ClassNumber: CS6240 Paralleled Data Processing
HWNumber: assignment4
Name: Yihan Tian

Source Code:

1. Secondary Sort

Pseudo Code

```
Input<key:object, value:text>
Output<key:compoundkey of flightId and month, value: delay>
Map {
   data = read from a line and validate data by flightYear, cancelled status and diverted status, then get flightId, month and delay;
   flightId = data.getId();
   month = data.getMonth();
   delay = data.getDelay();
   emit(<flightId, month>, delay);
}

Input<key:<fligthId, month>, value:delay>
Output<key:text of flightId, value:text  formatted monthly average delay>
Reduce {
   dataArr = Initialize array size of 12;
   month = flightId.getMonth();
   for (value : aggregated value sets) {
      dataArr[month-1][0] += value;  // compute total monthly delay
      dataArr[month-1][1] += 1;  // count total delays per month
   }
   Result  = “”;
   For (month =0; month<12; month++) {
      averageDelay = dataArr[month][0] / dataArr[month-1][1];
      result += (month + 1, averageDelay); //create formatted result output:
   }
   Emit(flightId, result);
}

Compare with two compound keys <flightId, month>
GroupComparator {
   return key1.flightId.compareTo(key2.flightId);
}
```

Source Code

SecondarySort Class with Main Function

```
/**
 * The type Secondary sort.
```

```java
 */
public class SecondarySort {

  /**
   * The type Flight mapper. Read from original csv data file, and parse data into
designed format,
   * which is using flightId and month as a compound key for output, and delay as
value for output.
   */
  public static class FlightMapper extends Mapper<Object, Text, FlightKey, Text> {

    private CSVReader csvReader;
    private StringReader strReader;

    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
      String info = value.toString();
      try {
        FlightKey outKey = validate(info);
        if (outKey != null) {
          Text outValue = new Text(outKey.getDelay());
          context.write(outKey, outValue);
        }
      } catch (CsvValidationException e) {
        e.printStackTrace();
      }
    }

    /**
     * Helper method that helps mapper read from data, and valid if the line of record
is valid to
     * parse to reducer. If the record is valid, put all information into a FlightKey
and return
     * it.
     */
    private FlightKey validate(String info) throws IOException, CsvValidationException
{
      strReader = new StringReader(info);
      csvReader = new CSVReader(strReader);
      String[] values = csvReader.readNext();
      String id = values[Constants.ID_IND];
      String year = values[Constants.YEAR_IND];
      String month = values[Constants.MONTH_IND];
      String canceled = values[Constants.CANCELED_IND];
      String derived = values[Constants.DERIVED_IND];
      String delay = values[Constants.DELAY_IND];

      // If any of the necessary info is blank in record, ignore that line.
      if (validString(id) || validString(year) || validString(month) ||
          validString(canceled) || validString(derived) || validString(delay)) {
        return null;
      }

      // If flight year is not 2008, or it is cancelled or diverted, then ignore the
record.
      if (!year.equals(Constants.VALID_YEAR)
|| !canceled.equals(Constants.VALID_STATUS) ||
          !derived.equals(Constants.VALID_STATUS)) {
        return null;
      }
      return new FlightKey(id, month, delay);
    }

    /**
     * Helper function that helps to check if any info from the record is blank.
     */
    private Boolean validString(String str) {
      return str == null || str.equals("") || str.length() == 0;
    }

    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
      super.cleanup(context);
      strReader.close();
      csvReader.close();
    }
  }

  /**
   * The type Group comparator. Combines FlightKey with same flightId into same group.
   */
  public static class GroupComparator extends WritableComparator {

    /**
     * Instantiates a new Group comparator.
     */
    public GroupComparator() {
      super(FlightKey.class, true);
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {
      FlightKey keyA = (FlightKey) a;
      FlightKey keyB = (FlightKey) b;
      return keyA.getAirId().compareTo(keyB.getAirId());
    }
  }

  /**
   * The type Sort comparator. Sorts FlightKey with ascending order.
   */
  public static class SortComparator extends WritableComparator {

    /**
     * Instantiates a new Sort comparator.
     */
    public SortComparator() {
      super(FlightKey.class, true);
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {
      FlightKey keyA = (FlightKey) a;
      FlightKey keyB = (FlightKey) b;
      return keyA.compareTo(keyB);
    }
  }

  /**
   * The type Flight reducer. Aggregate data from mapper, and calculate monthly
average delay for
```

```java
   * flights.
   */
  public static class FlightReducer extends Reducer<FlightKey, Text, Text, Text> {

    @Override
    protected void reduce(FlightKey key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

      // Initiate a double array which length is 12, to store each month's total delay
and
      // counts of delay for each flight.
      Double[][] monthDelay = new Double[Constants.MONTHS][2];

      for (int i = 0; i < Constants.MONTHS; i++) {
        monthDelay[i][0] = 0.0;
        monthDelay[i][1] = 0.0;
      }

      // Aggregate delay info from mapper.
      for (Text value : values) {
        int month = Integer.valueOf(key.getMonth());
        double delay = Double.valueOf(value.toString());
        monthDelay[month - 1][0] += 1.0;
        monthDelay[month - 1][1] += delay;
      }

      // Calculate monthly average delay, and use stringBuilder to format.
      StringBuilder stringBuilder = new StringBuilder();
      for (int i = 0; i < Constants.MONTHS; i++) {
        int delay = (int) Math.ceil(monthDelay[i][1] / monthDelay[i][0]);
        stringBuilder.append(Constants.LEFT).append(i +
1).append(Constants.SEPARATOR).append(delay)
            .append(Constants.RIGHT);
        if (i != (Constants.MONTHS - 1)) {
          stringBuilder.append(Constants.SEPARATOR);
        }
      }

      String res = stringBuilder.toString();
      context.write(new Text(key.getAirId()), new Text(res));
    }
  }

  /**
   * The entry point of application.
   *
   * @param args the input arguments
   * @throws IOException the io exception
   * @throws ClassNotFoundException the class not found exception
   * @throws InterruptedException the interrupted exception
   */
  public static void main(String[] args)
      throws IOException, ClassNotFoundException, InterruptedException {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "secondary sort");
    job.setJarByClass(SecondarySort.class);
    job.setMapperClass(FlightMapper.class);
    job.setReducerClass(FlightReducer.class);
    job.setGroupingComparatorClass(GroupComparator.class);
    job.setSortComparatorClass(SortComparator.class);
    job.setMapOutputKeyClass(FlightKey.class);
    job.setMapOutputValueClass(Text.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

FlightKey Class

```java
/**
 * The type Flight key. A compound key of flight mapper output, contains information
of flight Id,
 * month and minutes of delay.
 */
public class FlightKey implements WritableComparable<FlightKey> {

  private String airId;
  private String month;
  private String delay;

  /**
   * Instantiates a new Flight key.
   */
  public FlightKey() {
  }

  /**
   * Instantiates a new Flight key.
   *
   * @param airId the air id
   * @param month the month
   * @param delay the delay
   */
  public FlightKey(String airId, String month, String delay) {
    this.airId = airId;
    this.month = month;
    this.delay = delay;
  }

  /**
   * Gets air id.
   *
   * @return the air id
   */
  public String getAirId() {
    return airId;
  }

  /**
   * Sets air id.
   *
   * @param airId the air id
   */
  public void setAirId(String airId) {
    this.airId = airId;
  }

  /**
   * Gets month.
   *
   * @return the month
   */
  public String getMonth() {
    return month;
  }

  /**
   * Sets month.
   *
   * @param month the month
   */
  public void setMonth(String month) {
    this.month = month;
  }

  /**
   * Gets delay.
   *
   * @return the delay
   */
  public String getDelay() {
    return delay;
  }

  /**
   * Sets delay.
   *
   * @param delay the delay
   */
  public void setDelay(String delay) {
    this.delay = delay;
  }

  public int compareTo(FlightKey key) {
    int res = this.airId.compareTo(key.getAirId());
    if (res == 0) {
      int thisMonth = Integer.valueOf(this. month);
      int otherMonth = Integer.valueOf(key.getMonth());
      if (thisMonth < otherMonth) {
        res = -1;
      } else if (thisMonth > otherMonth) {
        res = 1;
      }
    }
    return res;
  }

  public void write(DataOutput dataOutput) throws IOException {
    dataOutput.writeUTF(this.airId);
    dataOutput.writeUTF(String.valueOf(this.month));
    dataOutput.writeUTF(this.delay);
  }

  public void readFields(DataInput dataInput) throws IOException {
    this.airId = dataInput.readUTF();
    this.month = dataInput.readUTF();
    this.delay = dataInput.readUTF();
  }
}
```

Constants Class, which contains constans in other classes

```java
/**
 * The type Constants.
 */
public class Constants {

  /**
   * The constant valid year.
   */
  public static final String VALID_YEAR = "2008";
  /**
   * The constant valid status.
   */
  public static final String VALID_STATUS = "0.00";
  /**
   * The constant SEPARATOR.
   */
  public static final String SEPARATOR = ",";
  /**
   * The constant LEFT bracket.
   */
  public static final String LEFT ="(";
  /**
   * The constant RIGHT bracket.
   */
  public static final String RIGHT = ")";

  /**
   * The constant flightId index.
   */
  public static final int ID_IND = 7;
  /**
   * The constant year index.
   */
  public static final int YEAR_IND = 0;
  /**
   * The constant month index.
   */
  public static final int MONTH_IND = 2;
  /**
   * The constant cancelled index.
   */
  public static final int CANCELED_IND = 41;
  /**
   * The constant diverted index.
   */
  public static final int DERIVED_IND = 43;
  /**
   * The constant delay index.
   */
  public static final int DELAY_IND = 37;
  /**
   * The constant total number of months.
   */
  public static final int MONTHS = 12;
}
```

2. HPopulate

   Pseudo Code

   Input<key:Object, value:text of a line>

Generates no output

```
Map {
   data = parse data from input value;
   rowKey = data.flightId() + currentTimeStamp;
   put = new Put();
   put.add(cf:verifyInfo, cl:year, val:data.getYear());
   put.add(cf:verifyInfo, cl:cancelled, val:data.getCancelled());
   put.add(cf:verifyInfo, cl:diverted, val:data.getDiverted());

   put.add(cf:essentialInfo, cl:month, val:data.getMonth());
   put.add(cf:essentialInfo, cl:delay, val:data.getDelay());

   put.add(cf:other, cl:other, val:data);

   emit(null, put);
}
```

Source Data

HPopulate Class which contains Mapper, Reducer and Main

```
/**
 * The type H populate. A Map-Only task that Read through original csv data file and
parse data into
 * HBase table.
 */
public class HPopulate {

  /**
   * The type Record mapper. Parse each line of record from csv file into HBase. HBase
table has
   * three column families, verifyInfo(including three columns: year, cancelled,
diverted),
   * essentialInfo(including month and delay), and other(including a string of all
other fields).
   */
  public static class RecordMapper extends Mapper<Object, Text,
ImmutableBytesWritable, Put> {

    private RecordProcessor processor;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
      super.setup(context);
      processor = RecordProcessor.getInstance();
    }

    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
      String info = value.toString();
      try {
        Put put = processor.createPut(info);
        context.write(null, put);
      } catch (CsvValidationException e) {
        e.printStackTrace();
```

```java
      }
    }

    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
      super.cleanup(context);
      processor.cleanUp();
    }
  }

  /**
   * Create a HBase table with designed 3 column families .
   *
   * @param tableName the table name
   * @throws IOException the io exception
   */
  public static void createTable(String tableName) throws IOException {
    HTableDescriptor htd = new HTableDescriptor(tableName);
    HColumnDescriptor verifyCol = new HColumnDescriptor(Constants.CF_VERIFY);
    HColumnDescriptor essentialInfo = new HColumnDescriptor(Constants.CF_ESSENTIAL);
    HColumnDescriptor other = new HColumnDescriptor(Constants.CF_OTHER);
    htd.addFamily(verifyCol);
    htd.addFamily(essentialInfo);
    htd.addFamily(other);
    Configuration conf = HBaseConfiguration.create();
    HBaseAdmin admin = new HBaseAdmin(conf);
    if (admin.tableExists(tableName)) {
      admin.disableTable(tableName);
      admin.deleteTable(tableName);
    }
    admin.createTable(htd);
  }

  /**
   * The entry point of application.
   *
   * @param args the input arguments
   * @throws IOException the io exception
   * @throws ClassNotFoundException the class not found exception
   * @throws InterruptedException the interrupted exception
   */
  public static void main(String[] args)
      throws IOException, ClassNotFoundException, InterruptedException {
    String tableName = Constants.TABLE_NAME;
    Configuration conf = new Configuration();
    conf.set(TableOutputFormat.OUTPUT_TABLE, tableName);
    createTable(tableName);
    Job job = new Job(conf, "HPopulate");
    job.setJarByClass(HPopulate.class);
    job.setMapperClass(RecordMapper.class);
//    job.setReducerClass(RecordReducer.class);
    job.setNumReduceTasks(0);
    job.setOutputKeyClass(ImmutableBytesWritable.class);
    job.setOutputValueClass(Put.class);
//
//    job.setMapOutputKeyClass(Text.class);
//    job.setMapOutputValueClass(Text.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TableOutputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

RecordProcessor that helps to process data from input file

```
/**
 * The type Record processor that helps read through original csv data file, and
process one record
 * into one put for mapper task.
 */
public class RecordProcessor {

  private StringReader stringReader;
  private CSVReader csvReader;
  private static RecordProcessor instance = null;

  private RecordProcessor() throws IOException {

  }

  /**
   * Gets instance.
   *
   * @return the instance
   * @throws IOException the io exception
   */
  public static RecordProcessor getInstance() throws IOException {
    if (instance == null) {
      instance = new RecordProcessor();
    }
    return instance;
  }

  /**
   * Create put put. Parse info from a line of record into a Put. Use flightId +
current time stamp
   * as rowKey, so that all record can be inserted into HBase. Put year, cancelled and
diverted from
   * input data into verifyInfo column family; month and delay into essentialInfo
column family, and
   * other info as a whole string in other column family.
   *
   * @param info the info
   * @return the put
   * @throws IOException the io exception
   * @throws CsvValidationException the csv validation exception
   */
  public Put createPut(String info) throws IOException, CsvValidationException {
    stringReader = new StringReader(info);
    csvReader = new CSVReader(stringReader);
    String[] values = csvReader.readNext();
    String timeStamp = String.valueOf(System.nanoTime());
    String airLineId = values[Constants.ID_IND];

    String key = airLineId + Constants.SEPARATOR + timeStamp;
    Put put = new Put(Bytes.toBytes(key));

    byte[] verifyColumn = Bytes.toBytes(Constants.CF_VERIFY);
    put.addColumn(verifyColumn, Bytes.toBytes(Constants.CL_YEAR),
        Bytes.toBytes(Constants.YEAR_IND));
    put.addColumn(verifyColumn, Bytes.toBytes(Constants.CL_CANCELLED),
        Bytes.toBytes(Constants.CANCELLED_IND));
```

```java
    put.addColumn(verifyColumn, Bytes.toBytes(Constants.CL_DIVERTED),
        Bytes.toBytes(Constants.DIVERTED_IND));

    byte[] essentialColumn = Bytes.toBytes(Constants.CF_ESSENTIAL);
    put.addColumn(essentialColumn, Bytes.toBytes(Constants.CL_MONTH),
        Bytes.toBytes(Constants.MONTH_IND));
    put.addColumn(essentialColumn, Bytes.toBytes(Constants.CL_DELAY),
        Bytes.toBytes(Constants.DELAY_IND));

    byte[] other = Bytes.toBytes(Constants.CF_OTHER);
    put.addColumn(other, Bytes.toBytes(Constants.CF_OTHER), Bytes.toBytes(info));

    return put;
  }

  /**
   * Gets string reader.
   *
   * @return the string reader
   */
  public StringReader getStringReader() {
    return stringReader;
  }

  /**
   * Sets string reader.
   *
   * @param stringReader the string reader
   */
  public void setStringReader(StringReader stringReader) {
    this.stringReader = stringReader;
  }

  /**
   * Gets csv reader.
   *
   * @return the csv reader
   */
  public CSVReader getCsvReader() {
    return csvReader;
  }

  /**
   * Sets csv reader.
   *
   * @param csvReader the csv reader
   */
  public void setCsvReader(CSVReader csvReader) {
    this.csvReader = csvReader;
  }

  /**
   * Clean up.
   *
   * @throws IOException the io exception
   */
  public void cleanUp() throws IOException {
    stringReader.close();
    csvReader.close();
  }
}
```

Constants

```java
/**
 * The type Constants.
 */
public class Constants {

  /**
   * The constant SEPARATOR.
   */
  public static final String SEPARATOR = ",";
  /**
   * The constant TABLE_NAME.
   */
  public static final String TABLE_NAME = "FlightMonthlyDelay";
  /**
   * The name for column family verifyInfo.
   */
  public static final String CF_VERIFY = "verifyInfo";
  /**
   * The name for column family essentialInfo.
   */
  public static final String CF_ESSENTIAL = "essentialInfo";

  /**
   * The constant CF_OTHER.
   */
  public static final String CF_OTHER = "other";

  /**
   * The name for column month.
   */
  public static final String CL_MONTH = "month";

  /**
   * The name for column delay.
   */
  public static final String CL_DELAY = "arrDelayMinutes";

  /**
   * The name for column cancelled.
   */
  public static final String CL_CANCELLED = "cancelled";

  /**
   * The name for column diverted.
   */
  public static final String CL_DIVERTED = "diverted";

  /**
   * The name for column year.
   */
  public static final String CL_YEAR = "year";

  /**
   * The constant YEAR_IND.
   */
  public static final int YEAR_IND = 0;

  /**
   * The constant MONTH_IND.
```

```
     */
    public static final int MONTH_IND = 2;

    /**
     * The constant ID_IND.
     */
    public static final int ID_IND = 7;

    /**
     * The constant DELAY_IND.
     */
    public static final int DELAY_IND = 37;

    /**
     * The constant CANCELLED_IND.
     */
    public static final int CANCELLED_IND = 41;

    /**
     * The constant DIVERTED_IND.
     */
    public static final int DIVERTED_IND = 43;

}
```

3. HCompute

Psuedo Code

```
Input<key: hbase rowKey(string of (flight,timestamp)), value: HBase Result>
Output<key: <flightId, month>, value: delay>
Map {
   flightId = inKey.toString().split(",")[0];
   year = inValue.getValue(cf:verifyInfo, cl:year);
   cancelled = inValue.getValue(cf:verifyInfo, cl:cancelled);
   diverted = inValue.getValue(cf:verifyInfo, cl:devirted);

   month = inValue.getValue(cf:essentialInfo, cl:month);
   delay = inValue.getValue(cf:essentialInfo, cl:delay);

   if (year == 2008 && cancelled == "0.00" && diverted == "0.00") {
      emit(<flightId, monnth>, delay);
   }
}

Input<key:<fligthId, month>, value:delay>
Output<key:text of flightId, value:text  formatted monthly average delay>
Reduce {
   dataArr = Initialize array size of 12;
   month = flightId.getMonth();
   for (value : aggregated value sets) {
      dataArr[month-1][0] += value;  // compute total monthly delay
      dataArr[month-1][1] += 1;  // count total delays per month
   }
   Result  = “”;
   For (month =0; month<12; month++) {
      averageDelay = dataArr[month][0] / dataArr[month-1][1];
      result += (month + 1, averageDelay); //create formatted result output:
   }
   Emit(flightId, result);
}
```

Compare with two compound keys <flightId, month>

```
GroupComparator {
   return key1.flightId.compareTo(key2.flightId);
}
```

Source Code

HCompute Class that read from a HBase table and compute average monthly flight delay

```
/**
 * The main task of HCompute. Read through the HBase to get information, and compute
flight monthly
 * delay for output.
 */
public class HCompute {

  /**
   * The type H mapper. Read from HBase table, and emit info containing flightId,
month, and delay.
   * Use flightId and month as compound key, delay as value.
   */
  public static class HMapper extends TableMapper<ReducerKey, Text> {

    @Override
    protected void map(ImmutableBytesWritable key, Result value, Context context)
        throws IOException, InterruptedException {
      String[] inputKey = Bytes.toString(key.get()).split(Constants.SEPARATOR);
      String flightId = inputKey[0];
      String month = Bytes
          .toString(value
              .getValue(Bytes.toBytes(Constants.CF_ESSENTIAL),
Bytes.toBytes(Constants.CL_MONTH)));
      String delay = Bytes
          .toString(value
              .getValue(Bytes.toBytes(Constants.CF_ESSENTIAL),
Bytes.toBytes(Constants.CL_DELAY)));
      String cancelled = Bytes
          .toString(value
              .getValue(Bytes.toBytes(Constants.CF_VERIFY),
Bytes.toBytes(Constants.CL_CANCELLED)));
      String diverted = Bytes
          .toString(value
              .getValue(Bytes.toBytes(Constants.CF_VERIFY),
Bytes.toBytes(Constants.CL_DIVERTED)));
      String year = Bytes
          .toString(
              value.getValue(Bytes.toBytes(Constants.CF_VERIFY),
Bytes.toBytes(Constants.CL_YEAR)));

      if (year.equals(Constants.REQUIRED_YEAR) &&
cancelled.equals(Constants.VALID_STATUES)
          && diverted.equals(Constants.VALID_STATUES)) {
        if (validStr(month) && validStr(delay)) {
          context.write(new ReducerKey(flightId, month), new Text(delay));
        }
      }
    }

    /**
     * Helper function that helps to check if info acquired from HBase is validate to
be computed in
     * reducer. If the checked string is empty or null, then the record will not be
send to reducer
     * for further calculation.
     */
    private Boolean validStr(String str) {
      return !str.equals("") && !(str.length() == 0) && !(str == null);
    }
  }


  /**
   * The type H reducer. Aggregate and calculate average monthly delay data for each
flight, and
   * output result with certain format.
   */
  public static class HReducer extends Reducer<ReducerKey, Text, Text, Text> {

    @Override
    protected void reduce(ReducerKey key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
      Double[][] monthDelay = new Double[Constants.MONTHS][2];

      for (int i = 0; i < Constants.MONTHS; i++) {
        monthDelay[i][0] = 0.0;
        monthDelay[i][1] = 0.0;
      }

      for (Text value : values) {
        int month = Integer.valueOf(key.getMonth());
        double delay = Double.valueOf(value.toString());
        monthDelay[month - 1][0] += 1.0;
        monthDelay[month - 1][1] += delay;
      }

      StringBuilder stringBuilder = new StringBuilder();
      for (int i = 0; i < Constants.MONTHS; i++) {
        int delay = (int) Math.ceil(monthDelay[i][1] / monthDelay[i][0]);
        stringBuilder.append(Constants.LEFT).append(i +
1).append(Constants.SEPARATOR).append(delay)
            .append(Constants.RIGHT);
        if (i != (Constants.MONTHS - 1)) {
          stringBuilder.append(Constants.SEPARATOR);
        }
      }

      String res = stringBuilder.toString();
      context.write(new Text(key.getFlightId()), new Text(res));
    }
  }
  /**
   * The type Group comparator. Compares two records, and if their flightId are the
same, then they
   * will be grouped as a same group.
   */
  public static class GroupComparator extends WritableComparator {

    /**
     * Instantiates a new Group comparator.
     */
    public GroupComparator() {
      super(ReducerKey.class, true);
    }

    @Override
    public int compare(Object a, Object b) {
      String flightIdA = ((ReducerKey) a).getFlightId();
      String flightIdB = ((ReducerKey) b).getFlightId();
      return flightIdA.compareTo(flightIdB);
    }
  }

  /**
   * The entry point of application.
   *
   * @param args the input arguments
   * @throws IOException the io exception
   * @throws ClassNotFoundException the class not found exception
   * @throws InterruptedException the interrupted exception
   * @throws ServiceException the service exception
   */
  public static void main(String[] args)
      throws IOException, ClassNotFoundException, InterruptedException,
ServiceException {
    Configuration conf = new Configuration();
    String hbaseSite = "/etc/hbase/conf/hbase-site.xml";
    conf.addResource(new File(hbaseSite).toURI().toURL());
    HBaseAdmin.checkHBaseAvailable(conf);

    conf.set(TableInputFormat.INPUT_TABLE, Constants.TABLE_NAME);
    Job job = Job.getInstance(conf, "HCompute");
    job.setJarByClass(HCompute.class);

    Scan scan = new Scan();
    scan.setCacheBlocks(false);
    scan.setCaching(500);
    scan.addFamily(Bytes.toBytes(Constants.CF_VERIFY));
    scan.addFamily(Bytes.toBytes(Constants.CF_ESSENTIAL));

    TableMapReduceUtil.initTableMapperJob(
        Constants.TABLE_NAME,
        scan,
        HMapper.class,
        ReducerKey.class,
        Text.class,
        job
    );

    job.setReducerClass(HReducer.class);
    job.setGroupingComparatorClass(GroupComparator.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    FileOutputFormat.setOutputPath(job, new Path(args[0]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

ReducerKey Class, contains flightId and month data

```
/**
 * The compound mapper output key. Contains information of flightId and month.
 */
public class ReducerKey implements WritableComparable<ReducerKey> {

  private String flightId;
  private String month;

  /**
   * Instantiates a new Reducer key.
   */
  public ReducerKey() {
  }

  /**
   * Instantiates a new Reducer key.
   *
   * @param flightId the flight id
   * @param month the month
   */
  public ReducerKey(String flightId, String month) {
    this.flightId = flightId;
    this.month = month;
  }

  /**
   * Gets flight id.
   *
   * @return the flight id
   */
  public String getFlightId() {
    return flightId;
  }

  /**
   * Sets flight id.
   *
   * @param flightId the flight id
   */
  public void setFlightId(String flightId) {
    this.flightId = flightId;
  }

  /**
   * Gets month.
   *
   * @return the month
   */
  public String getMonth() {
    return month;
  }
  /**
   * Sets month.
   *
   * @param month the month
   */
  public void setMonth(String month) {
    this.month = month;
  }

  public int compareTo(ReducerKey o) {
    if (this.flightId.equals(o.getFlightId())) {
      return this.month.compareTo(o.month);
    }
    return this.flightId.compareTo(o.flightId);
  }

  public void write(DataOutput dataOutput) throws IOException {
    dataOutput.writeUTF(flightId);
    dataOutput.writeUTF(month);
  }

  public void readFields(DataInput dataInput) throws IOException {
    this.flightId = dataInput.readUTF();
    this.month = dataInput.readUTF();
  }

  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (!(obj instanceof ReducerKey)) {
      return false;
    }
    return this.flightId.equals(((ReducerKey) obj).getFlightId()) && this.month
        .equals(((ReducerKey) obj).getMonth());
  }
}
```

Constants

```
/**
 * The type Constants.
 */
public class Constants {

  /**
   * The constant SEPARATOR.
   */
  public static final String SEPARATOR = ",";
  /**
   * The constant TABLE_NAME.
   */
  public static final String TABLE_NAME = "FlightMonthlyDelay";
  /**
   * The name for column family verifyInfo.
   */
  public static final String CF_VERIFY = "verifyInfo";
  /**
   * The name for column family essentialInfo.
   */
  public static final String CF_ESSENTIAL = "essentialInfo";
  /**
```

```
   * The name for column month.
   */
  public static final String CL_MONTH = "month";
  /**
   * The name for column delay.
   */
  public static final String CL_DELAY = "arrDelayMinutes";
  /**
   * The name for column cancelled.
   */
  public static final String CL_CANCELLED = "cancelled";
  /**
   * The name for column diverted.
   */
  public static final String CL_DIVERTED = "diverted";
  /**
   * The name for column year.
   */
  public static final String CL_YEAR = "year";
  /**
   * The constant of the valid year 2008.
   */
  public static final String REQUIRED_YEAR = "2008";
  /**
   * The constant VALID_STATUES.
   */
  public static final String VALID_STATUES = "0.00";
  /**
   * The constant left bracket.
   */
  public static final String LEFT = "(";
  /**
   * The constant right bracket.
   */
  public static final String RIGHT = ")";


  /**
   * The constant number of months.
   */
  public static final int MONTHS = 12;
}
```

Performance Analysis

	SecondarySort	Hpopulate	Hcompute
6 machines	106s	780s	168s
11 machines	70s	786	176s

1. Coding and setup
   For the setup part, SecondarySort is way easier than HBase. I firstly tried to configure on my own machine, but failed with many attempts. Then I start using AWS directly. This process is still very difficult, I tried a lot of ways to set up configuration in code setup, but for many times when there were problems, the EMR doesn't throw out any

exceptions or print any log files. To figure out what was wrong that caused the problem was extremely hard for me.

But the code was simpler and less in HBase part than in SecondarySort. One good thing for HBase is when everything is setup correctly, it is simpler to use.

2. Performance
    - Overall for the total runtime, SecondarySort performs better than HBase.

      In HPopulate, I have tried three ways to store data: (1) store every field of original data as a column, in a single column family, (2) store necessary data as a string in one column family, and other info as another string in another column family, (3) separate necessary info and store them in different columns and column families, and else as a string in another column family. It turns out that (2) showed a best performance among these 3, total runtime about 400s, and (1) the worst, the runtime was too long and I couldn't let it finish.

      In HCompute, query uses Scan, and in my design it will scan through a whole table to get records, which I believe has affected the overall performance. I should try to set validation conditions in HBase Table as a separate column or column family, and use filter in Scan to filter invalid data out before querying. This could be expected to better current performance.

    - The scalability is also better for the SecondarySort one. There is a significant difference between 6-machine pefromance and 11-machine performance of SecondarySort, but HBase tasks were not affected much by machine numbers. Scaling didn't make great difference for HBase tasks.

    - Used default load balancing settings. Both load balances seems okay from the syslog report.