CS6240 – PARALLEL DATA PROCESSING IN MAP-REDUCE

Class: CS6240-02

Homework Number: 4

Name: Arpit Mehta

PART 1 - SOURCE CODE:

SECONDARY SORT JAVA SOURCE CODE:

Pseudo-code:

```
FlightDataMapper: // The Mapper class
    // Input: <Object, line>
    // Output: <FlightDataMapperKey, Text>
    // FlightDataMapperKey is a composite key that implements WritableComparable class
    // It contains the String airlineId and integer flightMonth
    map(Object key, Text value, Context context):
        - Parse the input line to create FlightData object.
        - Check if flight information is valid for first leg:
                a. not cancelled
                b. year of the flight is 2008
        - If above conditions are true:
            a. Create a FlightDataMapperKey 'k' with the airlineId and month
            b. Create a output value = new Text(arrDelay)
        - Emit(k, value)

FlightDataPartitioner: // The partitioner class
    // Partitioning is based on the hashcode of airlineId in the FLightDataMapperKey
    public int getPartition(FlightDataMapperKey key, Text value, int numPartitions):
        partition = Math.abs(key.getAirlineId().hashCode()) % numPartitions
        return partition

FlightDataGroupComparator: // Grouping Comparator
    // Grouping comparison is done by comparing the airlineId's of 2 keys
    int compare(WritableComparable a, WritableComparable b):
        FlightDataMapperKey key1 = (FlightDataMapperKey) a
        FlightDataMapperKey key2 = (FlightDataMapperKey) b
        return (key1.getAirlineId().compareTo(key2.getAirlineId()))
```

```
FlightDataSortComparator: // Sort Comparison
    // Sorting of input keys in the reducer is done on the basis of airlineId and the month
    int compare(WritableComparable a, WritableComparable b):
        FlightDataMapperKey key1 = (FlightDataMapperKey) a
        FlightDataMapperKey key2 = (FlightDataMapperKey) b
        cpmResult = key1.getAirlineId().compareTo(key2.getAirlineId())
        if cmpResult == 0:
            cpmResult = compare(key1.getMonth(), key2.getMonth())
        return cmpResult

MonthlyFlightDataReducer: // The reducer
    // Input: <FlightDataMapperKey, Iterable<ArrDelay>>
    // Output: <airlineId, Comma separated list of (month, avg delay for that month)>
    void reduce(FlightDataMapperKey key, Iterable<Text> values, Context context):
        monthTotalDelay = 0.0
        monthTotalCount = 0
        // initialize integer array of size 12 to hold total delay of each month
        monthAvgDelay[12]
        prevMonth = 1
        for each delay value:
            month = key.getMonth()
            if month != previousMonth:
                  monthAvgDelay[prevMonth] = monthTotalDelay / monthTotalCount
                    // Reset values
                    monthTotalDelay = 0.0;
                    monthTotalCount = 0;
                    prevMonth = month;

             monthTotalCount++
            monthTotalDelay += ArrDelay

        // calculate average delay for last month
        monthAvgDelay[prevMonth] = monthTotalDelay / monthTotalCount

        // Initialize string builder
        StringBuilder valueSb = new StringBuilder();

        for(i=0; i<12; i++):
            if(i>0):
                valueSb.append(", ")
            valueSb.append("(").append(i + 1).append(",")
    .append(monthAvgDelay[i]).append(")");
```

```
// Emit
Emit(AirlineId, valueSb)
```

The FlightDataMapper class:

```
	/**
	 * Mapper class for flight data
	 *
	 * @author arpitm
	 *
	 */
	public static class FlightDataMapper extends
			Mapper<Object, Text, FlightDataMapperKey, Text> {
		// parser
		private FlightDataParser dataParser;

		@Override
		protected void setup(Context context) throws IOException,
				InterruptedException {
			super.setup(context);

			setParser(FlightDataParser.getInstance());
		}

		/**
		 * function parses the input data and outputs (k,v) pair. k is of
type
		 * FlightDataMapperKey and value is of type Text
		 */
		public void map(Object key, Text value, Context context)
				throws IOException, InterruptedException {

			// Get Flight Data
			FlightData fData =
dataParser.getFlightData(value.toString());

			// Define Mapper key and value
			FlightDataMapperKey outKey = null;
			Text outValue = null;

			if (FlightUtils.isValidFlight(fData)) {
				outKey = createKey(fData.getAirlineId().trim(),
						fData.getFlightMonth());
```

```
                        outValue = createValue(fData.getArrDelay().trim());

                        // Emit
                        if ((outKey != null) && (outValue != null)) {
                              // TODO For testing
                              // System.out.println(outKey.toString() + " --->
" +
                              // outValue);

                              context.write(outKey, outValue);
                        }
                  }
            }

            /**
             * Function returns an value for the map function
             *
             * @param arrDelay
             *
             * @return Text returnValue
             */
            private Text createValue(String arrDelay) {
                  Text returnValue = null;

                  if (!isNullString(arrDelay)) {
                        returnValue = new Text(arrDelay.trim());
                  }

                  return returnValue;
            }

            /**
             * returns a mapper key
             *
             * @param airlineId
             * @param month
             *
             * @return FlightDataMapperKey key
             */
            private FlightDataMapperKey createKey(String airlineId, int month)
{
                  FlightDataMapperKey key = null;
```

```
			if (!isNullString(airlineId) && (month >= 1) && (month <=
12)) {
				key = new FlightDataMapperKey(airlineId, month);
			}
			return key;
		}

		/**
		 * Helper function to check is a string is null.
		 *
		 * @param str
		 *
		 * @return boolean
		 */
		private boolean isNullString(String str) {
			if ((str == null) || (str.length() == 0)) {
				return true;
			} else {
				return false;
			}
		}

		@Override
		protected void cleanup(Context context) throws IOException,
				InterruptedException, NullPointerException {
			// Close the parser's string reader
			dataParser.getStrReader().close();

			// Close the parser's CSV reader
			dataParser.getCsvReader().close();

			super.cleanup(context);
		}

		/*
		 * Get & set methods for data parser.
		 */
		public FlightDataParser getParser() {
			return dataParser;
		}

		public void setParser(FlightDataParser parser) {
			this.dataParser = parser;
```

```
          }
     }

     /**
      * Partitions the FlightDataMapperKey based on the HashCode of
airlineId.
      * The getPartition function returns partition number between 0 and
      * numPartitions.
      *
      * @author arpitm
      *
      */
     public static class FlightDataPartitioner extends
               Partitioner<FlightDataMapperKey, Text> {

          @Override
          public int getPartition(FlightDataMapperKey key, Text value,
                    int numPartitions) {
               int partitionNum = (Math.abs(key.getAirlineId().hashCode())
% numPartitions);

               return partitionNum;
          }
     }

     /**
      * Groups input keys to the reducer based on the airlineId of the keys.
      *
      * @author arpitm
      *
      */
     public static class FlightDataGroupComparator extends WritableComparator
{

          protected FlightDataGroupComparator() {
               super(FlightDataMapperKey.class, true);
          }

          @Override
          public int compare(WritableComparable a, WritableComparable b) {
               FlightDataMapperKey key1 = (FlightDataMapperKey) a;
               FlightDataMapperKey key2 = (FlightDataMapperKey) b;
```

```
                  return (key1.getAirlineId().compareTo(key2.getAirlineId()));
            }
      }

      /**
       * Sort comparator class sorts the reducer input keys based on the
airlineId
       * and the month.
       *
       * @author arpitm
       *
       */
      public static class FlightDataSortComparator extends WritableComparator
{

            protected FlightDataSortComparator() {
                  super(FlightDataMapperKey.class, true);
            }

            @Override
            public int compare(WritableComparable a, WritableComparable b) {
                  FlightDataMapperKey key1 = (FlightDataMapperKey) a;
                  FlightDataMapperKey key2 = (FlightDataMapperKey) b;

                  int cmpResult =
key1.getAirlineId().compareTo(key2.getAirlineId());

                  if (cmpResult == 0) {
                        int m1 = key1.getMonth();
                        int m2 = key2.getMonth();

                        cmpResult = FlightDataMapperKey.compareMonths(m1, m2);
                  }

                  return cmpResult;
            }
      }

      /**
       * This reducer computes the average delay for each month for the input
       * FlightDataMapperKey.
       *
       * @author arpitm
```

```
         *
         */
        public static class MonthlyFlightDataReducer extends
                        Reducer<FlightDataMapperKey, Text, Text, Text> {

                public void reduce(FlightDataMapperKey key, Iterable<Text> values,
                                Context context) throws IOException,
InterruptedException {
                        // variable to hold total delay for each month
                        double totalDelay = 0.0;

                        // variable to hold count value for each month
                        int totalCount = 0;

                        // integer array to hold monthly average delay.
                        int[] monthAvgDelay = new int[12];

                        // previous month index
                        int prevMonth = 1;

                        for (Text value : values) {
                                int month = key.getMonth();

                                if (prevMonth != month) {
                                        monthAvgDelay[prevMonth - 1] = (int)
Math.ceil(totalDelay
                                                        / totalCount);

                                        // Reset values
                                        totalDelay = 0.0;
                                        totalCount = 0;
                                        prevMonth = month;
                                }

                                totalCount = totalCount + 1;

                                float delayMinutes =
Float.parseFloat(value.toString());
                                totalDelay += delayMinutes;
                        }

                        // Average delay for last month
                        monthAvgDelay[prevMonth - 1] = (int) Math.ceil(totalDelay
                        / totalCount);

                // String builder to hold the value string
                StringBuilder valueSb = new StringBuilder();

                // Loop through values to build reducer value string
                for (int i = 0; i < monthAvgDelay.length; i++) {
                    if (i > 0) {
                        valueSb.append(", ");
                    }

                    valueSb.append("(").append(i + 1).append(",")
                            .append(monthAvgDelay[i]).append(")");
                }

                // Emit
                context.write(new Text(key.getAirlineId()),
                        new Text(valueSb.toString()));
            }
        }

        /**
         * main: Driver function
         *
         * @param args
         *
         * @throws IOException
         * @throws ClassNotFoundException
         * @throws InterruptedException
         */
        public static void main(String[] args) throws IOException,
                InterruptedException, ClassNotFoundException {
            Configuration conf = new Configuration();

            String[] otherArgs = new GenericOptionsParser(conf, args)
                    .getRemainingArgs();

            // Arguments length check
            if (otherArgs.length != 2) {
                System.err
                        .println("Usage: MonthlyFlightDelay
    <input-file-path> <output-dir-path>");
                System.exit(2);
            }

            // Job: Monthly Average Flight Delay Calculation.
            Job job = new Job(conf,
                        "Airline-wise Monthly Average Flight Delay
Calculation.");
            job.setJarByClass(MonthlyFlightDelay.class);
            job.setMapperClass(FlightDataMapper.class);
            job.setPartitionerClass(FlightDataPartitioner.class);
            job.setGroupingComparatorClass(FlightDataGroupComparator.class);
            job.setSortComparatorClass(FlightDataSortComparator.class);
            job.setReducerClass(MonthlyFlightDataReducer.class);

            job.setMapOutputKeyClass(FlightDataMapperKey.class);
            job.setMapOutputValueClass(Text.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            // job.setNumReduceTasks(FlightConstants.NUM_REDUCE_TASKS);

            FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
            FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

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

FlightContants class: This class contains the constants used by the Map-Reduce program.

```
/**
 * @author arpitm
 *
 *          FlightConstants
 *
 *          Class contains constants for flight data processing
 *
 */
public class FlightConstants {
      /**
       * Number of reduce tasks
```

```
	 */
	public static final int NUM_REDUCE_TASKS = 20;

	/**
	 * The year
	 */
	public static final int YEAR = 2008;

	/**
	 * The delimiter used in Mappers
	 */
	public static final String DELIMITER = ":";

	/**
	 * Fixed fields in CSV flight data
	 */
	public static final int INDEX_FLIGHT_YEAR = 0;
	public static final int INDEX_FLIGHT_MONTH = 2;
	public static final int INDEX_FLIGHT_DATE = 5;
	public static final int INDEX_UNIQUE_CARRIER = 7;
	public static final int INDEX_ORIGIN = 11;
	public static final int INDEX_DESTINATION = 17;
	public static final int INDEX_DEP_TIME = 24;
	public static final int INDEX_ARR_TIME = 35;
	public static final int INDEX_ARR_DELAY_MINUTES = 37;
	public static final int INDEX_CANCELLED = 41;
	public static final int INDEX_DIVERTED = 43;
}
```

FlightData: Class contains flight data for each flight

```
/**
 * @author arpitm
 *
 *         FlightData
 *
 *         Class contains the relevant flight data & get and set functions
 *
 */
public class FlightData {

	/**
	 * The flight year
 */
private int flightYear;

/**
 * The month
 */
private int flightMonth;

/**
 * The unique carrier
 */
private String airlineId;

/**
 * Is flight cancelled
 */
private boolean isCancelled;

/**
 * s flight delayed
 */
private boolean isDiverted;

/**
 * Arrival delay in minutes
 */
private String arrDelay;

/**
 * Default constructor
 */
public FlightData() {
      setFlightYear(0);
      setFlightMonth(0);
      setCancelled(false);
      setDiverted(false);
      setArrDelay(new String());
      setAirlineId(new String());
}

@Override
public String toString() {
      String str = null;
```

```
            str = "Flight Data: [" + "airlineId = " + airlineId + "flightYear
= "
                        + flightYear + ", arrDelayMinutes = " + arrDelay
                        + ", isCancelled = "
                        + ((isCancelled == false) ? "false" : "true")
                        + ", isDiverted = "
                        + ((isDiverted == false) ? "false" : "true") + "]";

            return str;
      }

      /*
       * Getters & Setters
       */

      public int getFlightYear() {
            return flightYear;
      }

      public void setFlightYear(int flightYear) {
            this.flightYear = flightYear;
      }

      public boolean isCancelled() {
            return isCancelled;
      }

      public void setCancelled(boolean isCancelled) {
            this.isCancelled = isCancelled;
      }

      public boolean isDiverted() {
            return isDiverted;
      }

      public void setDiverted(boolean isDiverted) {
            this.isDiverted = isDiverted;
      }

      public String getArrDelay() {
            return arrDelay;
      }
```

```
	public void setArrDelay(String arrDelay) {
		this.arrDelay = arrDelay;
	}

	public String getAirlineId() {
		return airlineId;
	}

	public void setAirlineId(String airlineId) {
		this.airlineId = airlineId;
	}

	public int getFlightMonth() {
		return flightMonth;
	}

	public void setFlightMonth(int flightMonth) {
		this.flightMonth = flightMonth;
	}
}
```

FlightDataParser class: Is a singleton class that provides a parser object to parse though the input flight data from the csv file.

```
/**
 * FlightDataParser Singleton Class
 *
 * @author arpitm
 *
 */
public class FlightDataParser {
	/**
	 * The instance of FlightDataParser class
	 */
	private static FlightDataParser instance = null;
	/**
	 * The CSV Reader object
	 */
	private static CSVReader csvReader;

	/**
	 * The string reader object
	 */
	private static StringReader strReader;

	/**
	 * Constructor
	 */
	private FlightDataParser() {
		// A private Constructor prevents any other class from
instantiating.
	}

	/**
	 * getInstance
	 *
	 * @return object FlightDataParser
	 */
	public static FlightDataParser getInstance() {
		if (instance == null) {
			instance = new FlightDataParser();
		}

		return instance;
	}

	/**
	 * getFlightData: Function parses the CSV flight data and returns a
	 * FlightData object.
	 *
	 * @param String
	 *            line
	 *
	 * @return object FlightData
	 * @throws IOException
	 */
	public FlightData getFlightData(String line) throws IOException {
```

```
            FlightData fData = new FlightData();

            strReader = new StringReader(line);
            csvReader = new CSVReader(strReader);

            String[] values = csvReader.readNext();

            fData.setFlightYear(Integer

.parseInt(values[FlightConstants.INDEX_FLIGHT_YEAR].trim()));
            fData.setFlightMonth(Integer

.parseInt(values[FlightConstants.INDEX_FLIGHT_MONTH].trim()));
            fData.setArrDelay(values[FlightConstants.INDEX_ARR_DELAY_MINUTES]
                        .trim());
            fData.setCancelled((values[FlightConstants.INDEX_CANCELLED].trim()
                        .equals("0.00")) ? false : true);
            fData.setDiverted((values[FlightConstants.INDEX_DIVERTED].trim()
                        .equals("0.00")) ? false : true);

fData.setAirlineId(values[FlightConstants.INDEX_UNIQUE_CARRIER].trim());

            return fData;
      }

      /*
       * Get & Set methods
       */
      public CSVReader getCsvReader() {
            return csvReader;
      }

      public void setCsvReader(CSVReader csvReader) {
            FlightDataParser.csvReader = csvReader;
      }

      public StringReader getStrReader() {
            return strReader;
      }

      public void setStrReader(StringReader strReader) {
            FlightDataParser.strReader = strReader;
      }
```

```
}
```

FlightUtil class: Provides helper functions.

```
/**
 * Util class for Map-Reduce job. Provides helper functions.
 *
 * @author arpitm
 *
 */
public class FlightUtils {

      /**
       * Function checks if the flight is of interest.
       *
       * @param fData
       *            FlightData object
       *
       * @return boolean isValid
       */
      public static boolean isValidFlight(FlightData fData) {
            boolean isValid = false;

            isValid = (!fData.isCancelled() &&
isYearValid(fData.getFlightYear()));

            return isValid;
      }

      private static boolean isYearValid(int flightYear) {
            boolean yearValid = false;

            if (flightYear == FlightConstants.YEAR) {
                  yearValid = true;
            }

            return yearValid;
      }
}
```

Pseudocode for HPopulate:

```
// The mapper
HPopulateMapper:
    // Input: <Object, line>
    // Output: <rowKey<ImmutableBytesWritable>, rowData<Put>>
    void map(Object key, Text value, Context context):
        - Parse each line to get FlightData object.
        - Create rowKey: airlineId, year, 1 if cancelled else 0, 1 if diverted else 0, ststem
          time nanoseconds
        - Create rowData: month, arrDelayMinutes
        - Populate the HTable with rowKey and rowData

// There is no reducer

// Driver code
main:
    - Create HTable that needs to be populated
    - Set the table descriptor to TABLE_NAME = FlightData
    - Set column family descriptor
    - Set mapper output format class as TableOutputFormat.class
    - Set mapper output key class as ImmutableBytesWritable.class
    - Set mapper output value calss as Put.class
    - Set number of reduce tasks = 0. No reducers needed
    - wait for job to finish
```

Source Code:

```java
/**
 * @author arpitm
 *
 */
public class HPopulate {
	/**
	 * Mapper inserts data in HBase Table
	 *
	 * @author arpitm
	 *
	 */
	public static class HPopulateMapper extends
			Mapper<Object, Text, ImmutableBytesWritable, Put> {
		/**
		 * The parser to get FlightData object.
		 */
		FlightDataParser fDataParser = FlightDataParser.getInstance();
```

```
            public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {
                  FlightData fData =
fDataParser.getFlightData(value.toString());

                  int flightMonth = fData.getFlightMonth();
                  String flightDelayStr = fData.getArrDelay();

                  // Check if data is not null
                  if (!(isNullStr(flightDelayStr) && (flightMonth == 0))) {
                        String[] flightDelayStrParts =
flightDelayStr.split("\\.");

                        // Create row key for the hbase table
                        byte[] rKey = createRowKey(fData);

                        // Create data for the 'data' column of the table
                        String tableData = flightMonth + HConstants.DELIMITER
                                    + flightDelayStrParts[0].trim();

                        Put put = new Put(rKey);
                        put.add(Bytes.toBytes(HConstants.DATA_COLUMNFAMILY),
                                    Bytes.toBytes(HConstants.DATA_QUALIFIER),
                                    Bytes.toBytes(tableData));

                        context.write(new ImmutableBytesWritable(rKey), put);
                  }
            }

            /**
             * Program checks if the string is null or its length is 0
             *
             * @param str
             * @return
             */
            private boolean isNullStr(String str) {
                  if (("".equals(str)) || (str == null) || (str.length() ==
0)) {
                        return true;
                  } else {
                        return false;
                  }
            }
```

```
		/**
		 * returns a byte[] row key for the HBase table. The key contains
the
		 *
		 * @param fData
		 * @return
		 */
		private byte[] createRowKey(FlightData fData) {
			String key = fData.getAirlineId() + fData.getFlightYear()
					+ (fData.isCancelled() ? "1" : "0")
					+ (fData.isDiverted() ? "1" : "0");

			// The byte[] table row key
			byte[] rKey = new byte[2 * Bytes.SIZEOF_LONG];

			Bytes.putBytes(rKey, 0, Bytes.toBytes(Long.parseLong(key)),
0,
					Bytes.SIZEOF_LONG);

			// Add timestamp to rowKey
			long timeStamp = System.nanoTime();
			Bytes.putLong(rKey, Bytes.SIZEOF_LONG, timeStamp);

			return rKey;
		}
	}

	/**
	 * driver function
	 *
	 * @param args
	 * @throws IOException
	 * @throws InterruptedException
	 * @throws ClassNotFoundException
	 */
	public static void main(String[] args) throws IOException,
			ClassNotFoundException, InterruptedException {
		Configuration conf = HBaseConfiguration.create();

		String[] otherArgs = new GenericOptionsParser(conf, args)
				.getRemainingArgs();
```

```
            // Arguments length check
            if (otherArgs.length != 1) {
                  System.err.println("Usage: HPopulate <input-file-path>");
                  System.exit(2);
            }

            // Create table
            HBaseAdmin admin = new HBaseAdmin(conf);

            // Check if the table already exists
            if (!admin.tableExists(HConstants.FLIGHT_DATA_TABLE_NAME)) {
                  HTableDescriptor htd = new HTableDescriptor(
                              HConstants.FLIGHT_DATA_TABLE_NAME);
                  HColumnDescriptor hcd = new HColumnDescriptor(
                              HConstants.DATA_COLUMNFAMILY);
                  htd.addFamily(hcd);
                  admin.createTable(htd);
            } else {
                  System.out.println("HTable " +
HConstants.FLIGHT_DATA_TABLE_NAME
                              + " already exists!");
            }

            // TODO For testing
            System.out.println("HBase table " +
HConstants.FLIGHT_DATA_TABLE_NAME
                        + "created.");

            // The job: Populate HBase table
            Job job = new Job(conf, "HBase table populate");
            job.setJarByClass(HPopulate.class);
            job.setMapperClass(HPopulateMapper.class);
            job.setOutputFormatClass(TableOutputFormat.class);
            job.setOutputKeyClass(ImmutableBytesWritable.class);
            job.setOutputValueClass(Put.class);
            job.getConfiguration().set(TableOutputFormat.OUTPUT_TABLE,
                        HConstants.FLIGHT_DATA_TABLE_NAME);
            job.setNumReduceTasks(0);

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

            System.exit(job.waitForCompletion(true) ? 0 : 2);
            admin.close();
      }

}
```

Pseudocode for HCompute:

```
//Mapper
HComputeMapper:
      void setup(Context context):
            - Initialize HTable that receives the scanned data from hbase

      // input: <offset, line> This line is from a unique list of airlineId's preloaded in s3 bucket
      // output: <airlineId, Comma separated list of (month, avg delay for that month)>
      void map(Object key, Text value, Context context):
         -  initialize an int array of size 12 to hold the total delay for each of 12 months
         -  initialize an int array of size 12 to hold the count of a flight in each of 12 months
         -  Create scanner with start row: airlineId200801 and end row: airlineId200801
         -  For each scan result get the month and arrival delay for that month and increment
            corresponding int array values
         -  Loop from 1 to 12 and calculate avg delay for for each month and append it to a
            string
         -  Emit(airlineId, string)
```

Source Code:

```
public class HCompute {

      public static class HComputeMapper extends Mapper<Object, Text, Text,
Text> {
            /**
             * The client HBase table
             */
            HTable cliTable = null;

            @Override
            protected void setup(Context context) throws IOException,
                    InterruptedException {
                super.setup(context);

                // Initialize HBase table.
                cliTable = new HTable(context.getConfiguration(),
                            HConstants.FLIGHT_DATA_TABLE_NAME);
            }

            public void map(Object key, Text value, Context context)
                    throws IOException, InterruptedException {
                // integer array to hold monthly average delay.
                int[] monthTotalDelay = new int[12];

                // integer array for count of the flight for each month.
                int[] monthCount = new int[12];

                // Get the scan string
                Scan inputScan = getFlightDataScan(value.toString());

                ResultScanner resultScanner =
cliTable.getScanner(inputScan);

                // Loop through the scan result
                for (Result result : resultScanner) {
                    byte[] valBytes = result.getValue(

Bytes.toBytes(HConstants.DATA_COLUMNFAMILY),
                                Bytes.toBytes(HConstants.DATA_QUALIFIER));
                    String val = Bytes.toString(valBytes);

                    String[] valParts = val.split(HConstants.DELIMITER);
                    int month = Integer.parseInt(valParts[0]);
                    int delayMinutes = Integer.parseInt(valParts[1]);

                    // Increment delay and count
                    monthTotalDelay[month - 1] += delayMinutes;
                    monthCount[month - 1] += 1;
                }

                // Build output string
                StringBuilder sBuilder = new StringBuilder();

                // Loop through values to build reducer value string
```

```
                    for (int i = 0; i < monthTotalDelay.length; i++) {
                          if (i > 0) {
                                sBuilder.append(", ");
                          }

                          int monthAvgDelay = (int) Math
                                      .ceil(((double) monthTotalDelay[i]) /
monthCount[i]);

                          sBuilder.append("(").append(i + 1).append(",")
                                      .append(monthAvgDelay).append(")");
                    }

                    // Emit
                    context.write(value, new Text(sBuilder.toString()));

                    // Close result Scanner
                    if (resultScanner != null) {
                          resultScanner.close();
                    }
              }

              private Scan getFlightDataScan(String airlineId) {
                    // Need to start scanning all keys with input
                    // "airlineId + YEAR + Not cancelled + Not delayed"
                    String startScanString = airlineId + HConstants.YEAR + "00";
                    long startScanKey = Long.parseLong(startScanString);
                    byte[] startScanRKey =
Bytes.padTail(Bytes.toBytes(startScanKey),
                                Bytes.SIZEOF_LONG);

                    // Need to start scanning all keys with input
                    // "airlineId + YEAR + cancelled + Not delayed"
                    String stopScanString = airlineId + HConstants.YEAR + "01";
                    long stopScanKey = Long.parseLong(stopScanString);
                    byte[] stopScanRKey =
Bytes.padTail(Bytes.toBytes(stopScanKey),
                                Bytes.SIZEOF_LONG);

                    // Scan table.
                    Scan scan = new Scan(startScanRKey, stopScanRKey);
                    scan.setCaching(500);
                    scan.setCacheBlocks(false);
```

```
			return scan;
		}

		@Override
		protected void cleanup(Context context) throws IOException,
				InterruptedException {
			// Close the HTable
			cliTable.close();

			super.cleanup(context);
		}

	}

	/**
	 * driver function
	 *
	 * @param otherArgs
	 * @throws IOException
	 * @throws InterruptedException
	 * @throws ClassNotFoundException
	 */
	public static void main(String[] otherArgs) throws IOException,
			ClassNotFoundException, InterruptedException {
		Configuration conf = new Configuration();

		// Arguments length check
		if (otherArgs.length != 2) {
			System.err
					.println("Usage: HCompute <airline-id-file-path>
<output-dir-path>");
			System.exit(2);
		}

		// Job: Airline-wise monthly average delay computation
		Job job = new Job(conf, "Airline-wise monthly avg delay
calculation");
		job.setJarByClass(HCompute.class);
		job.setMapperClass(HComputeMapper.class);
		job.setMapOutputKeyClass(Text.class);
		job.setMapOutputValueClass(Text.class);
		job.setNumReduceTasks(0);
```

```
            // Read the airlines list from the txt files. 16 id's at a time
            FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
            FileInputFormat.setMaxInputSplitSize(job, 16);

            // Set output path
            FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

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

PERFORMANCE COMPARISON:

Run times:

5 Workers:

Program	Runtime
Java	2 min 7 sec
HPopulate	5 min 29 sec
HCompute	1 min 4 sec

11 Workers:

Program	Runtime
Java	1 min 57 sec
HPopulate	4 min 52 sec
HCompute	1 min 1 sec

DISCUSSION:

Coding and setup effort:

- The coding effort in case of HBase programs - HPopulate and HCompute were lesser than Java program. This is because the HBase programs didn’t require any Partitioner, Grouping Comparator, Sorting Comparator and Reducer.
- The setup effort in case HBase was also not very complicated as the Amazon EMR CLI commands dor setting up an hbase cluster and adding HPopulate and HCompute jar steps were very similar to normal Map-Reduce job cluster setup and jar steps.

Performance:
- The java program performs better with 10 workers compared to 5 workers. This is because the job gets distributed and done by more workers and hence faster.
- The HPopulate takes most amount of time, as it needs to read the entire data and populate the data into the HTables. Even with 10 workers HPopulate takes almost the same amount of time. The reason could be that with more workers, there are more HBase region server instances. So the all the workers need to populate data into more region servers, hence it takes almost same time.
- The total time of HBase job (HPopulate + HCompute) is greater then the Map-Reduce job, but we should compare the Java and the HCompute job. Because Map-reduce job reads the csv file and computes, whereas HCompute reads data from the HTable and computes. And in that aspect, the HCompute performs better and this is evident from the run-times reported. The reason is HCompute processes data only from those HTable rows that are valid for a particular airline and year (the rowkey is chosen such a way that the Scan start row and end row utilizes it efficiently). Also it is a map only job.

Scalability:
- The Map-Reduce program is scalable, as depending on the number of workers the mapper and reducer jobs will be determined. Each reducer gets all the records for a particular airlineId and given that we have 20 unique airlines, using 5 or 10 workers will evenly get the reducer jobs.
- I don’t observe a lot of scalability for HPopulate as all the workers has to populate the data into their HBase region servers.
- The HCompute is scalable which is explained in later section. (Load Balancing)

Load Balancing:
- The Map-Reduce job is load balanced well, as the reducer job gets divided by the airlineids. We have 20 unique airline Id’s and they get distributed well.
- HCompute is a map only task. And the input to this job is a list of airlineIds. Each of these airlineIds gets processed by a mapper. As this airlineId file is quite small (size < 64MB), all airlines will go to a single mapper by default. So this is bad load balancing. To improve this, I set the MaxInputSplitSize as 16 bytes, which makes sure that the airlineId’s file gets read in smaller chunks and are sent to many mapper tasks. Hence the task is well distributed.