is very large, how does Hadoop ensure efficient processing? Answer: Hadoop processes large data efficiently through:
   - Data locality: Moving computation to data rather than data to computation
   - Data splitting: Breaking large files into blocks and processing them in parallel
   - Distributed processing: Utilizing multiple nodes for parallel execution
   - Fault tolerance: Automatically recovering from node failures
   - Combiners: Performing local aggregations to reduce network transfer
8. Explain the data flow in this WordCount application. Answer:
   1. Input files are split and distributed to mappers
   2. Mappers tokenize text and emit (word, 1) pairs
   3. Pairs are partitioned, sorted by key, and sent to combiners (if configured)
   4. Combiners perform local aggregation, producing (word, partial_count) pairs
   5. These pairs are shuffled and sorted across the network
   6. Reducers receive all values for each key and sum them

   7. Final (word, total_count) pairs are written to output files

9. What configuration changes would you make to optimize the WordCount job for very large datasets?
   Answer:
   - Increase the number of reducers based on data size
   - Enable and configure combiners to reduce network traffic
   - Adjust memory allocation for mappers and reducers
   - Use compression for intermediate data
   - Tune the split size to optimize parallelism
   - Use a custom partitioner if word distribution is skewed

10. How would you modify this code to count word frequency in a specific file format like JSON or XML?
    Answer: I would replace the simple StringTokenizer with a proper parser for the specific format. For JSON, I might use a library like Jackson to parse the documents and extract relevant text fields before tokenization. Additionally, I would need to modify the mapper to handle the structured format and potentially extract only specific fields for word counting.

==

TOPIC 2: PROCESSING LOG FILES WITH MAPREDUCE

Sample Code Implementation:

This application processes system log files to extract and analyze information such as error counts, request patterns, or system performance metrics.

```
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

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;

public class LogAnalyzer {
```

```
// Regular expression pattern for Apache log format
private static final Pattern LOG_PATTERN = Pattern.compile(
"^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(.+?)" (\d{3}) (\d+) "([^"]+)" "([^"]+)"");

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

```
private final static IntWritable one = new IntWritable(1);
private Text outputKey = new Text();

@Override
public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
  String logEntry = value.toString();
  Matcher matcher = LOG_PATTERN.matcher(logEntry);

  if (matcher.find()) {
    // Extract IP address
    String ipAddress = matcher.group(1);

    // Extract HTTP status code
    String statusCode = matcher.group(6);

    // Extract request URL from the request field
    String request = matcher.group(5);
    String[] parts = request.split(" ");
    String url = "";
    if (parts.length > 1) {
      url = parts[1];
    }

    // Emit IP address count
    outputKey.set("IP:" + ipAddress);
    context.write(outputKey, one);

    // Emit status code count
    outputKey.set("STATUS:" + statusCode);
    context.write(outputKey, one);

    // Emit URL count
    if (!url.isEmpty()) {
      outputKey.set("URL:" + url);
      context.write(outputKey, one);
    }

    // Check for error status codes (4xx and 5xx)
    int status = Integer.parseInt(statusCode);
    if (status >= 400) {
      outputKey.set("ERROR");
      context.write(outputKey, one);
    }
    }
  }
```

}

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

```
  @Override
  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, "log analyzer");
job.setJarByClass(LogAnalyzer.class);
job.setMapperClass(LogMapper.class);
job.setCombinerClass(LogReducer.class);
job.setReducerClass(LogReducer.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);
}
}

LIKELY VIVA QUESTIONS ON LOG FILE PROCESSING

1. What is the purpose of processing log files using MapReduce? Answer: Processing log files with MapReduce allows for efficient analysis of large volumes of log data to extract valuable insights such as traffic patterns, error rates, user behavior, security issues, and system performance metrics. The distributed processing model of MapReduce handles the scale of log data that would overwhelm traditional processing methods.

2. Explain the regular expression pattern used in your log analyzer. What log format is it designed to parse? Answer: The regular expression pattern is designed to parse the Common Log Format (CLF) used by web servers like Apache. It captures components such as IP address, timestamp, HTTP method, URL, status code, bytes transferred, and user agent. Each component is captured in a separate group for easy extraction in the mapper.
3. How does your LogMapper extract meaningful information from log entries? Answer: The LogMapper uses regex pattern matching to extract key components from each log entry. It then categorizes and emits different types of data (IP addresses, status codes, URLs) with appropriate prefixes to enable multiple analyses from a single pass through the data. It also performs specific detection of error status codes (4xx and 5xx).
4. What are the advantages of using MapReduce for log analysis compared to traditional scripting languages? Answer:
   - Scalability: Handles terabytes or petabytes of log data across clusters
   - Parallelism: Processes logs in parallel across many machines
   - Fault tolerance: Continues processing despite node failures
   - Data locality: Minimizes network transfer by moving computation to data
   - Built-in aggregation: Efficiently counts and summarizes metrics
5. How would you modify this code to identify potential security threats like brute force attacks? Answer: To identify potential brute force attacks, I would:
   - Track login failure attempts by IP address
   - Use a time window to identify high-frequency failures
   - Implement a secondary MapReduce job that takes the output from the first job and identifies IPs with login failure counts above a threshold within a specific timeframe
   - Could use a custom writable class to track timestamp information along with counts
6. Explain how you would enhance this log analyzer to perform time-based analysis. Answer: I would extract and parse the timestamp field into a structured format (e.g., hour of day), then emit time-based keys along with the metrics of interest. For example: // Extract and parse timestamp String timestamp = matcher.group(4); SimpleDateFormat inputFormat = new SimpleDateFormat("dd/MMM/yyyy:HH:mm
   Z"); Date date = inputFormat.parse(timestamp); // Extract hour for hourly analysis Calendar cal = Calendar.getInstance(); cal.setTime(date); int hour = cal.get(Calendar.HOUR_OF_DAY); // Emit hourly metrics outputKey.set("HOUR:" + hour + ":STATUS:" + statusCode); context.write(outputKey, one);
7. How would you implement a secondary sort in MapReduce to sort log entries by both timestamp and status code? Answer: I would implement a composite key class that holds both timestamp and status code, along with a custom partitioner and comparator:
   1. Create a composite key class implementing WritableComparable

   2. Implement custom partitioning to ensure entries with the same timestamp go to the same reducer
   3. Implement a custom sort comparator that sorts by timestamp first, then by status code
   4. The natural sort order would be used in the reducer to process entries in the desired order

8. What challenges might you face when processing log files from multiple sources with different formats? Answer: Challenges include:
   - Handling different log formats requires multiple regex patterns or parsers
   - Time synchronization issues between different servers
   - Inconsistent field naming or meaning across sources
   - Varying severity levels and error classifications
   - Solutions include creating format-specific mappers or a flexible mapper that detects and adapts to different formats
9. How would you optimize this log analyzer for extremely large log datasets? Answer:
   - Use log file compression to reduce storage and I/O
   - Implement custom input formats to efficiently split compressed logs
   - Filter irrelevant log entries early in the mapper
   - Use counters instead of emitting data for common metrics
   - Implement sampling techniques for trend analysis rather than processing all entries
   - Use the combiner aggressively to reduce network transfer
   - Consider using a bloom filter for common patterns
10. How can you extend this MapReduce job to generate visualizations or reports from the log analysis? Answer: The MapReduce job would output structured data that can be:
    - Processed by visualization tools like Tableau or Power BI
    - Loaded into a database or data warehouse for reporting
    - Fed into another MapReduce job that generates HTML reports
    - Used as input for a charting library in a web application
    - Alternatively, we could write a custom reducer that formats the results directly into HTML or JSON format

==

TOPIC 3: WEATHER DATA ANALYSIS USING MAPREDUCE

Sample Code Implementation:

This application processes weather data to calculate average temperature, dew point, and wind speed.

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
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.output.FileOutputFormat;

public class WeatherDataAnalyzer {

public static class WeatherMapper extends Mapper<LongWritable, Text, Text, DoubleWritable> {
```

```java
private Text weatherAttribute = new Text();
private DoubleWritable value = new DoubleWritable();

@Override
public void map(LongWritable key, Text input, Context context) throws IOException,
InterruptedException {
  String line = input.toString();
  String[] fields = line.split(",");

  // Ensure we have a valid record with enough fields
  if (fields.length < 5) {
    return; // Skip invalid records
  }

  try {
    // Parse the temperature field (assuming position 2)
    // Skip if temperature is missing or invalid
    if (!fields[2].trim().equals("") && !fields[2].trim().equals("9999")) {
      double temperature = Double.parseDouble(fields[2].trim());
      weatherAttribute.set("temperature");
      value.set(temperature);
      context.write(weatherAttribute, value);
    }

    // Parse the dew point field (assuming position 3)
    // Skip if dew point is missing or invalid
    if (!fields[3].trim().equals("") && !fields[3].trim().equals("9999")) {
      double dewPoint = Double.parseDouble(fields[3].trim());
      weatherAttribute.set("dewpoint");
      value.set(dewPoint);
      context.write(weatherAttribute, value);
    }

    // Parse the wind speed field (assuming position 4)
    // Skip if wind speed is missing or invalid
    if (!fields[4].trim().equals("") && !fields[4].trim().equals("9999")) {
      double windSpeed = Double.parseDouble(fields[4].trim());
      weatherAttribute.set("windspeed");
      value.set(windSpeed);
      context.write(weatherAttribute, value);
    }
  } catch (NumberFormatException e) {
    // Skip record with parsing errors
    context.getCounter("WeatherData", "ParseErrors").increment(1);
    }
  }
```

}

public static class AverageReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {
private DoubleWritable result = new DoubleWritable();

```
  @Override
  public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
      throws IOException, InterruptedException {
    double sum = 0;
    long count = 0;

    // Calculate the sum and count for computing average
    for (DoubleWritable val : values) {
      sum += val.get();
      count++;
    }

    // Calculate average
    double average = sum / count;
    result.set(average);

    // Write the result
    context.write(key, result);

    // Write count as a counter for reference
    context.getCounter("WeatherData", key.toString() + "Count").setValue(count);
  }
```

}

public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "weather data analyzer");
job.setJarByClass(WeatherDataAnalyzer.class);
job.setMapperClass(WeatherMapper.class);
job.setCombinerClass(AverageReducer.class);
job.setReducerClass(AverageReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

LIKELY VIVA QUESTIONS ON WEATHER DATA ANALYSIS

1. Explain the purpose of this weather data analysis application and what insights it provides. Answer: This MapReduce application processes weather data to calculate average temperature, dew point, and wind speed across a dataset. These averages provide insights into climate patterns, weather trends, and can be used for meteorological research, agriculture planning, or climate change studies.
2. How does your application handle missing or invalid data in the weather dataset? Answer: The application employs several validation techniques:
   - Checks if records have a minimum number of fields
   - Skips empty values or specific sentinel values (like "9999")
   - Uses try-catch blocks to handle number format exceptions
   - Increments counters for parsing errors to track data quality issues
   - This ensures that only valid data points contribute to the averages, improving result accuracy.
3. Why is the reducer using a combiner pattern important in computing averages over large datasets? Answer: Using the same reducer as a combiner presents a challenge for computing averages because:
   - A true average requires the total sum and count of values
   - When using a combiner, we're computing "averages of averages" which is mathematically incorrect
   - A better approach would be to use a custom writable to track both sum and count separately
   - Another approach is to use the combiner only for sum calculation and track counts separately
4. What modifications would you make to analyze seasonal weather patterns? Answer: To analyze seasonal patterns, I would:
   - Extract month or season information from the timestamp in each record
   - Modify the mapper to emit composite keys containing both the weather attribute and season
   - The output would provide averages by season, enabling comparison across different times of year
   - Example composite key: "temperature_summer", "temperature_winter", etc.
5. How would you extend this application to find extreme weather events like temperature spikes or strong winds? Answer: To identify extreme weather events, I would:
   - Implement a separate job or mapper that identifies values exceeding certain thresholds
   - Use a secondary sort to find the top N extreme values
   - Group extremes by location and time to identify weather events

- Track frequency and duration of extreme conditions
- Store location and timestamp information alongside the extreme values

6. What are the limitations of using MapReduce for time-series weather data analysis? Answer: Limitations include:
   - Difficulty implementing complex time-series algorithms that require state preservation
   - Challenges in expressing sequential or sliding window operations
   - Performance overhead for iterative analyses
   - Limited support for real-time processing
   - For these cases, platforms like Spark Streaming or specialized time-series databases might be more appropriate
7. How would you modify this code to analyze weather correlation, such as the relationship between temperature and humidity? Answer: To analyze correlations:
   - Modify the mapper to emit location or time period as the key
   - Create a custom writable to store multiple weather attributes (temperature, humidity) together
   - Implement a reducer that calculates correlation coefficients
   - Calculate covariance and standard deviations in the reducer
   - Output correlation metrics between pairs of attributes
8. Explain how you would visualize the results of this weather data analysis. Answer: Visualization options include:
   - Generate time-series charts showing trends of temperature, dew point, and wind speed
   - Create heat maps showing spatial distribution of weather patterns
   - Build comparative visualizations showing seasonal variations
   - Develop anomaly highlighting for extreme weather events
   - Tools like Tableau, Power BI, or libraries like D3.js could be used to create these visualizations from the MapReduce output
9. How would you scale this application to handle global weather data with billions of records? Answer: Scaling strategies include:
   - Increase the number of nodes in the Hadoop cluster
   - Optimize input split size for the specific data characteristics
   - Use data compression for storage and transfer efficiency
   - Implement data partitioning by region or time period
   - Consider pre-filtering irrelevant data
   - Use sampling techniques for preliminary analysis
   - Add more reducers and optimize the partition function

10. What additional weather metrics could you analyze, and how would you incorporate them? Answer: Additional metrics could include:
    - Humidity and pressure: Add new attributes in the mapper
    - Precipitation levels: Track rainfall amounts and patterns
    - Cloud cover: Analyze relationship with temperature
    - Air quality metrics: Correlate with weather conditions
    - Storm indicators: Track frequency and intensity of storms
    - Each new metric would require adding parsing logic in the mapper and potentially new custom writables to store combined metrics efficiently

==
==========

TOPIC 4: APACHE SPARK PROGRAMMING WITH SCALA

Sample Code Implementation:

This example demonstrates a simple Spark application written in Scala for processing data.

```
import org.apache.spark.{SparkConf, SparkContext}

object SparkWordCount {
def main(args: Array[String]): Unit = {
// Create a Spark configuration and context
val conf = new SparkConf().setAppName("Spark Word Count").setMaster("local[*]")
val sc = new SparkContext(conf)
```

```
  // Set log level to reduce verbosity
  sc.setLogLevel("WARN")

  // Check if input and output paths are provided
  if (args.length < 2) {
    println("Usage: SparkWordCount <input_path> <output_path>")
    System.exit(1)
  }

  val inputPath = args(0)
  val outputPath = args(1)

  // Read input text file and create an RDD
  val textFile = sc.textFile(inputPath)

  // Split text into words, convert to lowercase, and remove empty words
  val words = textFile.flatMap(line => line.toLowerCase().split("\\W+"))
                      .filter(word => word.length > 0)

  // Count occurrences of each word
  val wordCounts = words.map(word => (word, 1))
                        .reduceByKey(_ + _)

  // Sort by word count in descending order
  val sortedWordCounts = wordCounts.sortBy(_._2, ascending = false)

  // Save the result to the output path
  sortedWordCounts.saveAsTextFile(outputPath)

  // Print the top 10 most frequent words
  println("Top 10 Words:")
  sortedWordCounts.take(10).foreach(println)

  // Stop the SparkContext
  sc.stop()

  println("Word count completed successfully!")
```

}
}

LIKELY VIVA QUESTIONS ON APACHE SPARK WITH SCALA

1. What are the key advantages of Apache Spark over traditional MapReduce? Answer: Key advantages include:
    - In-memory processing that makes Spark up to 100x faster than MapReduce

- Rich API with support for multiple programming languages (Java, Scala, Python, R)
- Unified framework for batch processing, interactive queries, streaming, machine learning, and graph processing
- Lazy evaluation that optimizes the execution plan
- Fault tolerance without relying on HDFS replication
- Significantly less code required for the same tasks

2. Explain the difference between an RDD, DataFrame, and Dataset in Spark. Answer:
    - RDD (Resilient Distributed Dataset): Low-level, fundamental data structure in Spark; distributed collection of objects with no schema
    - DataFrame: Distributed collection of data organized into named columns, similar to database tables; has schema information
    - Dataset: Strongly-typed collection of domain-specific objects that can be mapped to a relational schema; combines RDD's type safety with DataFrame's optimization
    - DataFrames and Datasets use Catalyst optimizer for better performance compared to RDDs
3. Explain the concept of transformations and actions in Spark. Identify them in your code. Answer:
    - Transformations: Operations that create a new RDD from an existing one but are lazily evaluated (not executed immediately)
        - In the code: flatMap(), filter(), map(), reduceByKey(), sortBy()
    - Actions: Operations that return a value to the driver program or write data to storage; trigger execution of transformations
        - In the code: saveAsTextFile(), take(), foreach()
    - Spark builds a DAG of operations and only executes when an action is called, enabling optimization
4. What is the role of SparkContext in a Spark application? Answer: SparkContext is the entry point for any Spark functionality. It:
    - Connects the application to a Spark cluster
    - Creates and manages RDDs, accumulators, and broadcast variables
    - Configures Spark execution parameters
    - Tracks all executors and other resources
    - Provides access to Spark services and resources
    - In Spark 2.0+, SparkContext is accessible through SparkSession
5. Explain the execution flow of this Spark word count application. Answer: The execution flow is:
    1. Create SparkConf and SparkContext
    2. Load text file from input path into an RDD
    3. Split text into words using flatMap (transformation)

4. Convert words to lowercase and filter empty words (transformations)
5. Create word-count pairs using map (transformation)
6. Aggregate counts using reduceByKey (transformation)
7. Sort by count in descending order (transformation)
8. Save results to output path (action)
9. Print top 10 words (action)
10. Stop SparkContext to release resources

6. What would you change in this code to run it on a production Spark cluster instead of local mode? Answer: To run on a production cluster, I would:
   - Remove the setMaster("local[*]") call or replace it with the cluster manager URL
   - Package the application as a JAR file
   - Submit the application using spark-submit with appropriate resource configurations
   - Specify the cluster manager (YARN, Mesos, Kubernetes) in the submission command
   - Add appropriate error handling and logging for production use
7. How would you implement a Spark application to process structured data like CSV files? Answer: For structured data processing:
   - Use SparkSession instead of SparkContext
   - Load CSV files using DataFrameReader: spark.read.option("header", "true").csv(path)
   - Define schema either inferred or explicitly using StructType and StructField
   - Use DataFrame operations like select(), filter(), groupBy(), and join()
   - For complex operations, register temporary views and use SQL
   - Process using DataFrame API for better performance than RDDs
8. Explain how Spark achieves fault tolerance for RDDs. Answer: Spark achieves fault tolerance through:
   - Lineage graphs: Tracking the sequence of transformations used to build each RDD
   - If a partition is lost, Spark can rebuild it by recomputing the operations from the lineage
   - Immutability of RDDs prevents corruption of in-memory data
   - Optional checkpointing for long lineage chains
   - Replication of data across nodes when necessary
   - This approach is more efficient than Hadoop's replication-based approach
9. How would you optimize a Spark application that needs to perform a join between two large datasets? Answer: Optimization strategies include:
   - Broadcast smaller datasets using broadcast joins
   - Partition data properly using partitionBy() before join
   - Filter datasets before joining to reduce data size

- Use appropriate join type (inner, outer, left, right) based on requirements
- Consider using DataFrame API for Catalyst optimizer benefits
- Cache frequently used datasets with persist() or cache()
- Use join hint API in Spark SQL for complex joins

10. Compare RDD operations like map, flatMap, and filter. When would you use each one? Answer:
    - map: One-to-one transformation that applies a function to each element and returns exactly one result per element. Use when each input maps to exactly one output of possibly different type.
    - flatMap: One-to-many transformation that returns zero or more elements for each input element. Use when an input can produce a variable number of outputs, like tokenizing text.
    - filter: Selectively keeps elements that satisfy a predicate function. Use when you need to exclude elements based on a condition.
    - In the word count example, flatMap splits lines into words (one-to-many), filter removes empty words, and map transforms words to key-value pairs.
11. How would you handle streaming data processing in Spark? Answer: For streaming data processing:
    - Use Spark Structured Streaming# Big Data Analytics - Viva Preparation Guide

Most Probable Questions for MapReduce and Apache Spark

Topic 1: WordCount Application using Hadoop MapReduce

Sample Code Implementation

The WordCount application is considered the "Hello World" of MapReduce. It counts occurrences of each word in a given input set.

WordCount.java

```
java
```

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

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;

public class WordCount {

  // Mapper Class
  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().toLowerCase());
        context.write(word, one);
      }

### Likely Viva Questions on Apache Spark with Scala

1. **What are the key advantages of Apache Spark over traditional MapReduce?**
   - *Answer*: Key advantages include:
     - In-memory processing that makes Spark up to 100x faster than MapReduce
     - Rich API with support for multiple programming languages (Java, Scala, Python, R)
     - Unified framework for batch processing, interactive queries, streaming, machine learnin
     - Lazy evaluation that optimizes the execution plan
     - Fault tolerance without relying on HDFS replication
     - Significantly less code required for the same tasks

2. **Explain the difference between an RDD, DataFrame, and Dataset in Spark.**
   - *Answer*:
     - **RDD (Resilient Distributed Dataset)**: Low-level, fundamental data structure in Spark;
     - **DataFrame**: Distributed collection of data organized into named columns, similar to c
     - **Dataset**: Strongly-typed collection of domain-specific objects that can be mapped to
    - DataFrames and Datasets use Catalyst optimizer for better performance compared to RDDs

3. **Explain the concept of transformations and actions in Spark. Identify them in your code.**
   - *Answer*:
    - **Transformations**: Operations that create a new RDD from an existing one but are lazil
      - In the code: `flatMap()`, `filter()`, `map()`, `reduceByKey()`, `sortBy()`
    - **Actions**: Operations that return a value to the driver program or write data to stora
      - In the code: `saveAsTextFile()`, `take()`, `foreach()`
    - Spark builds a DAG of operations and only executes when an action is called, enabling op

4. **What is the role of SparkContext in a Spark application?**
   - *Answer*: SparkContext is the entry point for any Spark functionality. It:
    - Connects the application to a Spark cluster
    - Creates and manages RDDs, accumulators, and broadcast variables
    - Configures Spark execution parameters
    - Tracks all executors and other resources
    - Provides access to Spark services and resources
    - In Spark 2.0+, SparkContext is accessible through SparkSession

5. **Explain the execution flow of this Spark word count application.**
   - *Answer*: The execution flow is:
    1. Create SparkConf and SparkContext
    2. Load text file from input path into an RDD
    3. Split text into words using flatMap (transformation)
    4. Convert words to lowercase and filter empty words (transformations)
    5. Create word-count pairs using map (transformation)
    6. Aggregate counts using reduceByKey (transformation)
    7. Sort by count in descending order (transformation)
    8. Save results to output path (action)
    9. Print top 10 words (action)
    10. Stop SparkContext to release resources

6. **What would you change in this code to run it on a production Spark cluster instead of loca
   - *Answer*: To run on

### Likely Viva Questions on Weather Data Analysis

1. **Explain the purpose of this weather data analysis application and what insights it provide
   - *Answer*: This MapReduce application processes weather data to calculate average temperatu

2. **How does your application handle missing or invalid data in the weather dataset?**
   - *Answer*: The application employs several validation techniques:
    - Checks if records have a minimum number of fields
    - Skips empty values or specific sentinel values (like "9999")
    - Uses try-catch blocks to handle number format exceptions
    - Increments counters for parsing errors to track data quality issues
    - This ensures that only valid data points contribute to the averages, improving result ac
```

```
3. **Why is the reducer using a combiner pattern important in computing averages over large dat
   - *Answer*: Using the same reducer as a combiner presents a challenge for computing averages
     - A true average requires the total sum and count of values
     - When using a combiner, we're computing "averages of averages" which is mathematically ir
     - A better approach would be to use a custom writable to track both sum and count separate
     - Another approach is to use the combiner only for sum calculation and track counts separa

4. **What modifications would you make to analyze seasonal weather patterns?**
   - *Answer*: To analyze seasonal patterns, I would:
     - Extract month or season information from the timestamp in each record
     - Modify the mapper to emit composite keys containing both the weather attribute and seasc
     - The output would provide averages by season, enabling comparison across different times
     - Example composite key: "temperature_summer", "temperature_winter", etc.

5. **How would you extend this application to find extreme weather events like temperature spik
   - *Answer*: To identify extreme weather events, I would:
     - Implement a separate job or mapper that identifies values exceeding certain thresholds
     - Use a secondary sort to find the top N extreme values
     - Group extremes by location and time to identify weather events
     - Track frequency and duration of extreme conditions
     - Store location and timestamp information alongside the extreme values

6. **What are the limitations of using MapReduce for time-series weather data analysis?**
   - *Answer*: Limitations include:
     - Difficulty implementing complex time-series algorithms that require state preservation
     - Challenges in expressing sequential or sliding window operations
     - Performance overhead for iterative analyses
     - Limited support for real-time processing
     - For these cases, platforms like Spark Streaming or specialized time-series databases mig

7. **How would you modify this code to analyze weather correlation, such as the relationship be
   - *Answer*: To analyze correlations:
     - Modify the mapper to emit location or time period as the key
     - Create a custom writable to store multiple weather attributes (temperature, humidity) tc
     - Implement a reducer that calculates correlation coefficients
     - Calculate covariance and standard deviations in the reducer
     - Output correlation metrics between pairs of attributes

8. **Explain how you would visualize the results of this weather data analysis.**
   - *Answer*: Visualization options include:
     - Generate time-series charts showing trends of temperature, dew point, and wind speed
     - Create heat maps showing spatial distribution of weather patterns
     - Build comparative visualizations showing seasonal variations
     - Develop anomaly highlighting for extreme weather events
     - Tools like Tableau, Power BI, or libraries like D3.js could be used to create these visu
```

9. **How would you scale this application to handle global weather data with billions of recor
   - *Answer*: Scaling strategies include:
     - Increase the number of nodes in the Hadoop cluster
     - Optimize input split size for the specific data characteristics
     - Use data compression for storage and transfer efficiency
     - Implement data partitioning by region or time period
     - Consider pre-filtering irrelevant data
     - Use sampling techniques for preliminary analysis
     - Add more reducers and optimize the partition function

10. **What additional weather metrics could you analyze, and how would you incorporate them?**
    - *Answer*: Additional metrics could include:
      - Humidity and pressure: Add new attributes in the mapper
      - Precipitation levels: Track rainfall amounts and patterns
      - Cloud cover: Analyze relationship with temperature
      - Air quality metrics: Correlate with weather conditions
      - Storm indicators: Track frequency and intensity of storms
      - Each new metric would require adding parsing logic in the mapper and potentially new cu

Topic 4: Apache Spark Programming with Scala

Sample Code Implementation

This example demonstrates a simple Spark application written in Scala for processing data.

SparkWordCount.scala

```scala
import org.apache.spark.{SparkConf, SparkContext}

object SparkWordCount {
  def main(args: Array[String]): Unit = {
    // Create a Spark configuration and context
    val conf = new SparkConf().setAppName("Spark Word Count").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Set log level to reduce verbosity
    sc.setLogLevel("WARN")

    // Check if input and output paths are provided
    if (args.length < 2) {
      println("Usage: SparkWordCount <input_path> <output_path>")
      System.exit(1)
    }

    val inputPath = args(0)
    val outputPath = args(1)
    // Read input text file and create an RDD
    val textFile = sc.textFile(inputPath)

    // Split text into words, convert to lowercase, and remove empty words
    val words = textFile.flatMap(line => line.toLowerCase().split("\\W+"))
                        .filter(word => word.length > 0)

    // Count occurrences of each word
    val wordCounts = words.map(word => (word, 1))
                          .reduceByKey(_ + _)

    // Sort by word count in descending order
    val sortedWordCounts = wordCounts.sortBy(_._2, ascending = false)

    // Save the result to the output path
    sortedWordCounts.saveAsTextFile(outputPath)

    // Print the top 10 most frequent words
    println("Top 10 Words:")
    sortedWordCounts.take(10).foreach(println)

    // Stop the SparkContext
    sc.stop()

    println("Word count completed successfully!")
  }
}

### Likely Viva Questions on Log File Processing

1. **What is the purpose of processing log files using MapReduce?**
   - *Answer*: Processing log files with MapReduce allows for efficient analysis of large volum

2. **Explain the regular expression pattern used in your log analyzer. What log format is it de
   - *Answer*: The regular expression pattern is designed to parse the Common Log Format (CLF)

3. **How does your LogMapper extract meaningful information from log entries?**
   - *Answer*: The LogMapper uses regex pattern matching to extract key components from each lo

4. **What are the advantages of using MapReduce for log analysis compared to traditional script
   - *Answer*:
     - Scalability: Handles terabytes or petabytes of log data across clusters
     - Parallelism: Processes logs in parallel across many machines
     - Fault tolerance: Continues processing despite node failures
     - Data locality: Minimizes network transfer by moving computation to data
     - Built-in aggregation: Efficiently counts and summarizes metrics
```

~~~
5. **How would you modify this code to identify potential security threats like brute force att
   - *Answer*: To identify potential brute force attacks, I would:
     - Track login failure attempts by IP address
     - Use a time window to identify high-frequency failures
     - Implement a secondary MapReduce job that takes the output from the first job and identif
     - Could use a custom writable class to track timestamp information along with counts

6. **Explain how you would enhance this log analyzer to perform time-based analysis.**
   - *Answer*: I would extract and parse the timestamp field into a structured format (e.g., hc
   ```java
   // Extract and parse timestamp
   String timestamp = matcher.group(4);
   SimpleDateFormat inputFormat = new SimpleDateFormat("dd/MMM/yyyy:HH:mm:ss Z");
   Date date = inputFormat.parse(timestamp);

   // Extract hour for hourly analysis
   Calendar cal = Calendar.getInstance();
   cal.setTime(date);
   int hour = cal.get(Calendar.HOUR_OF_DAY);

   // Emit hourly metrics
   outputKey.set("HOUR:" + hour + ":STATUS:" + statusCode);
   context.write(outputKey, one);
~~~

7. **How would you implement a secondary sort in MapReduce to sort log entries by both timestamp and status code?**
   - *Answer*: I would implement a composite key class that holds both timestamp and status code, along with a custom partitioner and comparator:
     1. Create a composite key class implementing WritableComparable
     2. Implement custom partitioning to ensure entries with the same timestamp go to the same reducer
     3. Implement a custom sort comparator that sorts by timestamp first, then by status code
     4. The natural sort order would be used in the reducer to process entries in the desired order

8. **What challenges might you face when processing log files from multiple sources with different formats?**
   - *Answer*: Challenges include:
     - Handling different log formats requires multiple regex patterns or parsers
     - Time synchronization issues between different servers
     - Inconsistent field naming or meaning across sources
     - Varying severity levels and error classifications

    - Solutions include creating format-specific mappers or a flexible mapper that detects and adapts to different formats

9. **How would you optimize this log analyzer for extremely large log datasets?**
   - *Answer*:
     - Use log file compression to reduce storage and I/O
     - Implement custom input formats to efficiently split compressed logs
     - Filter irrelevant log entries early in the mapper
     - Use counters instead of emitting data for common metrics
     - Implement sampling techniques for trend analysis rather than processing all entries
     - Use the combiner aggressively to reduce network transfer
     - Consider using a bloom filter for common patterns

10. **How can you extend this MapReduce job to generate visualizations or reports from the log analysis?**
    - *Answer*: The MapReduce job would output structured data that can be:
      - Processed by visualization tools like Tableau or Power BI
      - Loaded into a database or data warehouse for reporting
      - Fed into another MapReduce job that generates HTML reports
      - Used as input for a charting library in a web application
      - Alternatively, we could write a custom reducer that formats the results directly into HTML or JSON format

Topic 3: Weather Data Analysis using MapReduce

Sample Code Implementation

This application processes weather data to calculate average temperature, dew point, and wind speed.

WeatherDataAnalyzer.java

java

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
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.output.FileOutputFormat;

public class WeatherDataAnalyzer {

  public static class WeatherMapper extends Mapper<LongWritable, Text, Text, DoubleWritable> {

    private Text weatherAttribute = new Text();
    private DoubleWritable value = new DoubleWritable();

    @Override
    public void map(LongWritable key, Text input, Context context) throws IOException, Interrup
      String line = input.toString();
      String[] fields = line.split(",");

      // Ensure we have a valid record with enough fields
      if (fields.length < 5) {
        return; // Skip invalid records
      }

      try {
        // Parse the temperature field (assuming position 2)
        // Skip if temperature is missing or invalid
        if (!fields[2].trim().equals("") && !fields[2].trim().equals("9999")) {
          double temperature = Double.parseDouble(fields[2].trim());
          weatherAttribute.set("temperature");
          value.set(temperature);
          context.write(weatherAttribute, value);
        }

        // Parse the dew point field (assuming position 3)
        // Skip if dew point is missing or invalid
        if (!fields[3].trim().equals("") && !fields[3].trim().equals("9999")) {
          double dewPoint = Double.parseDouble(fields[3].trim());
          weatherAttribute.set("dewpoint");
          value.set(dewPoint);
          context.write(weatherAttribute, value);
      }

      // Parse the wind speed field (assuming position 4)
      // Skip if wind speed is missing or invalid
      if (!fields[4].trim().equals("") && !fields[4].trim().equals("9999")) {
        double windSpeed = Double.parseDouble(fields[4].trim());
        weatherAttribute.set("windspeed");
        value.set(windSpeed);
        context.write(weatherAttribute, value);
      }
    } catch (NumberFormatException e) {
      // Skip record with parsing errors
      context.getCounter("WeatherData", "ParseErrors").increment(1);
    }
  }
}

public static class AverageReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable
  private DoubleWritable result = new DoubleWritable();

  @Override
  public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
      throws IOException, InterruptedException {
    double sum = 0;
    long count = 0;

    // Calculate the sum and count for computing average
    for (DoubleWritable val : values) {
      sum += val.get();
      count++;
    }

    // Calculate average
    double average = sum / count;
    result.set(average);

    // Write the result
    context.write(key, result);

    // Write count as a counter for reference
    context.getCounter("WeatherData", key.toString() + "Count").setValue(count);
  }
}

public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "weather data analyzer");
```

~~~
    job.setJarByClass(WeatherDataAnalyzer.class);
    job.setMapperClass(WeatherMapper.class);
    job.setCombinerClass(AverageReducer.class);
    job.setReducerClass(AverageReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}

### Likely Viva Questions on WordCount MapReduce Application

1. **What is the purpose of the WordCount application in Hadoop?**
   - *Answer*: WordCount is a basic MapReduce application that counts the occurrence of each wc

2. **Explain the role of Mapper and Reducer in your WordCount implementation.**
   - *Answer*: The `TokenizerMapper` class splits input text into words and emits a key-value p

3. **What are IntWritable and Text classes in Hadoop, and why are they used instead of standarc
   - *Answer*: `IntWritable` and `Text` are Hadoop's serializable counterparts to Java's `Integ

4. **Explain the importance of the Combiner in your WordCount application.**
   - *Answer*: The Combiner acts as a mini-reducer that runs on the mapper's output before data

5. **How would you modify this WordCount application to ignore common words like "the", "and",
   - *Answer*: I would create a set of stop words in the Mapper class and check each tokenized
   ```java
   private Set<String> stopWords = new HashSet<>(Arrays.asList("the", "and", "a", "to", "of", "

   // In map method:
   if (!stopWords.contains(word.toString())) {
       context.write(word, one);
   }
~~~

6. **How would you make the WordCount application case-insensitive?**
   - *Answer*: In the map method, I would convert each token to lowercase before emitting it as a key:

     java

     ```
     word.set(itr.nextToken().toLowerCase());
     ```

7. **If your input data is very large, how does Hadoop ensure efficient processing?**
   - *Answer*: Hadoop processes large data efficiently through:
     - Data locality: Moving computation to data rather than data to computation

    - Data splitting: Breaking large files into blocks and processing them in parallel
    - Distributed processing: Utilizing multiple nodes for parallel execution
    - Fault tolerance: Automatically recovering from node failures
    - Combiners: Performing local aggregations to reduce network transfer

8. **Explain the data flow in this WordCount application.**
   - *Answer*:
     1. Input files are split and distributed to mappers
     2. Mappers tokenize text and emit (word, 1) pairs
     3. Pairs are partitioned, sorted by key, and sent to combiners (if configured)
     4. Combiners perform local aggregation, producing (word, partial_count) pairs
     5. These pairs are shuffled and sorted across the network
     6. Reducers receive all values for each key and sum them
     7. Final (word, total_count) pairs are written to output files

9. **What configuration changes would you make to optimize the WordCount job for very large datasets?**
   - *Answer*:
     - Increase the number of reducers based on data size
     - Enable and configure combiners to reduce network traffic
     - Adjust memory allocation for mappers and reducers
     - Use compression for intermediate data
     - Tune the split size to optimize parallelism
     - Use a custom partitioner if word distribution is skewed

10. **How would you modify this code to count word frequency in a specific file format like JSON or XML?**
    - *Answer*: I would replace the simple `StringTokenizer` with a proper parser for the specific format. For JSON, I might use a library like Jackson to parse the documents and extract relevant text fields before tokenization. Additionally, I would need to modify the mapper to handle the structured format and potentially extract only specific fields for word counting.

Topic 2: Processing Log Files with MapReduce

Sample Code Implementation

This application processes system log files to extract and analyze information such as error counts, request patterns, or system performance metrics.

LogAnalyzer.java

java

```java
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

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;

public class LogAnalyzer {

  // Regular expression pattern for Apache log format
  private static final Pattern LOG_PATTERN = Pattern.compile(
      "^([\\d.]+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\\d{3}) (\\d+) \"([^

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

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

    @Override
    public void map(Object key, Text value, Context context) throws IOException, InterruptedExc
      String logEntry = value.toString();
      Matcher matcher = LOG_PATTERN.matcher(logEntry);

      if (matcher.find()) {
        // Extract IP address
        String ipAddress = matcher.group(1);

        // Extract HTTP status code
        String statusCode = matcher.group(6);

        // Extract request URL from the request field
        String request = matcher.group(5);
        String[] parts = request.split(" ");
        String url = "";
        if (parts.length > 1) {
          url = parts[1];
        }

        // Emit IP address count
        outputKey.set("IP:" + ipAddress);
        context.write(outputKey, one);

        // Emit status code count
        outputKey.set("STATUS:" + statusCode);
        context.write(outputKey, one);

        // Emit URL count
        if (!url.isEmpty()) {
          outputKey.set("URL:" + url);
          context.write(outputKey, one);
        }

        // Check for error status codes (4xx and 5xx)
        int status = Integer.parseInt(statusCode);
        if (status >= 400) {
          outputKey.set("ERROR");
          context.write(outputKey, one);
        }
      }
    }
  }

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

    @Override
    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, "log analyzer");
    job.setJarByClass(LogAnalyzer.class);
    job.setMapperClass(LogMapper.class);
    job.setCombinerClass(LogReducer.class);
    job.setReducerClass(LogReducer.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);
  }
}
    }
  }

  // Reducer Class
  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);
    }
  }

  // Main Method
  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);
  }
}
```