Name: Khoa Cu Dang (Cody) Cao

Course: CS6240 Spring 2024

Homework 4 Report

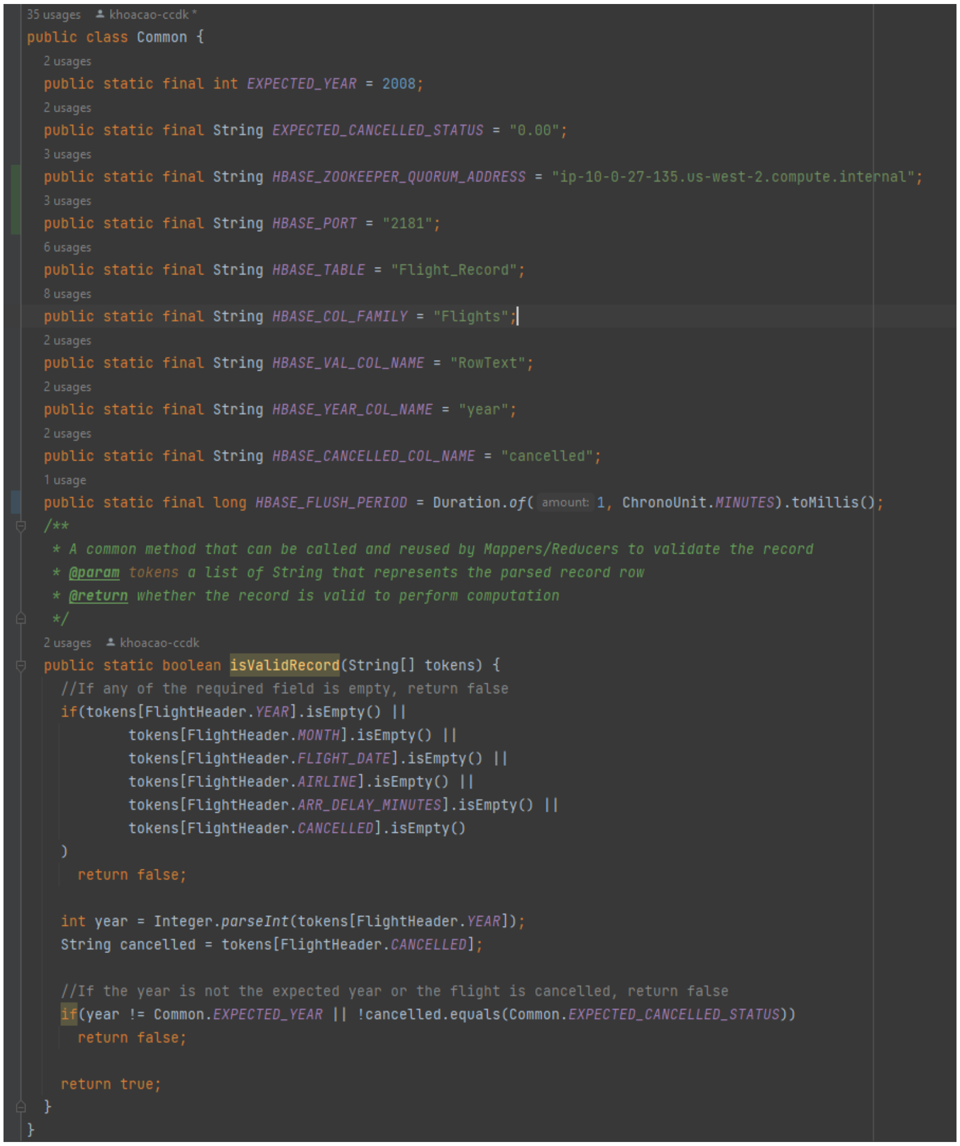
Source Code

Utility Classes

These classes are used commonly between the HBase-Compute and Secondary Sort program.

Common

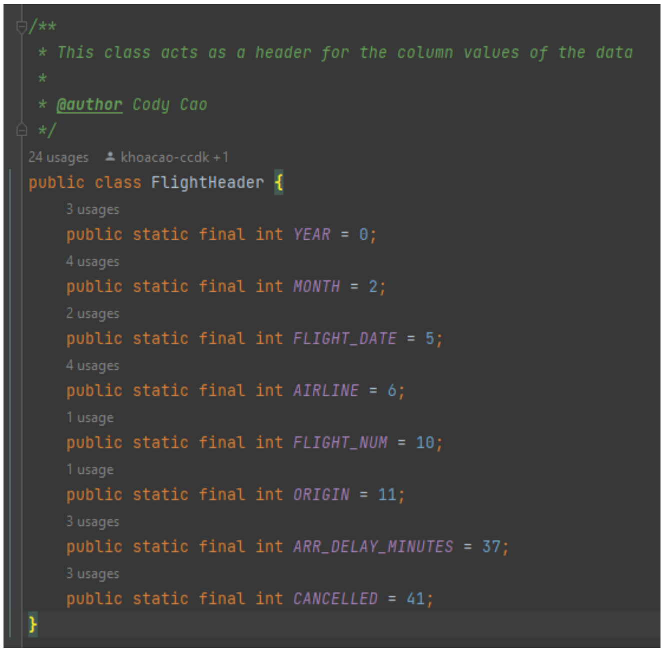
This class defines the constants used by both program, as well as a function to validate the flight records.



```java
35 usages  khoacao-ccdk *
public class Common {
    2 usages
    public static final int EXPECTED_YEAR = 2008;
    2 usages
    public static final String EXPECTED_CANCELLED_STATUS = "0.00";
    3 usages
    public static final String HBASE_ZOOKEEPER_QUORUM_ADDRESS = "ip-10-0-27-135.us-west-2.compute.internal";
    3 usages
    public static final String HBASE_PORT = "2181";
    6 usages
    public static final String HBASE_TABLE = "Flight_Record";
    8 usages
    public static final String HBASE_COL_FAMILY = "Flights";
    2 usages
    public static final String HBASE_VAL_COL_NAME = "RowText";
    2 usages
    public static final String HBASE_YEAR_COL_NAME = "year";
    2 usages
    public static final String HBASE_CANCELLED_COL_NAME = "cancelled";
    1 usage
    public static final long HBASE_FLUSH_PERIOD = Duration.of( amount: 1, ChronoUnit.MINUTES).toMillis();
    /**
     * A common method that can be called and reused by Mappers/Reducers to validate the record
     * @param tokens a list of String that represents the parsed record row
     * @return whether the record is valid to perform computation
     */
    2 usages  khoacao-ccdk
    public static boolean isValidRecord(String[] tokens) {
      //If any of the required field is empty, return false
      if(tokens[FlightHeader.YEAR].isEmpty() ||
              tokens[FlightHeader.MONTH].isEmpty() ||
              tokens[FlightHeader.FLIGHT_DATE].isEmpty() ||
              tokens[FlightHeader.AIRLINE].isEmpty() ||
              tokens[FlightHeader.ARR_DELAY_MINUTES].isEmpty() ||
              tokens[FlightHeader.CANCELLED].isEmpty()
      )
        return false;

      int year = Integer.parseInt(tokens[FlightHeader.YEAR]);
      String cancelled = tokens[FlightHeader.CANCELLED];

      //If the year is not the expected year or the flight is cancelled, return false
      if(year != Common.EXPECTED_YEAR || !cancelled.equals(Common.EXPECTED_CANCELLED_STATUS))
        return false;

      return true;
    }
}
```

FlightHeader

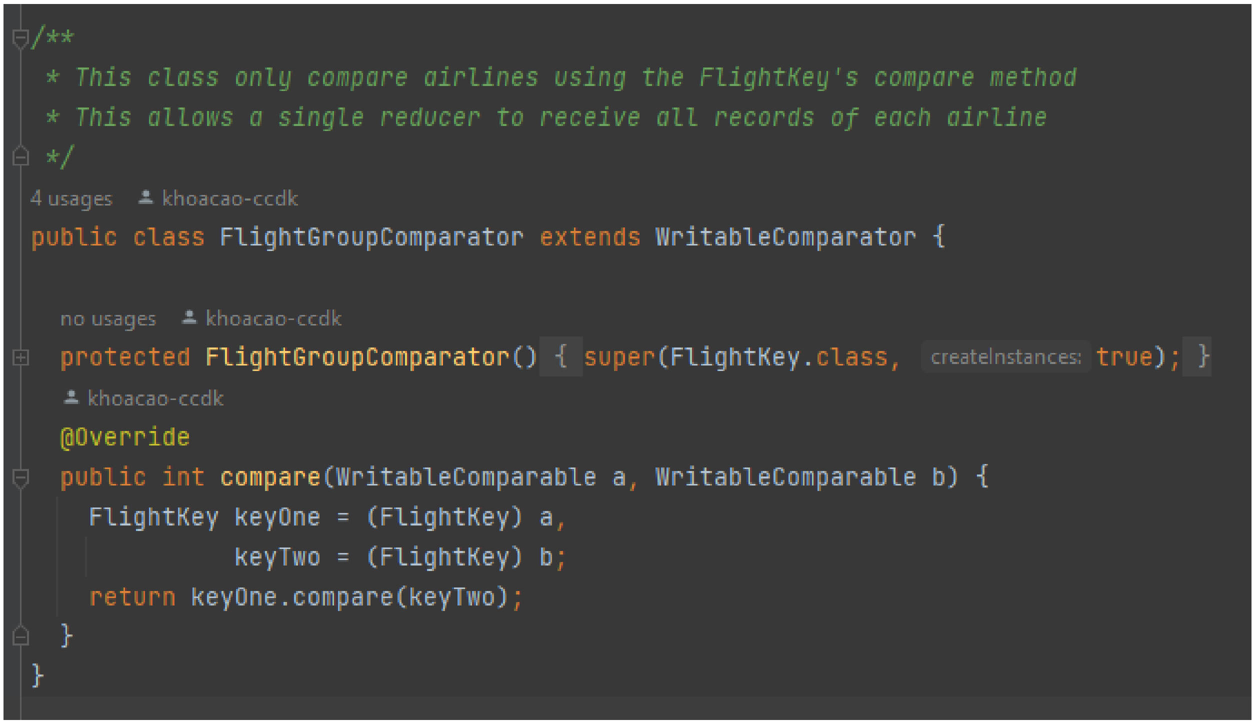
This class defines the columns being used from the data



```java
/**
 * This class acts as a header for the column values of the data
 *
 * @author Cody Cao
 */
24 usages   khoacao-ccdk +1
public class FlightHeader {
    3 usages
    public static final int YEAR = 0;
    4 usages
    public static final int MONTH = 2;
    2 usages
    public static final int FLIGHT_DATE = 5;
    4 usages
    public static final int AIRLINE = 6;
    1 usage
    public static final int FLIGHT_NUM = 10;
    1 usage
    public static final int ORIGIN = 11;
    3 usages
    public static final int ARR_DELAY_MINUTES = 37;
    3 usages
    public static final int CANCELLED = 41;
}
```

FlightGroup Comparator

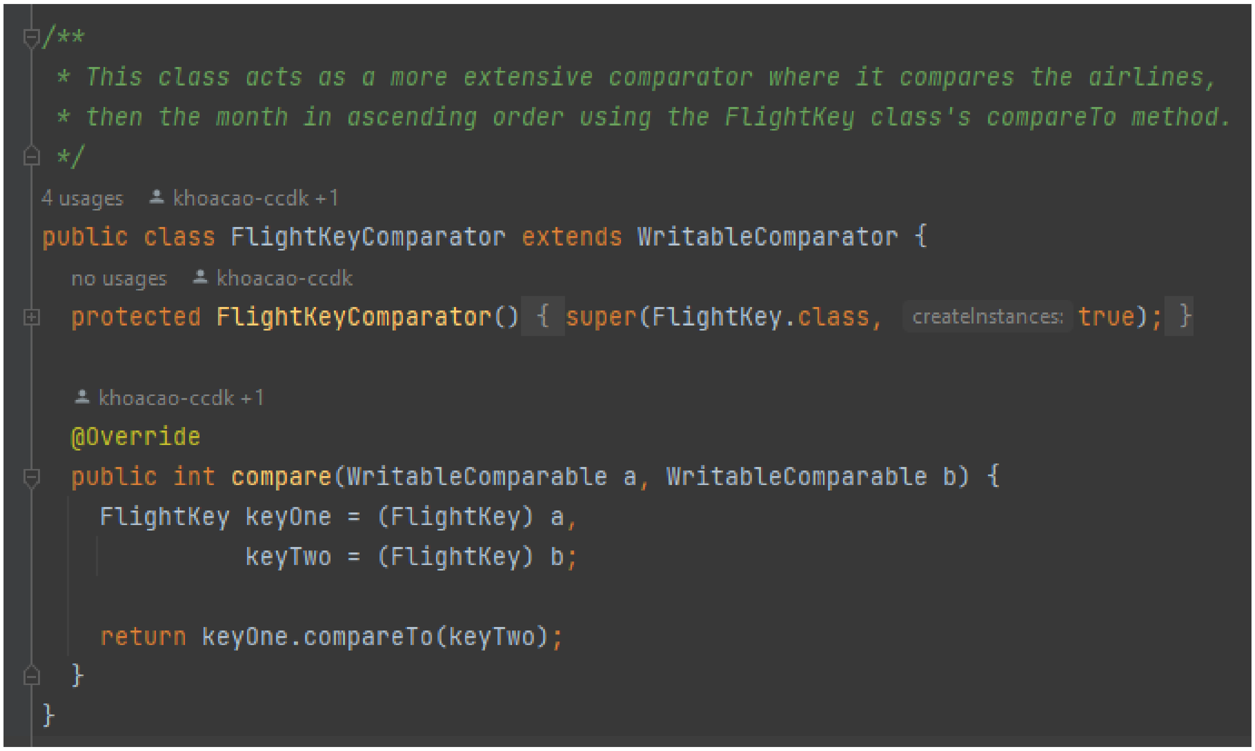
This class is used as a grouping comparator class in order to group key-value pairs with similar carrier into the same reducer.



```java
/**
 * This class only compare airlines using the FlightKey's compare method
 * This allows a single reducer to receive all records of each airline
 */
4 usages   khoacao-ccdk
public class FlightGroupComparator extends WritableComparator {

    no usages   khoacao-ccdk
    protected FlightGroupComparator() { super(FlightKey.class,  createInstances: true); }
    khoacao-ccdk
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        FlightKey keyOne = (FlightKey) a,
                  keyTwo = (FlightKey) b;
        return keyOne.compare(keyTwo);
    }
}
```

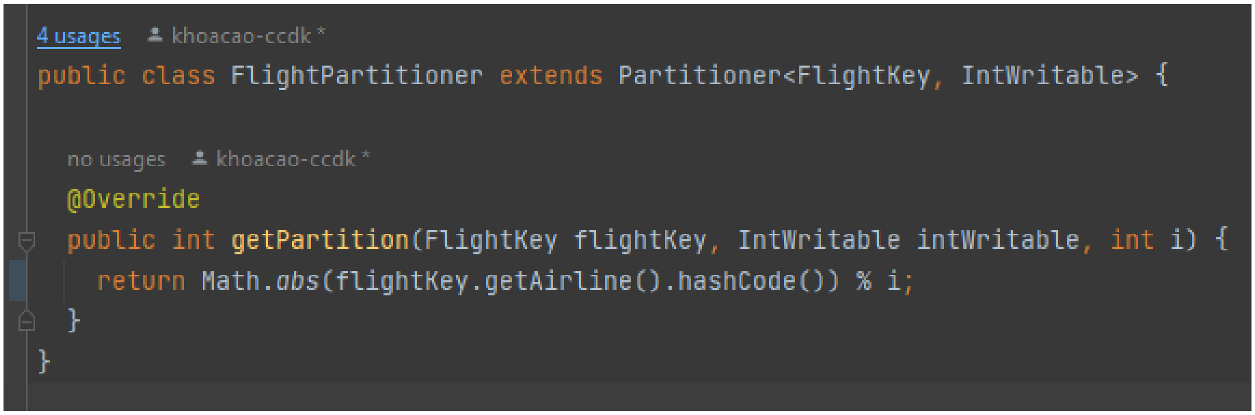
FlightKey Comparator

This class is used as a sort comparator class how output key-value pairs from the map are sorted.



```
/**
 * This class acts as a more extensive comparator where it compares the airlines,
 * then the month in ascending order using the FlightKey class's compareTo method.
 */
4 usages  khoacao-ccdk +1
public class FlightKeyComparator extends WritableComparator {
    no usages  khoacao-ccdk
    protected FlightKeyComparator() { super(FlightKey.class, createInstances: true); }

    khoacao-ccdk +1
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        FlightKey keyOne = (FlightKey) a,
                  keyTwo = (FlightKey) b;

        return keyOne.compareTo(keyTwo);
    }
}
```

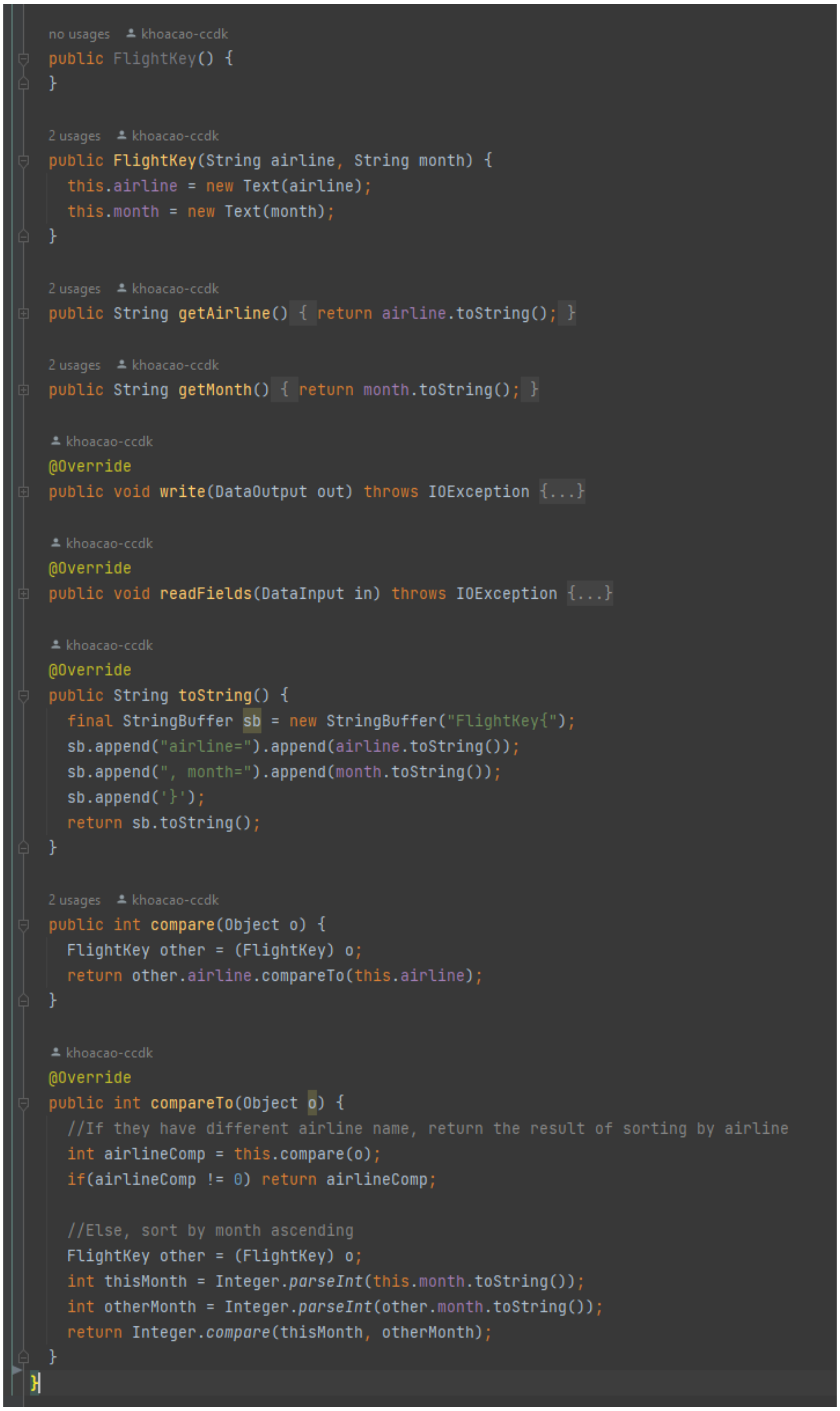
FlightPartitioner



```
4 usages  khoacao-ccdk *
public class FlightPartitioner extends Partitioner<FlightKey, IntWritable> {

    no usages  khoacao-ccdk *
    @Override
    public int getPartition(FlightKey flightKey, IntWritable intWritable, int i) {
        return Math.abs(flightKey.getAirline().hashCode()) % i;
    }
}
```

FlightKey

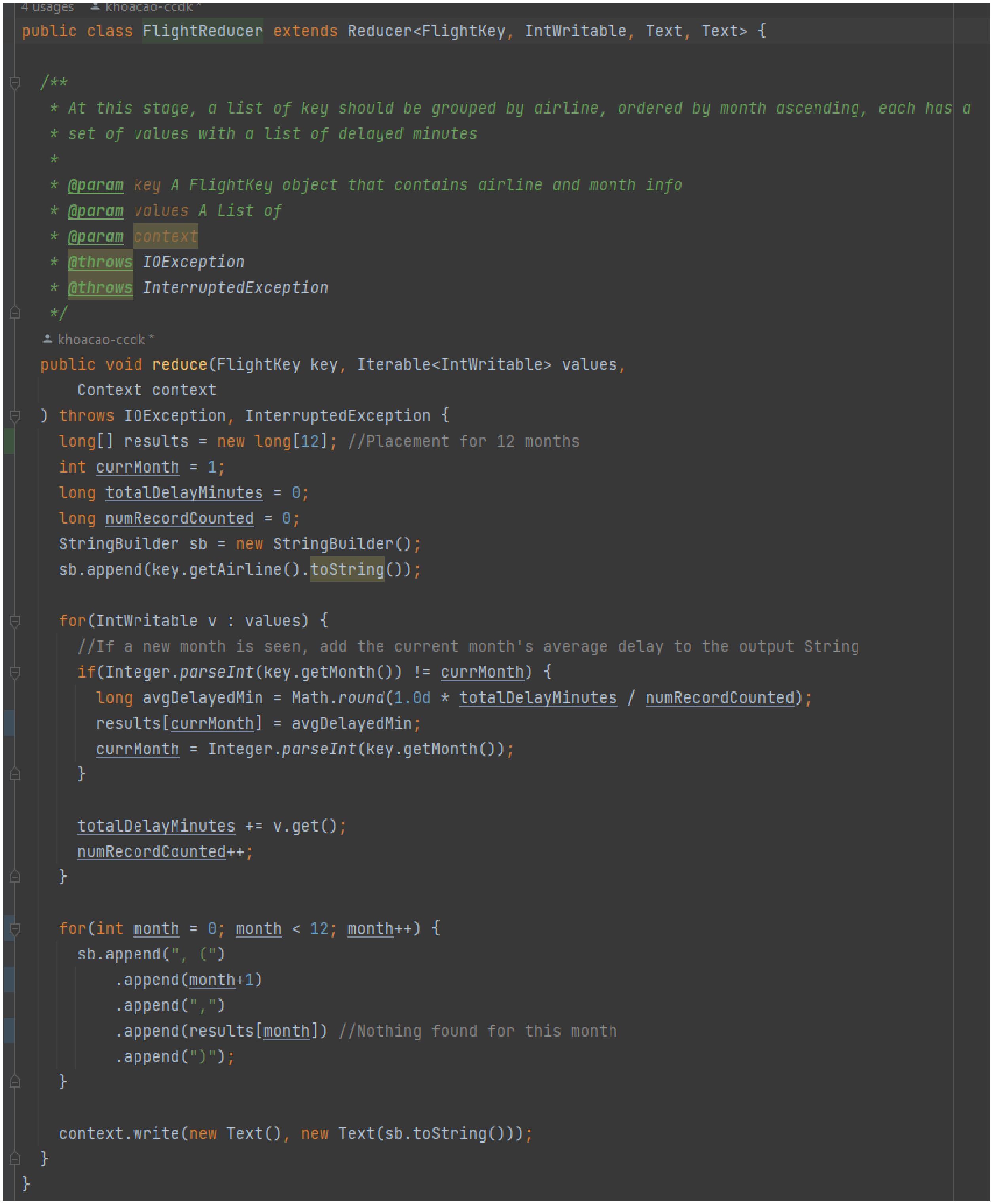
This class acts as the key for the map phase. It contains the String that contains the name of the airline, as well as the month that the flight record happened. It also contains two compare methods: one for comparing the airline name, the other comparing both the name and the month.



```java
no usages  khoacao-ccdk
public FlightKey() {
}

2 usages  khoacao-ccdk
public FlightKey(String airline, String month) {
  this.airline = new Text(airline);
  this.month = new Text(month);
}

2 usages  khoacao-ccdk
public String getAirline() { return airline.toString(); }

2 usages  khoacao-ccdk
public String getMonth() { return month.toString(); }

khoacao-ccdk
@Override
public void write(DataOutput out) throws IOException {...}

khoacao-ccdk
@Override
public void readFields(DataInput in) throws IOException {...}

khoacao-ccdk
@Override
public String toString() {
  final StringBuffer sb = new StringBuffer("FlightKey{");
  sb.append("airline=").append(airline.toString());
  sb.append(", month=").append(month.toString());
  sb.append('}');
  return sb.toString();
}

2 usages  khoacao-ccdk
public int compare(Object o) {
  FlightKey other = (FlightKey) o;
  return other.airline.compareTo(this.airline);
}

khoacao-ccdk
@Override
public int compareTo(Object o) {
  //If they have different airline name, return the result of sorting by airline
  int airlineComp = this.compare(o);
  if(airlineComp != 0) return airlineComp;

  //Else, sort by month ascending
  FlightKey other = (FlightKey) o;
  int thisMonth = Integer.parseInt(this.month.toString());
  int otherMonth = Integer.parseInt(other.month.toString());
  return Integer.compare(thisMonth, otherMonth);
}
}
```

Reducer

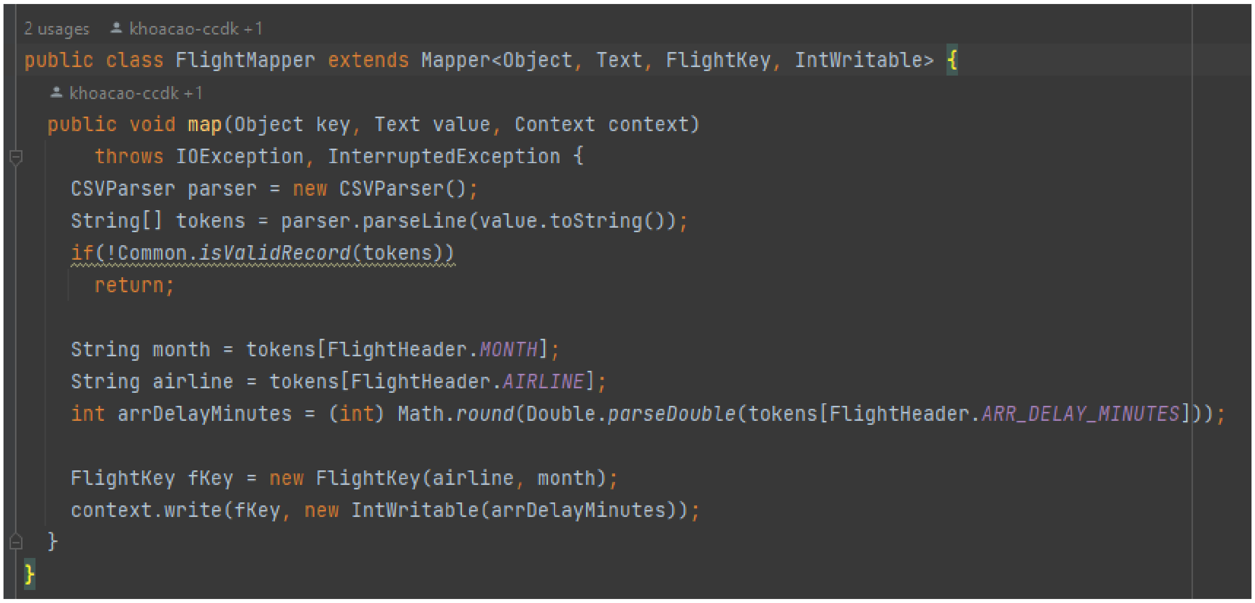
Like other Utility classes, I'm using the same Reducer approach to Secondary Sort and H-Computer since it would make things easier for development, given the fact that the expected output from the map phase of two programs should be identical.



```java
4 usages   khoacao-ccdk *
public class FlightReducer extends Reducer<FlightKey, IntWritable, Text, Text> {

  /**
   * At this stage, a list of key should be grouped by airline, ordered by month ascending, each has a
   * set of values with a list of delayed minutes
   *
   * @param key A FlightKey object that contains airline and month info
   * @param values A List of
   * @param context
   * @throws IOException
   * @throws InterruptedException
   */
  khoacao-ccdk *
  public void reduce(FlightKey key, Iterable<IntWritable> values,
      Context context
  ) throws IOException, InterruptedException {
    long[] results = new long[12]; //Placement for 12 months
    int currMonth = 1;
    long totalDelayMinutes = 0;
    long numRecordCounted = 0;
    StringBuilder sb = new StringBuilder();
    sb.append(key.getAirline().toString());

    for(IntWritable v : values) {
      //If a new month is seen, add the current month's average delay to the output String
      if(Integer.parseInt(key.getMonth()) != currMonth) {
        long avgDelayedMin = Math.round(1.0d * totalDelayMinutes / numRecordCounted);
        results[currMonth] = avgDelayedMin;
        currMonth = Integer.parseInt(key.getMonth());
      }

      totalDelayMinutes += v.get();
      numRecordCounted++;
    }

    for(int month = 0; month < 12; month++) {
      sb.append(", (")
          .append(month+1)
          .append(",")
          .append(results[month]) //Nothing found for this month
          .append(")");
    }

    context.write(new Text(), new Text(sb.toString()));
  }
}
```

Mapper

Secondary Sort

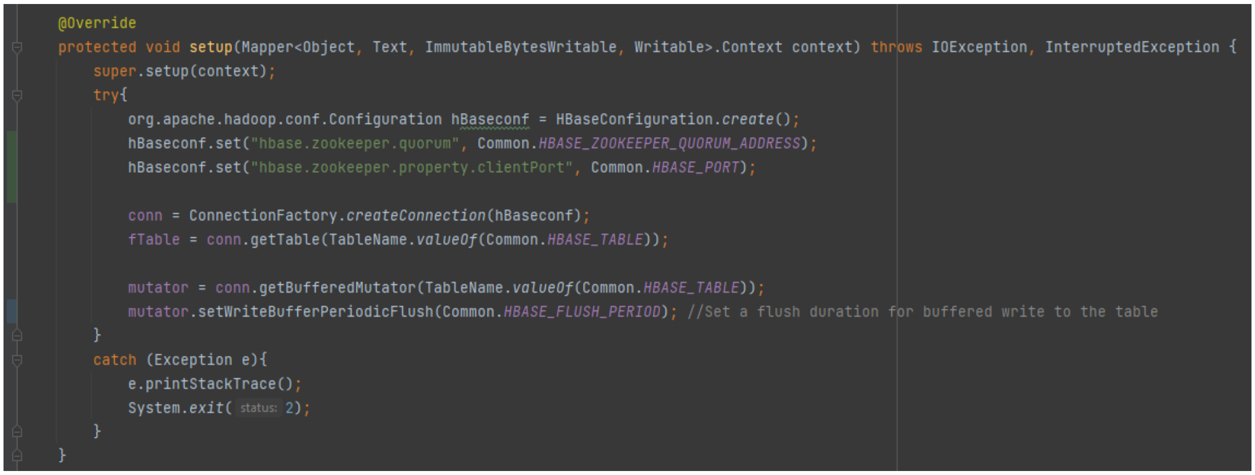


```
public class FlightMapper extends Mapper<Object, Text, FlightKey, IntWritable> {
  public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
    CSVParser parser = new CSVParser();
    String[] tokens = parser.parseLine(value.toString());
    if(!Common.isValidRecord(tokens))
      return;

    String month = tokens[FlightHeader.MONTH];
    String airline = tokens[FlightHeader.AIRLINE];
    int arrDelayMinutes = (int) Math.round(Double.parseDouble(tokens[FlightHeader.ARR_DELAY_MINUTES]));

    FlightKey fKey = new FlightKey(airline, month);
    context.write(fKey, new IntWritable(arrDelayMinutes));
  }
}
```

The map phase of Secondary Sort validates the data (including filtering records of flights happened in 2008) and emits the key, as well as the arrival delay minutes value.

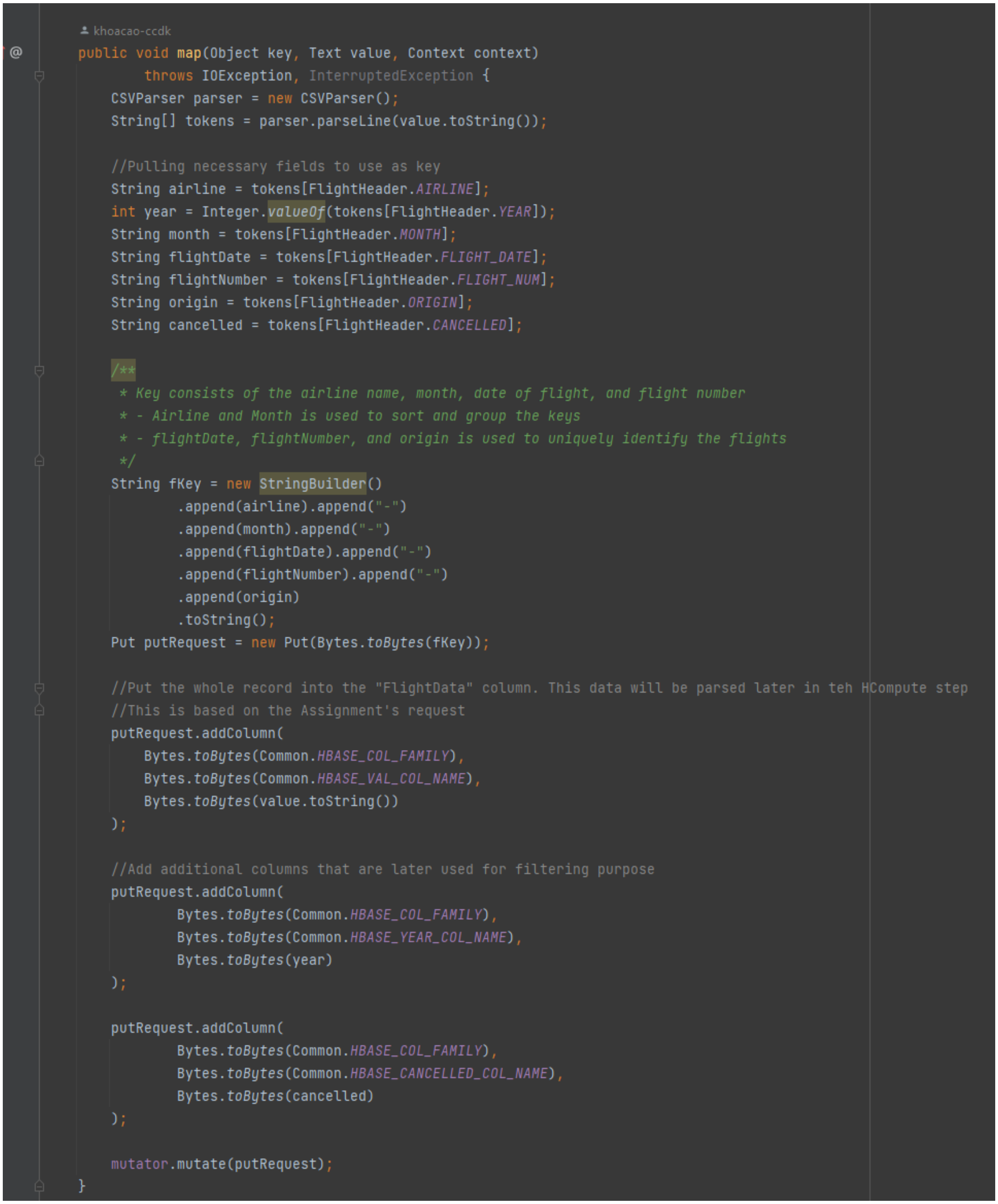
H-Populate

For each map instance, I set up a connection towards the table in HBase, as well as set up a one-minute flush period, which means the mappers would only write to HBase every minute.



```
@Override
protected void setup(Mapper<Object, Text, ImmutableBytesWritable, Writable>.Context context) throws IOException, InterruptedException {
    super.setup(context);
    try{
        org.apache.hadoop.conf.Configuration hBaseconf = HBaseConfiguration.create();
        hBaseconf.set("hbase.zookeeper.quorum", Common.HBASE_ZOOKEEPER_QUORUM_ADDRESS);
        hBaseconf.set("hbase.zookeeper.property.clientPort", Common.HBASE_PORT);

        conn = ConnectionFactory.createConnection(hBaseconf);
        fTable = conn.getTable(TableName.valueOf(Common.HBASE_TABLE));

        mutator = conn.getBufferedMutator(TableName.valueOf(Common.HBASE_TABLE));
        mutator.setWriteBufferPeriodicFlush(Common.HBASE_FLUSH_PERIOD); //Set a flush duration for buffered write to the table
    }
    catch (Exception e){
        e.printStackTrace();
        System.exit( status: 2);
    }
}
```

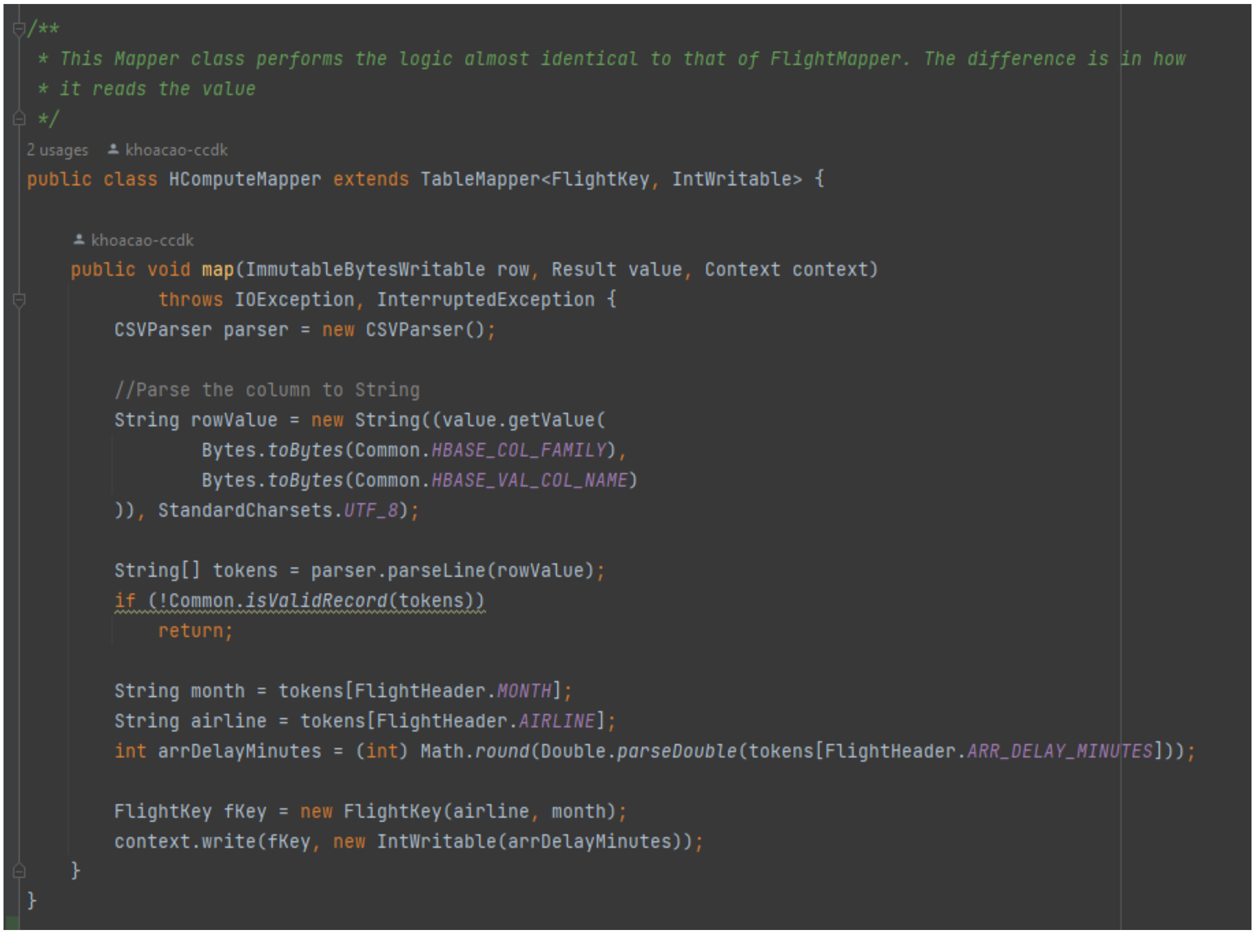
For quick filtering in later stage, the map function would create some additional columns to the data row (year and cancelled), while also writing the whole record as an additional column according to the requirements of the assignment. No filtering efforts is performed at this stage.



```
khoacao-ccdk
public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
    CSVParser parser = new CSVParser();
    String[] tokens = parser.parseLine(value.toString());

    //Pulling necessary fields to use as key
    String airline = tokens[FlightHeader.AIRLINE];
    int year = Integer.valueOf(tokens[FlightHeader.YEAR]);
    String month = tokens[FlightHeader.MONTH];
    String flightDate = tokens[FlightHeader.FLIGHT_DATE];
    String flightNumber = tokens[FlightHeader.FLIGHT_NUM];
    String origin = tokens[FlightHeader.ORIGIN];
    String cancelled = tokens[FlightHeader.CANCELLED];

    /**
     * Key consists of the airline name, month, date of flight, and flight number
     * - Airline and Month is used to sort and group the keys
     * - flightDate, flightNumber, and origin is used to uniquely identify the flights
     */
    String fKey = new StringBuilder()
            .append(airline).append("-")
            .append(month).append("-")
            .append(flightDate).append("-")
            .append(flightNumber).append("-")
            .append(origin)
            .toString();
    Put putRequest = new Put(Bytes.toBytes(fKey));

    //Put the whole record into the "FlightData" column. This data will be parsed later in teh HCompute step
    //This is based on the Assignment's request
    putRequest.addColumn(
        Bytes.toBytes(Common.HBASE_COL_FAMILY),
        Bytes.toBytes(Common.HBASE_VAL_COL_NAME),
        Bytes.toBytes(value.toString())
    );

    //Add additional columns that are later used for filtering purpose
    putRequest.addColumn(
            Bytes.toBytes(Common.HBASE_COL_FAMILY),
            Bytes.toBytes(Common.HBASE_YEAR_COL_NAME),
            Bytes.toBytes(year)
    );

    putRequest.addColumn(
            Bytes.toBytes(Common.HBASE_COL_FAMILY),
            Bytes.toBytes(Common.HBASE_CANCELLED_COL_NAME),
            Bytes.toBytes(cancelled)
    );

    mutator.mutate(putRequest);
}
```

H-Compute

This essentially performs the same task as that of the SecondarySort program.

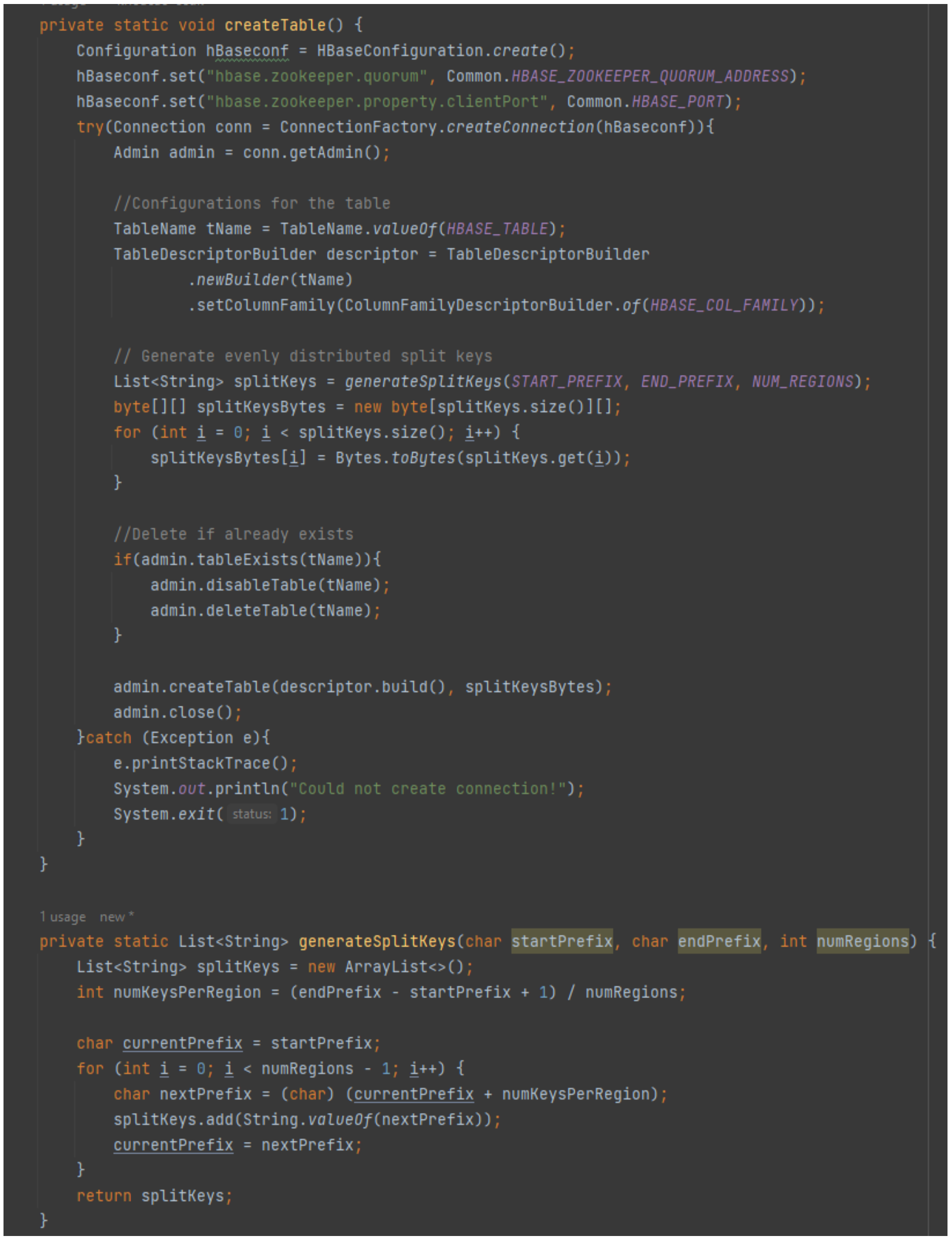


```java
/**
 * This Mapper class performs the logic almost identical to that of FlightMapper. The difference is in how
 * it reads the value
 */
public class HComputeMapper extends TableMapper<FlightKey, IntWritable> {

    public void map(ImmutableBytesWritable row, Result value, Context context)
            throws IOException, InterruptedException {
        CSVParser parser = new CSVParser();

        //Parse the column to String
        String rowValue = new String((value.getValue(
                Bytes.toBytes(Common.HBASE_COL_FAMILY),
                Bytes.toBytes(Common.HBASE_VAL_COL_NAME)
        )), StandardCharsets.UTF_8);

        String[] tokens = parser.parseLine(rowValue);
        if (!Common.isValidRecord(tokens))
            return;

        String month = tokens[FlightHeader.MONTH];
        String airline = tokens[FlightHeader.AIRLINE];
        int arrDelayMinutes = (int) Math.round(Double.parseDouble(tokens[FlightHeader.ARR_DELAY_MINUTES]));

        FlightKey fKey = new FlightKey(airline, month);
        context.write(fKey, new IntWritable(arrDelayMinutes));
    }
}
```

Main program

HPopulate

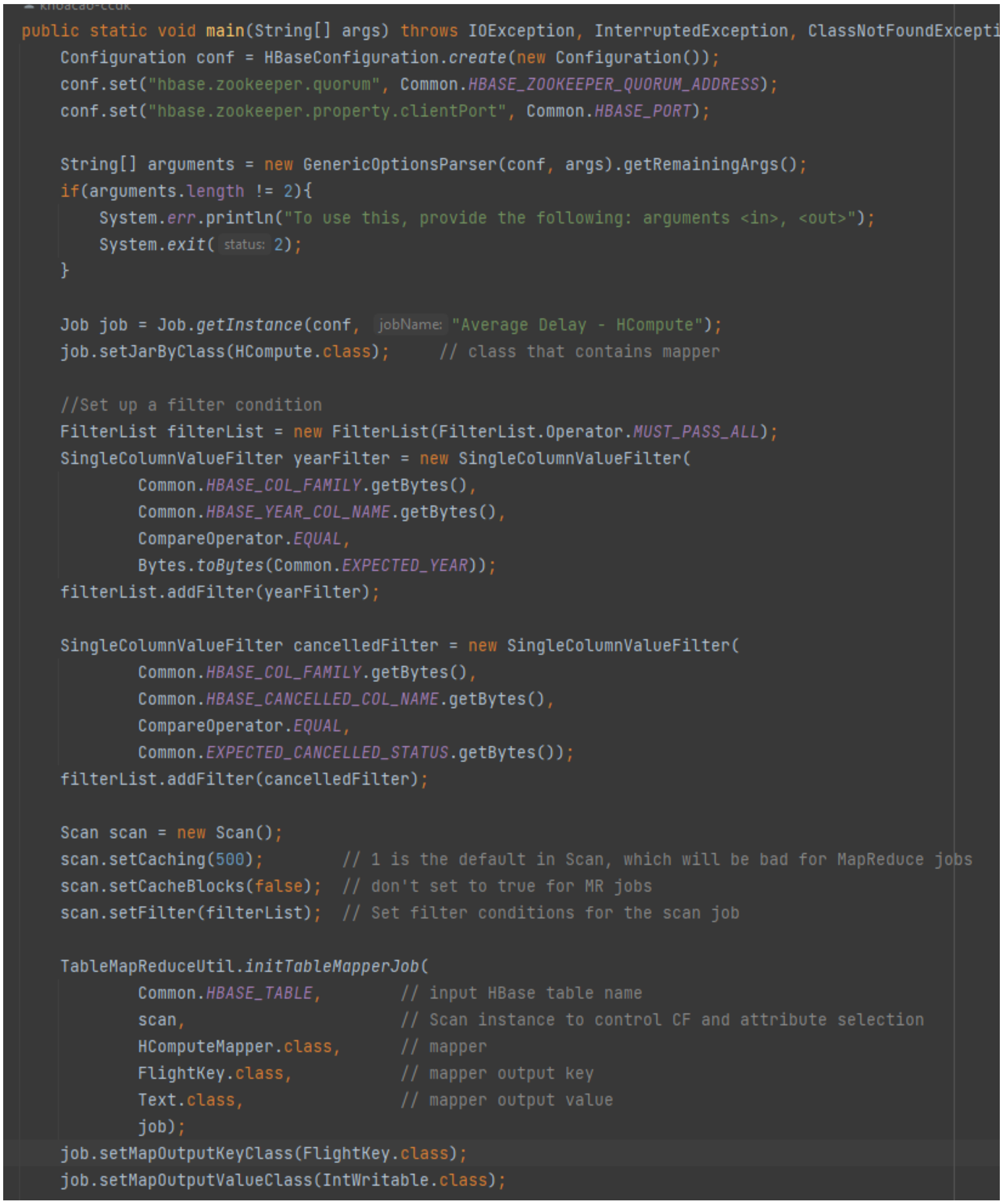
Omitting the unnecessary parts, here is the logic to create the table in HBase.



```
private static void createTable() {
    Configuration hBaseconf = HBaseConfiguration.create();
    hBaseconf.set("hbase.zookeeper.quorum", Common.HBASE_ZOOKEEPER_QUORUM_ADDRESS);
    hBaseconf.set("hbase.zookeeper.property.clientPort", Common.HBASE_PORT);
    try(Connection conn = ConnectionFactory.createConnection(hBaseconf)){
        Admin admin = conn.getAdmin();

        //Configurations for the table
        TableName tName = TableName.valueOf(HBASE_TABLE);
        TableDescriptorBuilder descriptor = TableDescriptorBuilder
                .newBuilder(tName)
                .setColumnFamily(ColumnFamilyDescriptorBuilder.of(HBASE_COL_FAMILY));

        // Generate evenly distributed split keys
        List<String> splitKeys = generateSplitKeys(START_PREFIX, END_PREFIX, NUM_REGIONS);
        byte[][] splitKeysBytes = new byte[splitKeys.size()][];
        for (int i = 0; i < splitKeys.size(); i++) {
            splitKeysBytes[i] = Bytes.toBytes(splitKeys.get(i));
        }

        //Delete if already exists
        if(admin.tableExists(tName)){
            admin.disableTable(tName);
            admin.deleteTable(tName);
        }

        admin.createTable(descriptor.build(), splitKeysBytes);
        admin.close();
    }catch (Exception e){
        e.printStackTrace();
        System.out.println("Could not create connection!");
        System.exit( status: 1);
    }
}

1 usage  new *
private static List<String> generateSplitKeys(char startPrefix, char endPrefix, int numRegions) {
    List<String> splitKeys = new ArrayList<>();
    int numKeysPerRegion = (endPrefix - startPrefix + 1) / numRegions;

    char currentPrefix = startPrefix;
    for (int i = 0; i < numRegions - 1; i++) {
        char nextPrefix = (char) (currentPrefix + numKeysPerRegion);
        splitKeys.add(String.valueOf(nextPrefix));
        currentPrefix = nextPrefix;
    }
    return splitKeys;
}
```

HCompute

Here is the logic of setting up the mappers, as well as filtering unnecessary records.



```
khoacao-ccuk
public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundExcepti
    Configuration conf = HBaseConfiguration.create(new Configuration());
    conf.set("hbase.zookeeper.quorum", Common.HBASE_ZOOKEEPER_QUORUM_ADDRESS);
    conf.set("hbase.zookeeper.property.clientPort", Common.HBASE_PORT);

    String[] arguments = new GenericOptionsParser(conf, args).getRemainingArgs();
    if(arguments.length != 2){
        System.err.println("To use this, provide the following: arguments <in>, <out>");
        System.exit( status: 2);
    }

    Job job = Job.getInstance(conf,  jobName: "Average Delay - HCompute");
    job.setJarByClass(HCompute.class);     // class that contains mapper

    //Set up a filter condition
    FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL);
    SingleColumnValueFilter yearFilter = new SingleColumnValueFilter(
            Common.HBASE_COL_FAMILY.getBytes(),
            Common.HBASE_YEAR_COL_NAME.getBytes(),
            CompareOperator.EQUAL,
            Bytes.toBytes(Common.EXPECTED_YEAR));
    filterList.addFilter(yearFilter);

    SingleColumnValueFilter cancelledFilter = new SingleColumnValueFilter(
            Common.HBASE_COL_FAMILY.getBytes(),
            Common.HBASE_CANCELLED_COL_NAME.getBytes(),
            CompareOperator.EQUAL,
            Common.EXPECTED_CANCELLED_STATUS.getBytes());
    filterList.addFilter(cancelledFilter);

    Scan scan = new Scan();
    scan.setCaching(500);        // 1 is the default in Scan, which will be bad for MapReduce jobs
    scan.setCacheBlocks(false);  // don't set to true for MR jobs
    scan.setFilter(filterList);  // Set filter conditions for the scan job

    TableMapReduceUtil.initTableMapperJob(
            Common.HBASE_TABLE,        // input HBase table name
            scan,                      // Scan instance to control CF and attribute selection
            HComputeMapper.class,      // mapper
            FlightKey.class,           // mapper output key
            Text.class,                // mapper output value
            job);
    job.setMapOutputKeyClass(FlightKey.class);
    job.setMapOutputValueClass(IntWritable.class);
```

Pseudo code

Secondary Sort

Mapper

For each flight record:

1. Check if the record is valid (no necessary field missing, the year is 2008)
2. Construct a key (FlightKey) from the airline name and the month value.
3. Emit the key and the arrDelayMinutes value.

Grouping Comparator

Group keys in accordance with the name of the airline. Thus, keys with the same airline name are grouped together and sent towards the same reducer.

Sort Comparator

Sort key in accordance with both name of the airline and the month value. This allows grouped key to further sorted and grouped into individual months within 2008.

Reducer

For each key-value pair:

1. Check if the month value of the key is the same as the currently considered month.
    a. If true:
        i. totalDelayMinutes += delay minutes
        ii. numRecordCounted++
    b. If false:
        i. Calculate the average delay minute value of the previous month.
        ii. Set the value currently considered month to the key’s month value.
2. For each month within the year, format their String (month: average delay minutes) and append it to the output String.
3. Emit the result.

HPopulate

Mapper

1. At the setup step, create a connection towards HBase. Setup flush period to 1 minute.
2. For each flight record:
    a. Extracts airline, month, flight date, flight number, and origin for key to uniquely identify a record.
    b. Add the whole record line as a column.
    c. Add two additional columns (year and cancelled) for filtering later.
    d. Add to the buffered write queue.
3. At the cleanup step, flush any pending writes, then close the connection.

HCompute

1. Setup a connection towards HBase.
2. Setup filter rule (year = 2008, not cancelled).
3. Setup scan and add filter rules.
4. Initiate mapper jobs for the scan.

Mapper

For each record given:

1. Validates the records (not missing necessary values)
2. Construct a key (FlightKey) from the airline name and the month value.
3. Emit the key and the arrDelayMinutes value.

Grouping Comparator

Group keys in accordance with the name of the airline. Thus, keys with the same airline name are grouped together and sent towards the same reducer.

Sort Comparator

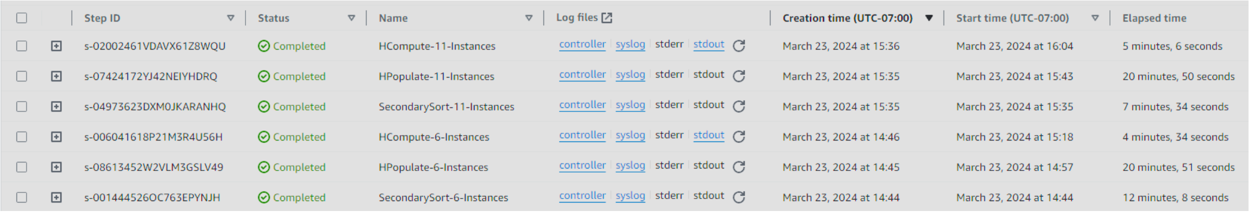
Sort key in accordance with both name of the airline and the month value. This allows grouped key to further sorted and grouped into individual months within 2008.

Reducer

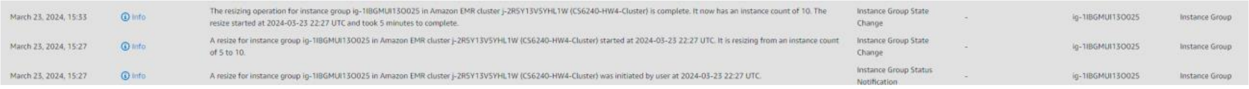
For each key-value pair:

1. Check if the month value of the key is the same as the currently considered month.
   a. If true:
      i. totalDelayMinutes += delay minutes
      ii. numRecordCounted++
   b. If false:
      i. Calculate the average delay minute value of the previous month.
      ii. Set the value currently considered month to the key's month value.
2. For each month within the year, format their String (month: average delay minutes) and append it to the output String.
3. Emit the result.

Proof of EMR execution

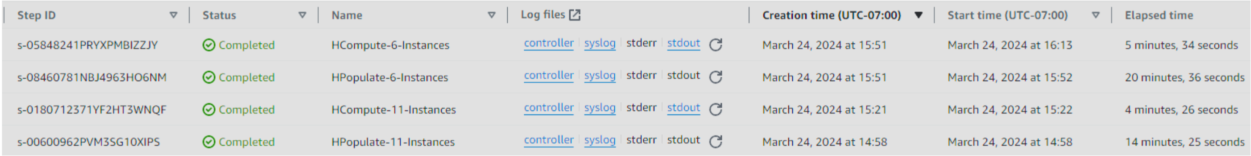


□		Step ID	Status	Name	Log files	Creation time (UTC-07:00)	Start time (UTC-07:00)	Elapsed time
□	⊞	s-02002461VDAVX61Z8WQU	✓ Completed	HCompute-11-Instances	controller | syslog | stderr | stdout	March 23, 2024 at 15:36	March 23, 2024 at 16:04	5 minutes, 6 seconds
□	⊞	s-07424172YJ42NEIYHDRQ	✓ Completed	HPopulate-11-Instances	controller | syslog | stderr | stdout	March 23, 2024 at 15:35	March 23, 2024 at 15:43	20 minutes, 50 seconds
□	⊞	s-04973623DXM0JKARANHQ	✓ Completed	SecondarySort-11-Instances	controller | syslog | stderr | stdout	March 23, 2024 at 15:35	March 23, 2024 at 15:35	7 minutes, 34 seconds
□	⊞	s-006041618P21M3R4U56H	✓ Completed	HCompute-6-Instances	controller | syslog | stderr | stdout	March 23, 2024 at 14:46	March 23, 2024 at 15:18	4 minutes, 34 seconds
□	⊞	s-08613452W2VLM3GSLV49	✓ Completed	HPopulate-6-Instances	controller | syslog | stderr | stdout	March 23, 2024 at 14:45	March 23, 2024 at 14:57	20 minutes, 51 seconds
□	⊞	s-001444526OC763EPYNJH	✓ Completed	SecondarySort-6-Instances	controller | syslog | stderr | stdout	March 23, 2024 at 14:44	March 23, 2024 at 14:44	12 minutes, 8 seconds

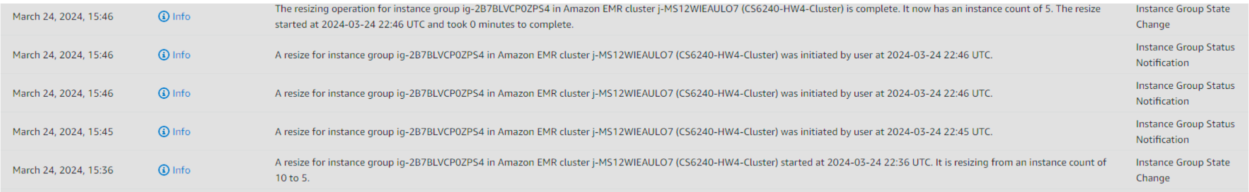


March 23, 2024, 15:33	ⓘ Info	The resizing operation for instance group ig-1IBGMUI13O025 in Amazon EMR cluster j-2R5Y13V5YHL1W (CS6240-HW4-Cluster) is complete. It now has an instance count of 10. The resize started at 2024-03-23 22:27 UTC and took 5 minutes to complete.	Instance Group State Change	-	ig-1IBGMUI13O025	Instance Group
March 23, 2024, 15:27	ⓘ Info	A resize for instance group ig-1IBGMUI13O025 in Amazon EMR cluster j-2R5Y13V5YHL1W (CS6240-HW4-Cluster) started at 2024-03-23 22:27 UTC. It is resizing from an instance count of 5 to 10.	Instance Group State Change	-	ig-1IBGMUI13O025	Instance Group
March 23, 2024, 15:27	ⓘ Info	A resize for instance group ig-1IBGMUI13O025 in Amazon EMR cluster j-2R5Y13V5YHL1W (CS6240-HW4-Cluster) was initiated by user at 2024-03-23 22:27 UTC.	Instance Group Status Notification	-	ig-1IBGMUI13O025	Instance Group

Added HBase input splits



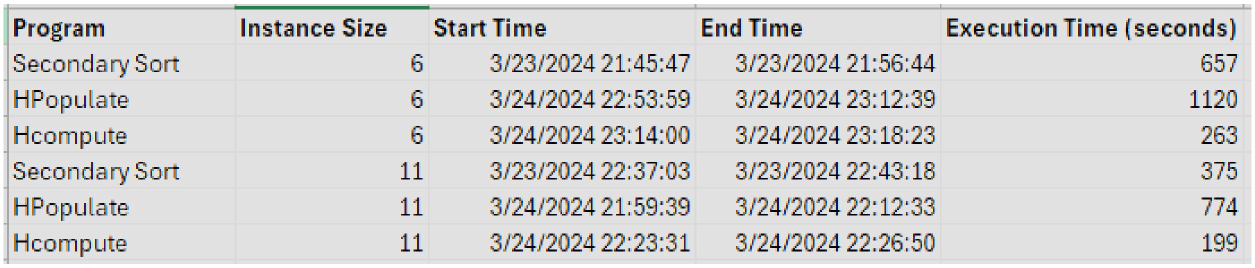
Step ID	Status	Name	Log files	Creation time (UTC-07:00)	Start time (UTC-07:00)	Elapsed time
s-05848241PRYXPMBIZZJY	Completed	HCompute-6-Instances	controller syslog stderr stdout	March 24, 2024 at 15:51	March 24, 2024 at 16:13	5 minutes, 34 seconds
s-08460781NBJ4963HO6NM	Completed	HPopulate-6-Instances	controller syslog stderr stdout	March 24, 2024 at 15:51	March 24, 2024 at 15:52	20 minutes, 36 seconds
s-0180712371YF2HT3WNQF	Completed	HCompute-11-Instances	controller syslog stderr stdout	March 24, 2024 at 15:21	March 24, 2024 at 15:22	4 minutes, 26 seconds
s-00600962PVM3SG10XIPS	Completed	HPopulate-11-Instances	controller syslog stderr stdout	March 24, 2024 at 14:58	March 24, 2024 at 14:58	14 minutes, 25 seconds



March 24, 2024, 15:46	Info	The resizing operation for instance group ig-2B7BLVCP0ZPS4 in Amazon EMR cluster j-MS12WIEAULO7 (CS6240-HW4-Cluster) is complete. It now has an instance count of 5. The resize started at 2024-03-24 22:46 UTC and took 0 minutes to complete.	Instance Group State Change
March 24, 2024, 15:46	Info	A resize for instance group ig-2B7BLVCP0ZPS4 in Amazon EMR cluster j-MS12WIEAULO7 (CS6240-HW4-Cluster) was initiated by user at 2024-03-24 22:46 UTC.	Instance Group Status Notification
March 24, 2024, 15:46	Info	A resize for instance group ig-2B7BLVCP0ZPS4 in Amazon EMR cluster j-MS12WIEAULO7 (CS6240-HW4-Cluster) was initiated by user at 2024-03-24 22:46 UTC.	Instance Group Status Notification
March 24, 2024, 15:45	Info	A resize for instance group ig-2B7BLVCP0ZPS4 in Amazon EMR cluster j-MS12WIEAULO7 (CS6240-HW4-Cluster) was initiated by user at 2024-03-24 22:45 UTC.	Instance Group Status Notification
March 24, 2024, 15:36	Info	A resize for instance group ig-2B7BLVCP0ZPS4 in Amazon EMR cluster j-MS12WIEAULO7 (CS6240-HW4-Cluster) started at 2024-03-24 22:36 UTC. It is resizing from an instance count of 10 to 5.	Instance Group State Change

Performance Comparison

Here I'm comparing the results between using HBase vs MapReduce HDFS.



Program	Instance Size	Start Time	End Time	Execution Time (seconds)
Secondary Sort	6	3/23/2024 21:45:47	3/23/2024 21:56:44	657
HPopulate	6	3/24/2024 22:53:59	3/24/2024 23:12:39	1120
Hcompute	6	3/24/2024 23:14:00	3/24/2024 23:18:23	263
Secondary Sort	11	3/23/2024 22:37:03	3/23/2024 22:43:18	375
HPopulate	11	3/24/2024 21:59:39	3/24/2024 22:12:33	774
Hcompute	11	3/24/2024 22:23:31	3/24/2024 22:26:50	199

HPopulate

An interesting thing that I found when running HPopulate is that it takes quite a long time to populate the data, despite me using buffered write and only opening/closing connection towards HBase once for every map instance. When testing with either 6 or 11 instances on EMR (m1.large), I find that it hovers around 15 minutes to read data from S3 and write to HBase region server. The HBase table size is also larger than the original data in HDFS (5.03Gb vs 4Gb). This can be explained by me creating redundant columns, as well as a complex key structure to uniquely differentiate flight records).

What I'm doing is to pre-define table splits based on the alphabet. The reason is based on the way I'm constructing the keys (airline-month-date-flightnumber-origin). Airlines are a string with two characters. Thus, I believe that separating them this way would yield a better performance. I tried running the program with and without the pre-defined split conditions and it yields a better performance (15 minutes with vs 20 minutes without). Interestingly, the result stays the same regardless of the cluster size.

Additionally, when running 6 instances vs 11 instances, there is a reduction in the runtime of HPopulate (20 minutes vs 15 minutes)

HCompute vs Secondary Sort

When strictly comparing reading from HBase vs HDFS, seems like the HBase scan with filter functionality performs better, given that I created redundant columns for filtering effort, which reduces the number of data being transferred between the region server and the map instance, thus reduce the number of input records in the map task (13,395,076 for Secondary Sort vs 5,824,423 for HCompute).

However, considering the additional effort of setting up HPopulate and the time it takes to write data to HBase, I'd still consider the Secondary Sort approach, unless there is a use case where after the data is put into HBase, I can spin up multiple programs to read and create different outputs.

Generally, the time trade-off between the two approaches is as follow:

- **Secondary sort:** # of compute tasks (**α**)* runtime per task (a)
- **HBase:** HPopulate runtime (b) + # of compute tasks(**α**) * runtime per compute task (c)

If there is large enough number of compute task (**α**) to makes the HBase method faster, HBase might worth the extra development effort.