InputFormat

|  |  |
| --- | --- |
| **Objective:** | Write a MapReduce job that uses a custom InputFormat. |
| **Location of Files:** | ~/materials/data/NYSE\_daily\_prices\_F.csv |
| **Successful Outcome:** | The output files will contain the closing prices of stocks, sorted by date. |
| **Before You Begin:** | Open Eclipse in your lab instance. |
| **JAR File** | movingaverage.jar |
| **Eclipse Project** | moving\_average |
| **Exercise directory** | ~/workspace/moving\_average |

1.      Locate the Project

1.1.   Open the **moving\_average** project in Eclipse. You'll notice some compiler errors.

1.2. Open the **Stock** class. You should recognize it this class is basically the same Stock class from the dividends lab, with the addition of a toString method. This class will be the input key to your Mapper.

1.3.   Open the **StockPrices** class. This class will be the input value to your Mapper in this application.

1.4.   View the data: right-click on the file NYSE\_daily\_prices\_F.csvin the data folder and select **Open With Text Editor**. This file is a sample of the input data.

1.5.   Notice the first row is a header row (that needs to be ignored). The other rows contain the stock symbol, date, and various prices of the stock for that day. In this lab, you are going to write a custom InputFormat that reads in this data as a Stock/StockPrices set of key/value pairs and outputs the stock symbol, date, and closing price.

2.      View the PreprocessorMapper

2.1.   Notice there is static inner class in MovingAverageDriver named PreprocessorMapper.

2.2.   Notice the data type of the key coming in is Stock and the incoming value is StockPrices. You are going to write the Input Format that makes this possible.

NOTE: Notice how much simpler it is to write the map method when the data coming in is already formatted for your particular application! The map method simply grabs the closing price of the stock and sends it to the Reducer

NOTE: This job uses the Reducer class as its Reducer - which simply outputs the same <key,value> pairs that it receives. However, this does not mean a Reducer is not necessary - the records will still be shuffled and sorted.

3.      Write the **StockInputFormat** Class

3.1.   Add a new class named StockInputFormat. Have it extend the FileInputFormat class in the *org.apache.hadoop.mapreduce.lib.input* package.

3.2.   Change the K generic to Stock, and the V generic to StockPrices.

3.3.   Have Eclipse generate the unimplemented method of FileInputStream, which is the createRecordReader method.

3.4.   Add a single line of code to createRecordReader:

return new StockReader();

You will get a compiler error, but you will write the StockReader class next.

4.      Write the StockReader Class

4.1.   Add a new static inner class to StockInputFormat named StockReader. Have it extend the RecordReader class.

4.2.   Set the key generic of RecordReader to Stock and the value generic to StockPrices.

4.3.   Let Eclipse generate the unimplemented methods of RecordReader.

4.4.   Add the following fields to StockReader:

private Stock key = new Stock();

private StockPrices value = new StockPrices();

private BufferedReader in;  
 private LineReader in;

private long start;

private long end;

private long currentPos;private Text line = new Text();

4.5.   In the initialize method of StockReader, cast the InputSplit parameter to a FileSplit instance:

FileSplit split = (FileSplit) arg0;

4.6.   Use the open method of FileSystem to open the path of the FileSplit:

Configuration conf = context.getConfiguration();

Path path = split.getPath();

InputStream is = path.getFileSystem(conf).open(path);

in = new LineReader(is, conf);

4.7.   Also in the initialize method, initialize the fields for tracking the offset in the file:

start = split.getStart();

end = start + split.getLength();

is.seek(start);

if (start != 0) {

start += in.readLine(new Text(), 0, (int) Math.min(Integer.MAX\_VALUE, end -start));

}

currentPos = start;

4.8.   In the close method, close the in stream object.

5.      Write the nextKeyValue Method

5.1.   Most of the work in RecordReader is done in the nextKeyValue method. Within this method, start by reading in the next line of input from the split:

if (currentPos > end) {

return false;

}

currentPos += in.readLine(line);

5.2.   If line is empty, return false. (This happens when you reach the end of the input split.):

if (line.getLength() == 0) { return false;}

5.3.   If the first line starts with “exchange”, then skip it:

if (line.toString().startsWith("exchange")) {

currentPos += in.readLine(line);

}

5.4.   Use the StringUtils.split method with a comma separator to split the line into an array of String objects:

String[] values = StringUtils.split(line.toString(), ',');

5.5.   Set the second String in the array as the stock symbol of the key:

key.setSymbol(values[1]);

5.6.   Set the third String as the date of the key:

key.setDate(values[2]);

The key field is now initialized with both the stock symbol and date.

5.7.   The next six String objects in the array represent the open, high, low, close, volume and adjusted close of the stock. Initialize all six fields of the StockPrices value field using these numbers.

5.8.   Return true from the nextKeyValue method.

5.9.   In the getCurrentKey method of StockReader, return the Stock field.

5.10.   In the getCurrentValue method, return the StockPrices field.

5.11.   Save your changes to **StockInputFormat.java**.

6.      Configure the Custom Input Format

6.1.   Notice in the run() method of MovingAverageDriver that the input format is

StockInputFormat.

6.2.   Notice also that the output format is a sequence file.

7.      Run the Job

7.1.   Build the project to create **movingaverage.jar**.

7.2.   Put the stock prices into HDFS:

$ cd ~/data/stock\_prices

$ hdfs dfs -mkdir stocks

$ hdfs dfs -put NYSE\_daily\_prices\_F.csv stocks

7.3.   Run the MapReduce job:

$ yarn jar movingaverage.jar

7.4.   Verify that the job executed properly. You should see a sequence file in the closingprices folder of HDFS:

$ hdfs dfs -ls closingprices

Found 1 items

-rw-r--r-- 1 train hdfs 5716158 closingprices/part-r-00000

NOTE: The output files are sequence files (a binary format), so you cannot view the contents using -cat. The hdfs dfs -text command can be used to read sequence files, but you would need Stock.class in your CLASSPATH for this work on your VM.

7.5.   Try viewing the contents of the output file:

$ hdfs dfs -cat closingprices/part-r-00000

Notice the output file is a sequence file, which is in binary format. You can use the -text command to view a sequence file in text format, but you would need the serialized classes in your CLASSPATH.

7.6.   Change directories to the **bin** folder so that Stock.class is in your CLASSPATH

$ cd bin

7.7.   View the contents of the sequence file:

$ hdfs dfs -text closingprices/part-r-00000

Notice the output of the sequence file this time is text, and the output should consist of a stock symbol, date and closing price on each line.

Result: The sequence file in HDFS is going to be the input for a future lab. The data in these files is read in as Java objects, simplifying any MapReduce job that wants to operate on this data.

**END**