1. **Install Apache Hadoop.**

Prerequisites:

Java Development Kit (JDK): Install the latest version of Oracle JDK or OpenJDK. Ensure that Java is correctly installed and added to the system's PATH environment variable. You can verify Java installation by running java -version in Command Prompt.

Hadoop Binary: Download the appropriate version of Hadoop for Windows from the official Apache Hadoop website (https://hadoop.apache.org/releases.html).

Configure Java:

Set the JAVA_HOME environment variable to point to the JDK installation directory. You can do this by:

Right-clicking on "This PC" or "My Computer" and selecting "Properties."

Clicking on "Advanced system settings" and then "Environment Variables."

Adding a new system variable named JAVA_HOME with the path to the JDK directory (e.g., C:\Program Files\Java\jdk1.8.0_291).

Extract Hadoop:

Extract the downloaded Hadoop binary (e.g., hadoop-X.Y.Z.tar.gz) to a directory of your choice (e.g., C:\hadoop).

Configure Hadoop:

Navigate to the Hadoop installation directory (e.g., C:\hadoop) and edit the configuration files in the etc/hadoop directory:

core-site.xml: Configure Hadoop core settings, such as the filesystem name and default block size.

hdfs-site.xml: Configure Hadoop Distributed File System (HDFS) settings, such as replication factor and namenode data directories.

mapred-site.xml: Configure MapReduce settings.

yarn-site.xml: Configure Yet Another Resource Negotiator (YARN) settings.

Configure Windows Environment Variables:

Add the following environment variables:

HADOOP_HOME: Set it to the Hadoop installation directory (e.g., C:\hadoop).

Update the PATH environment variable to include %HADOOP_HOME%\bin.

Format HDFS:

Open Command Prompt as administrator and navigate to the Hadoop installation directory.

Run the following command to format the Hadoop Distributed File System (HDFS):

```
hadoop namenode -format
```

Start Hadoop Services:

Start the Hadoop daemons by running the following command in Command Prompt:

```
start-dfs.cmd
start-yarn.cmd
```

This script starts all Hadoop daemons, including NameNode, DataNode, ResourceManager, and NodeManager.

Verify Installation:

After starting Hadoop services, you can verify the installation by accessing the Hadoop Web UI. By default, the Hadoop Web UI is available at http://localhost:9870 for HDFS and http://localhost:8088 for YARN.

2. Develop a MapReduce program to calculate the frequency of a given word in a given file.

Here's a MapReduce program written in Java to calculate the frequency of a given word in a given file.

```
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.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 WordFrequency {

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

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

      public void map(LongWritable 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(WordFrequency.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);
    }
}
```

To run this program, you'll need to compile it into a JAR file and then execute it using Hadoop's hadoop command, passing in the input file and output directory as arguments.

hadoop jar WordFrequency.jar WordFrequency input.txt output

Replace WordFrequency with the name of your compiled JAR file, input.txt with the input file containing the text you want to analyze, and output with the directory where you want the results to be stored.

3. Develop a MapReduce program to find the maximum temperature in each year.

To find the maximum temperature in each year using MapReduce, you'll typically have input data in the form of records where each record represents a temperature reading with a

timestamp (including the year) and the corresponding temperature value. You'll then use MapReduce to process these records, grouping them by year and finding the maximum temperature for each year. Below is the program of how you can implement this using Java and Hadoop MapReduce:

```
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.*;
import org.apache.hadoop.mapreduce.*;

public class MaxTemperature {

   // Mapper class
   public static class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
      private static final int MISSING = 9999;

      @Override
      public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
         String line = value.toString();
         String year = line.substring(15, 19);
         int airTemperature;

         if (line.charAt(87) == '+') { // Positive temperature
            airTemperature = Integer.parseInt(line.substring(88, 92));
         } else {
            airTemperature = Integer.parseInt(line.substring(87, 92));
         }

         String quality = line.substring(92, 93);
         if (airTemperature != MISSING && quality.matches("[01459]")) { // Check quality
            context.write(new Text(year), new IntWritable(airTemperature));
         }
      }
   }

   // Reducer class
   public static class MaxTemperatureReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
      @Override
      public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
         int maxTemperature = Integer.MIN_VALUE;
         for (IntWritable value : values) {
            maxTemperature = Math.max(maxTemperature, value.get());
         }
         context.write(key, new IntWritable(maxTemperature));
```

```
    }
  }

  // Driver method
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "max temperature");
    job.setJarByClass(MaxTemperature.class);
    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Here's a brief explanation of the code:

- The MaxTemperatureMapper class reads each line of input data, extracts the year and temperature, and emits them as key-value pairs where the key is the year and the value is the temperature.
- The MaxTemperatureReducer class receives key-value pairs grouped by year and finds the maximum temperature for each year.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your input data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

hadoop jar MaxTemperature.jar input_path output_path

This will run the MapReduce job to find the maximum temperature in each year and store the results in the specified output path.

4. **Develop a MapReduce program to find the grades of student's.**

To develop a MapReduce program to find the grades of students, you typically need input data that includes student information along with their marks or scores. Based on the scores, you can then determine the grades for each student. Below is the program of how you can implement this using Java and Hadoop MapReduce:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class StudentGrades {

    // Mapper class
    public static class GradeMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] tokens = value.toString().split(",");

            // Assuming the input format is: student_name,subject,marks
            if (tokens.length == 3) {
                String studentName = tokens[0];
                String subject = tokens[1];
                int marks = Integer.parseInt(tokens[2]);
                String grade = calculateGrade(marks); // Calculate grade based on marks
                context.write(new Text(studentName), new Text(subject + ":" + grade));
            }
        }

        // Method to calculate grade based on marks
        private String calculateGrade(int marks) {
            if (marks >= 90) {
                return "A+";
            } else if (marks >= 80) {
                return "A";
            } else if (marks >= 70) {
                return "B";
            } else if (marks >= 60) {
                return "C";
            } else if (marks >= 50) {
                return "D";
            } else {
                return "F";
            }
        }
    }

    // Reducer class
```

```
    public static class GradeReducer extends Reducer<Text, Text, Text, Text> {
        @Override
        public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            StringBuilder result = new StringBuilder();
            for (Text value : values) {
                result.append(value.toString()).append(", ");
            }
            context.write(key, new Text(result.toString()));
        }
    }

    // Driver method
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "student grades");
        job.setJarByClass(StudentGrades.class);
        job.setMapperClass(GradeMapper.class);
        job.setReducerClass(GradeReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Here's a brief explanation of the code:

- The GradeMapper class reads each line of input data, splits it into tokens (assuming CSV format), extracts student name, subject, and marks. It then calculates the grade based on marks and emits key-value pairs where the key is the student name and the value is the subject and grade separated by ":".
- The GradeReducer class receives key-value pairs grouped by student name and concatenates the subjects and grades for each student.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your input data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

hadoop jar StudentGrades.jar input_path output_path

This will run the MapReduce job to find the grades of students and store the results in the specified output path.

5. **Develop a MapReduce program to implement Matrix Multiplication.**

Implementing Matrix Multiplication using MapReduce involves breaking down the multiplication process into smaller tasks that can be executed in parallel. Below is the program of how you can implement Matrix Multiplication using Java and Hadoop MapReduce:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class MatrixMultiplication {

    // Mapper class
    public static class MatrixMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] tokens = value.toString().split(",");
            String matrixName = tokens[0];
            int row = Integer.parseInt(tokens[1]);
            int col = Integer.parseInt(tokens[2]);
            int val = Integer.parseInt(tokens[3]);

            if (matrixName.equals("A")) {
                for (int k = 0; k < context.getConfiguration().getInt("matrixK", 1); k++) {
                    context.write(new Text(row + "," + k), new Text(matrixName + "," + col + "," +
val));
                }
            } else { // Matrix B
                for (int i = 0; i < context.getConfiguration().getInt("matrixK", 1); i++) {
                    context.write(new Text(i + "," + col), new Text(matrixName + "," + row + "," +
val));
                }
            }
        }
    }

    // Reducer class
    public static class MultiplicationReducer extends Reducer<Text, Text, Text, IntWritable>
{
        @Override
        public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            int[] vectorA = new int[context.getConfiguration().getInt("matrixK", 1)];
            int[] vectorB = new int[context.getConfiguration().getInt("matrixK", 1)];
            for (Text value : values) {
                String[] tokens = value.toString().split(",");
                String matrixName = tokens[0];
                int index = Integer.parseInt(tokens[1]);
```

```
            int val = Integer.parseInt(tokens[2]);

            if (matrixName.equals("A")) {
                vectorA[index] = val;
            } else { // Matrix B
                vectorB[index] = val;
            }
        }

        int result = 0;
        for (int i = 0; i < vectorA.length; i++) {
            result += vectorA[i] * vectorB[i];
        }
        context.write(key, new IntWritable(result));
    }
}

// Driver method
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.setInt("matrixK", Integer.parseInt(args[2])); // Set matrixK (number of columns in
A / number of rows in B)
    Job job = Job.getInstance(conf, "matrix multiplication");
    job.setJarByClass(MatrixMultiplication.class);
    job.setMapperClass(MatrixMapper.class);
    job.setReducerClass(MultiplicationReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0])); // Input path for matrix A
    FileInputFormat.addInputPath(job, new Path(args[1])); // Input path for matrix B
    FileOutputFormat.setOutputPath(job, new Path(args[3])); // Output path
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Here's a brief explanation of the code:

- The MatrixMapper class reads each line of input data, splits it into tokens, and emits intermediate key-value pairs where the key is the row-column index of the result matrix and the value is the matrix element from either A or B along with its row/column index.
- The MultiplicationReducer class receives key-value pairs grouped by the row-column index and calculates the dot product of the corresponding rows of A and columns of B.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your input matrices stored in HDFS (matrix A and matrix B).
3. Execute the JAR file using Hadoop, specifying input and output paths, and the number of columns in matrix A / number of rows in matrix B (matrixK).

hadoop jar MatrixMultiplication.jar input_matrix_A input_matrix_B matrixK output_path

This will run the MapReduce job to perform matrix multiplication and store the result in the specified output path.

6. **Develop a MapReduce to find the maximum electrical consumption in each year given Electrical consumption for each month in each year.**

To find the maximum electrical consumption in each year given the electrical consumption for each month in each year, we can use a MapReduce approach. In this approach, we will emit key-value pairs where the key is the year, and the value is the electrical consumption for that year. Then, in the reducer, we will find the maximum consumption for each year.

Below is the implementation of this MapReduce program:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class MaxElectricalConsumption {

    // Mapper class
    public static class MaxConsumptionMapper extends Mapper<LongWritable, Text,
IntWritable, IntWritable> {
        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] tokens = value.toString().split(",");
            if (tokens.length == 3) {
                int year = Integer.parseInt(tokens[0]);
                int month = Integer.parseInt(tokens[1]);
                int consumption = Integer.parseInt(tokens[2]);
                context.write(new IntWritable(year), new IntWritable(consumption));
            }
        }
    }

    // Reducer class
    public static class MaxConsumptionReducer extends Reducer<IntWritable, IntWritable,
IntWritable, IntWritable> {
        @Override
        public void reduce(IntWritable key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
            int maxConsumption = Integer.MIN_VALUE;
            for (IntWritable value : values) {
                maxConsumption = Math.max(maxConsumption, value.get());
            }
            context.write(key, new IntWritable(maxConsumption));
        }
    }
```

```
    // Driver method
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "max electrical consumption");
        job.setJarByClass(MaxElectricalConsumption.class);
        job.setMapperClass(MaxConsumptionMapper.class);
        job.setReducerClass(MaxConsumptionReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0])); // Input path
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Here's a brief explanation of the code:

- The MaxConsumptionMapper class reads each line of input data, splits it into tokens, and emits key-value pairs where the key is the year and the value is the electrical consumption for that year.
- The MaxConsumptionReducer class receives key-value pairs grouped by the year and finds the maximum consumption for each year.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your input data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

```
hadoop jar MaxElectricalConsumption.jar input_path output_path
```

This will run the MapReduce job to find the maximum electrical consumption in each year and store the results in the specified output path.

7. **Develop a MapReduce to analyze weather data set and print whether the day is shinny or cool day.**

To analyze weather data and determine whether a day is sunny or cool, we need to define some criteria or conditions based on the weather data provided. For simplicity, let's assume that a day is considered sunny if the temperature is above a certain threshold, and it's considered cool otherwise.

Below is the implementation of the MapReduce program to analyze weather data and determine whether each day is sunny or cool:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class WeatherAnalyzer {

    // Mapper class
    public static class WeatherMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] tokens = value.toString().split(",");
            if (tokens.length == 3) {
                String date = tokens[0];
                int temperature = Integer.parseInt(tokens[1]);
                String weatherCondition = tokens[2];
                String result = (temperature > 70) ? "sunny" : "cool"; // Define criteria for sunny or
cool day
                context.write(new Text(date), new Text(result));
            }
        }
    }

    // Reducer class
    public static class WeatherReducer extends Reducer<Text, Text, Text, Text> {
        @Override
        public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            for (Text value : values) {
                context.write(key, value);
            }
        }
    }

    // Driver method
```

```
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "weather analyzer");
        job.setJarByClass(WeatherAnalyzer.class);
        job.setMapperClass(WeatherMapper.class);
        job.setReducerClass(WeatherReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0])); // Input path
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

In this implementation:

- The WeatherMapper class reads each line of weather data, splits it into tokens, and emits key-value pairs where the key is the date and the value is either "sunny" or "cool" based on the temperature threshold.
- The WeatherReducer class simply passes through the key-value pairs emitted by the mapper.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your weather data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

hadoop jar WeatherAnalyzer.jar input_weather_data output_path

This will run the MapReduce job to analyze the weather data and determine whether each day is sunny or cool, and store the results in the specified output path.

8. **Develop a MapReduce program to find the number of products sold in each country by considering sales data containing fields like: Transaction_Date, Product, Price, Payment_Type, Name, City, State, Country, Account_Created, Last_Login, Latitude, Longitude.**

To find the number of products sold in each country from sales data using MapReduce, we'll use a Mapper to emit key-value pairs where the key is the country and the value is 1 for each product sold in that country. Then, in the Reducer, we'll sum up these values for each country to get the total number of products sold in each country.

Here's the implementation of the MapReduce program:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class ProductSalesAnalyzer {

    // Mapper class
    public static class SalesMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        private static final int COUNTRY_INDEX = 7; // Index of the country field

        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] fields = value.toString().split(",");
            if (fields.length >= COUNTRY_INDEX + 1) {
                String country = fields[COUNTRY_INDEX].trim();
                context.write(new Text(country), new IntWritable(1));
            }
        }
    }

    // Reducer class
    public static class SalesReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
            int sum = 0;
            for (IntWritable value : values) {
                sum += value.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
```

```
    // Driver method
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "product sales analyzer");
        job.setJarByClass(ProductSalesAnalyzer.class);
        job.setMapperClass(SalesMapper.class);
        job.setReducerClass(SalesReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0])); // Input path
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Explanation:

- The SalesMapper class reads each line of sales data, splits it into fields, and extracts the country field. It then emits key-value pairs where the key is the country and the value is 1 to indicate the sale of a product in that country.
- The SalesReducer class receives key-value pairs grouped by country and sums up the values to get the total number of products sold in each country.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your sales data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

hadoop jar ProductSalesAnalyzer.jar input_sales_data output_path

This will run the MapReduce job to analyze the sales data and find the number of products sold in each country, storing the results in the specified output path.

9. **Develop a MapReduce program to find the tags associated with each movie by analyzing movie lens data.**

To find the tags associated with each movie by analyzing MovieLens data using MapReduce, we'll use a Mapper to extract movie ID and tags, then in the Reducer, we'll aggregate the tags associated with each movie.

Here's the implementation of the MapReduce program:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class MovieTagsAnalyzer {

    // Mapper class
    public static class TagsMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] fields = value.toString().split("::");
            if (fields.length >= 4) {
                String movieId = fields[1].trim();
                String tags = fields[2].trim();
                context.write(new Text(movieId), new Text(tags));
            }
        }
    }

    // Reducer class
    public static class TagsReducer extends Reducer<Text, Text, Text, Text> {
        @Override
        public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            StringBuilder tagsBuilder = new StringBuilder();
            for (Text value : values) {
                tagsBuilder.append(value.toString()).append(", ");
            }
            String allTags = tagsBuilder.toString().trim();
            context.write(key, new Text(allTags));
        }
    }
```

```
// Driver method
   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf, "movie tags analyzer");
      job.setJarByClass(MovieTagsAnalyzer.class);
      job.setMapperClass(TagsMapper.class);
      job.setReducerClass(TagsReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(Text.class);
      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(Text.class);
      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TextOutputFormat.class);
      FileInputFormat.addInputPath(job, new Path(args[0])); // Input path
      FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path
      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

Explanation:

- The TagsMapper class reads each line of MovieLens data, splits it into fields, and extracts movie ID and tags. It then emits key-value pairs where the key is the movie ID and the value is the tags associated with that movie.
- The TagsReducer class receives key-value pairs grouped by movie ID and aggregates the tags associated with each movie.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your MovieLens data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

```
hadoop jar MovieTagsAnalyzer.jar input_movie_lens_data output_path
```

This will run the MapReduce job to analyze the MovieLens data and find the tags associated with each movie, storing the results in the specified output path.

10. **Develop a MapReduce program to find the frequency of books published each year and find in which year maximum number of books were published using the following data: Title, Author Published, year, Author, country, Language, No of pages**

To find the frequency of books published each year and determine the year with the maximum number of books published, we'll use a MapReduce program. In this program, the Mapper will extract the published year from each book record, and the Reducer will count the occurrences of each year and find the year with the maximum count.

Here's the implementation of the MapReduce program:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class BookPublicationFrequency {

   // Mapper class
   public static class PublicationYearMapper extends Mapper<LongWritable, Text,
IntWritable, IntWritable> {
      private static final int YEAR_INDEX = 2; // Index of the year field

      @Override
      public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
         String[] fields = value.toString().split(",");
         if (fields.length >= YEAR_INDEX + 1) {
            int year = Integer.parseInt(fields[YEAR_INDEX].trim());
            context.write(new IntWritable(year), new IntWritable(1));
         }
      }
   }

   // Reducer class
   public static class PublicationFrequencyReducer extends Reducer<IntWritable,
IntWritable, IntWritable, IntWritable> {
      @Override
      public void reduce(IntWritable key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
         int count = 0;
         for (IntWritable value : values) {
            count += value.get();
         }
         context.write(key, new IntWritable(count));
```

```
        }
    }

    // Driver method
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "book publication frequency");
        job.setJarByClass(BookPublicationFrequency.class);
        job.setMapperClass(PublicationYearMapper.class);
        job.setReducerClass(PublicationFrequencyReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0])); // Input path
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Explanation:

- The PublicationYearMapper class reads each line of book data, splits it into fields, and extracts the publication year. It then emits key-value pairs where the key is the publication year and the value is 1 to indicate the publication of a book in that year.
- The PublicationFrequencyReducer class receives key-value pairs grouped by publication year and calculates the total number of books published in each year.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your book data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

hadoop jar BookPublicationFrequency.jar input_book_data output_path

This will run the MapReduce job to analyze the book data and find the frequency of books published each year, storing the results in the specified output path.

11. **Develop a MapReduce program to analyze Uber data set to find the days on which each basement has more trips using the following dataset. The Uber dataset consists of four columns they are: dispatching_base_number, date, active_vehicles, trips**

To analyze the Uber dataset and find the days on which each basement has more trips, we'll use a MapReduce program. In this program, the Mapper will extract the basement (dispatching_base_number) and date from each record, and the Reducer will count the number of trips for each basement on each day.

Here's the implementation of the MapReduce program:

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;

public class UberTripAnalyzer {

    // Mapper class
    public static class TripMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        private static final int BASEMENT_INDEX = 0; // Index of the basement field
        private static final int DATE_INDEX = 1; // Index of the date field
        private static final int TRIPS_INDEX = 3; // Index of the trips field

        @Override
        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            String[] fields = value.toString().split(",");
            if (fields.length >= TRIPS_INDEX + 1) {
                String basement = fields[BASEMENT_INDEX].trim();
                String date = fields[DATE_INDEX].trim();
                int trips = Integer.parseInt(fields[TRIPS_INDEX].trim());
                context.write(new Text(basement + "," + date), new IntWritable(trips));
            }
        }
    }

    // Reducer class
    public static class TripReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
            int totalTrips = 0;
            for (IntWritable value : values) {
                totalTrips += value.get();
```

```
            }
            context.write(key, new IntWritable(totalTrips));
        }
    }

    // Driver method
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "uber trip analyzer");
        job.setJarByClass(UberTripAnalyzer.class);
        job.setMapperClass(TripMapper.class);
        job.setReducerClass(TripReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0])); // Input path
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output path
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Explanation:

- The TripMapper class reads each line of Uber data, splits it into fields, and extracts the basement and date. It then emits key-value pairs where the key is the basement and date combined, and the value is the number of trips.
- The TripReducer class receives key-value pairs grouped by basement and date, and calculates the total number of trips for each basement on each day.
- In the main method, the job configuration is set up, specifying the Mapper, Reducer, input and output formats, and other relevant details.

To use this code:

1. Compile it into a JAR file.
2. Make sure you have your Uber data stored in HDFS.
3. Execute the JAR file using Hadoop, specifying input and output paths.

hadoop jar UberTripAnalyzer.jar input_uber_data output_path

This will run the MapReduce job to analyze the Uber data and find the days on which each basement has more trips, storing the results in the specified output path.