

Project Final Report

Analysis Goals:

- Analysis Task – “Finding the busiest airport for each quarter for the years ranging from 2000 to 2008 in the dataset”.
 1. Main Task – **Passenger Count** – Compute the count of number of passengers to an airport for each quarter of the year. We used secondary sort to sort the input records of Origin Destination survey dataset by year & quarter.
 2. Main Task – “Finding the average delay of flights for all the busiest airports for the respective year & quarter determined in the above Analysis task”
 - Helper Task – **TransportPopulate** - Populate the On-time performance dataset into the HBase table to perform the equi-join.
 - Helper Task – **Average Delay calculation** – Compute the average delay for the busiest airports for each quarter of the years ranging from 2000 to 2008. We used HBase as index to store the On-time performance dataset. We performed an equi-join of the On-time performance dataset and the result of the previous analysis task.
- Analysis Task – “Finding the best airline for the year 2007 based on a comprehensive ranking system of each airline by finding largest number of air routes, longest distance covered, lowest delay time and lowest delay percentage for each airline.”
 1. Main Task – **Best Airline** – Finding the best airline for the year 2007 by computing the average rank for each airline based on the below mentioned categories and reporting a 10-element-rank from highest to lowest.
 - Helper Task – **Largest Number of Air Routes** – Finding the number of air routes for each airline and selecting top 10 such airlines with the largest count.
 - Helper Task – **Longest Distance** – Finding the total distances traveled of each airline and reporting a 10-element-rank
 - Helper Task – **Lowest Delay Time** – Finding the total delay time of each airline and reporting a 10-element-rank from lowest to highest.
 - Helper Task – **Lowest Delay Percentage** – Finding the number of total flights of each airline and the number of delayed flights to get percentages. Results will be a 10-element-list of ranks from lowest percentage to highest.
- Analysis Task – **StatisticalDelayCalculation** – “Finding the average delay based of metrics like - carrier, month, hour.”
 1. Main Task – **CarrierAvgDelay** – “Finding the average delay for each unique airline carrier”.
 - Helper Task - **CarrierAvgArrDelay** - Calculating the average delay time for each airline carrier based on arrival delay minutes.
 - Helper Task - **CarrierAvgDeptDelay** - Calculating the average delay time for each airline carrier based on departure delay minutes.
 2. Main Task – **HourlyAvgDelay** – “Finding the average delay for each hour of the day”.
 - Helper Task - **HourlyAvgArrDelay** - Calculating the average delay time for each hour of the day based on arrival delay minutes.
 - Helper Task - **HourlyAvgDeptDelay** - Calculating the average delay time for each hour of the day based on departure delay minutes.
 3. Main Task – **MonthlyAvgDelay** – “Finding the average delay for each month of the year”.
 - Helper Task - **MonthlyAvgArrDelay** - Calculating the average delay time for each month of the year 2007 based on arrival delay minutes.
 - Helper Task - **MonthlyAvgDeptDelay** - Calculating the average delay time for each month of the year 2007 based on departure delay minutes.

Data Description:

- The Transportation dataset is available on AWS as a public database under the “Economics” category. We have accessed this on AWS console using a AWS-snapshot-id (snap-37668b5e).
- Link: <https://aws.amazon.com/datasets/transportationdatabases/?tag=datasets%23keywords%23economics>
- The data is available as CSV format. This dataset has Aviation data for 20 years from the year 1988 to 2008. The size of the entire dataset is 15GB.
- We used 2 datasets under Aviation category which are
 1. Origin-Destination Survey Dataset - Data includes origin, destination and other itinerary details of passengers transported. This database is used to determine air traffic patterns, air carrier market shares and passenger flows.
 2. On-time performance Dataset - The U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS) tracks the on-time performance of domestic flights operated by large air carriers. Summary information on the number of on-time, delayed, canceled and diverted flights from the year 1988 to 2008 are available in this dataset.
- Sample data set:

Origin-Destination Survey dataset:

ItinID	MktID	SeqNum	Coupons	Year	Quarter	Origin	OriginAptlr	OriginCityA	OriginCoun	OriginState	OriginState	OriginState	OriginWac	Dest	DestAptlnd	DestCityNu
200825000000	200828000000	2	4	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 BUF		0	13742
200825000000	200828000000	3	4	2008	2	BUF	0	13742 US		36 NY	New York		22 PHL		1	69880
200825000000	200828000000	4	4	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 RDU		0	74830
200825000000	200828000000	1	4	2008	2	RDU	0	74830 US		37 NC	North Caro		36 PHL		1	69880
200825000000	200828000000	2	4	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 BUF		0	13742
200825000000	200828000000	3	4	2008	2	BUF	0	13742 US		36 NY	New York		22 PHL		1	69880
200825000000	200828000000	4	4	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 RDU		0	74830
200825000000	200828000000	1	4	2008	2	RDU	0	74830 US		37 NC	North Caro		36 PHL		1	69880
200825000000	200828000000	2	4	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 BUF		0	13742
200825000000	200828000000	3	4	2008	2	BUF	0	13742 US		36 NY	New York		22 PHL		1	69880
200825000000	200828000000	4	4	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 RDU		0	74830
200825000000	200828000000	1	2	2008	2	RDU	0	74830 US		37 NC	North Caro		36 PHL		1	69880
200825000000	200828000000	2	2	2008	2	PHL	1	69880 US		42 PA	Pennsylvan		23 BWI		0	7231

On-time Performance dataset:

Year	Quarter	Month	DayofMont	DayOfWeel	FlightDate	UniqueCarr	AirlineID	Carrier	TailNum	FlightNum	Origin	OriginCityA	OriginState	OriginState	OriginState	OriginWac	Dest	DestCityNa
1988	4	10	12	3	10/1/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	13	4	10/2/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	14	5	10/3/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	16	7	10/4/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	17	1	10/5/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	18	2	10/6/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	19	3	10/7/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	20	4	10/8/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles
1988	4	10	21	5	10/9/1988	UA	19977	UA		259	ORD	Chicago, IL	IL		17	Illinois	41 LAX	Los Angeles

Technical Discussion:

- Analysis Task – “Finding the busiest airport for each quarter for the years ranging from 2000 to 2008 in the dataset”. Our hypothesis is that the airports in the cities like San Francisco, New York, Los Angeles and Chicago are the busiest.
1. Main Task – **Passenger Count** – Compute the count of number of passengers to an airport for each quarter of the year. We used secondary sort to sort the input records of Origin Destination survey dataset by year & quarter.
 - To solve the problem, we need to sort the dataset on year, quarter and then by passenger count.
 - We used the secondary sort logic to sort by Year and Quarter. In the reduce task, for each pair of year and quarter we store the passenger count for each destination using a hashmap.
 - We find the destination with the maximum passenger count using the ValueComparator helper class. Thus, we emit the destination with maximum passenger count for each pair of year and quarter.

Pseudo Code:

```
Map(year, quarter, destination, passengerCount){
    emit(IntPair(Year, Quarter), (destination + passengerCount));
}

getPartition(IntPair(Year, Quarter), (destination + passengerCount)){
    return myPartition(Year);
}

keyComparator(IntPair(Year, Quarter), (destination + passengerCount)){
    // sort in increasing order of year first
    // If the year is equal sort in increasing order of quarter next
}

groupingComparator(IntPair(Year, Quarter), (destination + passengerCount)){
    // sort in increasing order of year first
    // Does not consider quarter for sorting
    // two keys with same year values are considered identical, irrespective of quarter value.
}

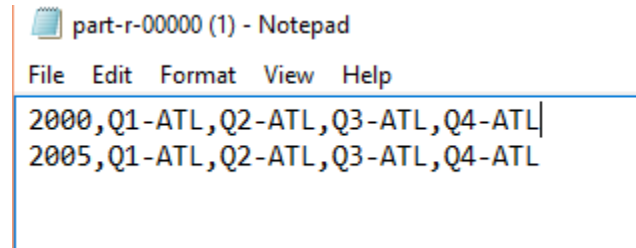
reduce(IntPair(Year, Quarter), [(destination1 + passengerCount1), (destination2 + passengerCount2),....] ){
    Map<destination, passengerCount> map;
    String result;
    For(all values v){
        If(prevQuarter = currentQuarter){
            If(map contains destination)
                map.value += passengerCount
            Else
                map.put(destination, passengerCount)
        }
        Else{
            // find the destination with max passenger count from map
            Result += (quarter, destination)
        }
    }

    emit (Null, result);
}
```

Reference File: BusiestAirport.java

Output format: year ,[<Quarter – Airport>....]

OUTPUT:



```
part-r-00000 (1) - Notepad
File Edit Format View Help
2000, Q1-ATL, Q2-ATL, Q3-ATL, Q4-ATL
2005, Q1-ATL, Q2-ATL, Q3-ATL, Q4-ATL
```

- The Above output shows that busiest airport for each quarter reported for the year 2000 and 2005 is ATL.
- Similarly, we collected the result for years 2000-2008 and found ATL as the busiest airport.

RunTime Analysis:

Task	Time taken in cluster of 5 machines	Time taken of 5 reduce tasks (locally)	Time taken of 10 reduce tasks (locally)
BusiestAirport	1738 seconds	1577 seconds	1643 seconds

Performance Analysis:

Cluster with 5 machines - 1 Master - m1.large
4 Cores - m1.large

- With more number of reduce tasks, I/O operations are increased as we need to create more files as each reducer create its own file.
- Each reduce need to start up and be created/instantiated in the nodes, which result in an increase of startup time. Also, data need to be split across the entire number of reducers which require more network transfer time and parsing time. Thus time taken with 10 reduce tasks > time taken by 5 reduce tasks.
- We ran the above task on a Cluster of 5 machines (1 master and 4 core) each of which were of type m1.large

2. **Main Task** – “Finding the average delay of flights for all the busiest airports for the respective year & quarter determined in the above Analysis task”

- **Helper Task – TransportPopulate** - Populate the On-time performance dataset in to the HBase table to perform the equi-join.

PseudoCode:

```
map(key, value){

    // Load the data into HBase table "transportTable" as below

    rowkey = Year + '%' + Quarter + '%' + Count;

    Put p(rowKey);

    p.columnFamily[transportData].add("originCity", Origin);

    p.columnFamily[transportData].add("destinationCity", Destination);

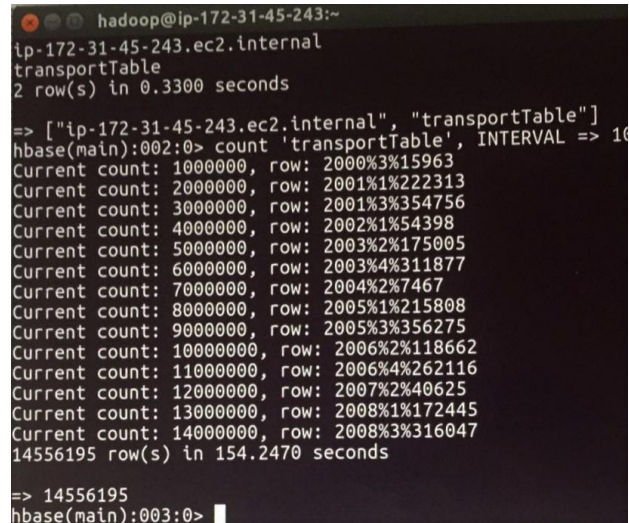
    p.columnFamily[transportData].add("delayMin", Delay);

    emit(rowKey,p);

}
```

Reference File: Transportpopulate.java

Output :



```
hadoop@ip-172-31-45-243:~
ip-172-31-45-243.ec2.internal
transportTable
2 row(s) in 0.3300 seconds

=> ["ip-172-31-45-243.ec2.internal", "transportTable"]
hbase(main):002:0> count 'transportTable', INTERVAL => 10
Current count: 1000000, row: 2000%3%15963
Current count: 2000000, row: 2001%1%222313
Current count: 3000000, row: 2001%3%354756
Current count: 4000000, row: 2002%1%54398
Current count: 5000000, row: 2003%2%175005
Current count: 6000000, row: 2003%4%311877
Current count: 7000000, row: 2004%2%7467
Current count: 8000000, row: 2005%1%215808
Current count: 9000000, row: 2005%3%356275
Current count: 10000000, row: 2006%2%118662
Current count: 11000000, row: 2006%4%262116
Current count: 12000000, row: 2007%2%40625
Current count: 13000000, row: 2008%1%172445
Current count: 14000000, row: 2008%3%316047
14556195 row(s) in 154.2470 seconds

=> 14556195
hbase(main):003:0>
```

The above output screenshot shows the HBase -transportTable loaded for the years 2000-2008 of OnTime Performance dataset. The total number of rows reported are 14556195.

RunTime Analysis:

Task	Time taken in cluster of 5 machines
TransportPopulate	2620 seconds

Performance Analysis:

Cluster with 5 machines - 1 Master - m1.large
4 Cores - m1.large

- **Helper Task – Average Delay calculation** – Compute the average delay for the busiest airports for each quarter of the years ranging from 2000 to 2008. We performed an equi-join of the On-time performance dataset and the result of the previous analysis task (PassengerCount).
 - We added the result from the PassengerCount task to an arraylist <String> - busiestAirport
 - For each year, quarter and destination from the arraylist we find the average delay from the transportTable in HBase.

PseudoCode:

```

Class Transportmapper{
    ArrayList<String> busiestAirport;
    setup(context){
        for(each output file f){
            while( line = f.readline is not null)
                busiestAirport.add(line);
        }
    }
    map(key, value){
        // get the data from HBase table: "transportTable"

        for(each v in busiestAirport){
            if(value.year = v.year){
                for(each quarter q in v){
                    if(value.quarter = q. quarter && value.dest = q.dest)
                        emit((year, quarter), (delay, dest));
                }
            }
        }
    }
}

getPartition(IntPair(Year, Quarter), (delay, dest)){
    return myPartition(Year);
}

groupingComparator(IntPair(Year, Quarter), (destination + passengerCount)){
    // sort in increasing order of year first
    // Does not consider quarter for sorting
    // two keys with same year values are considered identical, irrespective of quarter
}

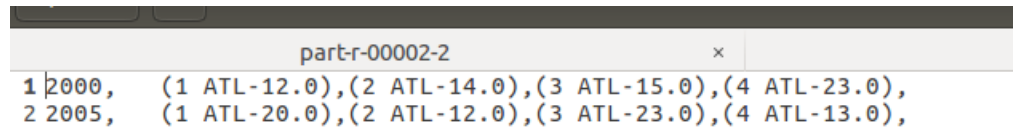
reduce(IntPair(Year, Quarter), [(delay1, destination1), (delay2, dest2),....] ){
    result="";
    for(all values v){
        If(prevQuarter = currentQuarter){
            DelaySum+=delay;
            flightCount++;
        }else{
            avgDelay = DelaySum/flighgtCount;
            result +=(quarter, dest, delay);
        }
    }
    emit(year, result);
}

```

Reference File: TransportCompute.java

Output format:year ,[<Quarter-Airport,Delay>....]

OUTPUT:



part-r-00002-2				
1 2000,	(1 ATL-12.0),	(2 ATL-14.0),	(3 ATL-15.0),	(4 ATL-23.0),
2 2005,	(1 ATL-20.0),	(2 ATL-12.0),	(3 ATL-23.0),	(4 ATL-13.0),

- The Above output shows that delay for the busiest airport of each quarter reported for the year 2000 and 2005.
- Similarly, we collected the result for years 2000-2008 and collected the result for all the busiest airport.

RunTime Analysis:

Task	Time taken in cluster of 5 machines	Time taken of 5 reduce tasks (locally)	Time taken of 10 reduce tasks (locally)	Time taken of 10 reduce tasks (locally) with load balancing
Transport Compute	168 seconds	439 seconds	493 seconds	413 seconds

Performance Analysis:

Cluster with 5 machines - 1 Master - m1.large
5 Cores - m1.large

- 10 reduce tasks took more time than 5 because of the shuffle overhead.
- Too many reduces adversely affects the shuffle crossbar. Also, in extreme cases it results in too many small files created as the output of the job — this hurts both the NameNode and performance of subsequent Map-Reduce applications who need to process lots of small files.
- Load Balancing Technique:
 - The Region-Split policy used was 'KeyPrefixRegionSplitPolicy'. This is a split policy already made available by HBase.
 - This SplitPolicy groups rows by a prefix of the row-key This ensures that a region is not split "inside" a prefix of a row key. i.e. rows can be co-located in a region by their prefix. "prefix_split_key_policy.prefix_length" attribute of the table defines the prefix length. We have used this length to be 4. Assuming the year is of length 4 in the provided dataset. So, all records with the same year are guaranteed to be collocated for faster sorting purpose. Thus, the time taken for 10 reduce tasks when using region split policy is better than that of 5 reduce tasks.

- **Analysis Task** – “Finding the best airline for the year 2007 based on a comprehensive ranking system of each airline by finding largest number of air routes, longest distance covered, lowest delay time and lowest delay percentage for each airline.”

1. **Main Task – Best Airline** – Finding the best airline for the year 2007 by computing the average rank for each airline based on the below mentioned categories and reporting a 10-element-rank from highest to lowest.

PseudoCode:

```
map(key,value){
    //rownumber indicates the rank
    emit(year, rank)
}

Class Reducer{
    Map<carrier, rank> map;

    reduce(year, [rank1, rank2.....]){
        rankSum =0;;
        count = 0;
        for (each rank r){
            count++;
            rankSum+=r;
        }
        while(count!=4){
            count++;
            rankSum+=11;
        }
        avgRank = rankSum / count;

        map.put(carrier, avgRank);
    }
    cleanup(context){
        //sort the map by value

        For(i=1.....10){
            emit(carrier, avgRank);
        }
    }
}
```

Reference File: BestAirline.java

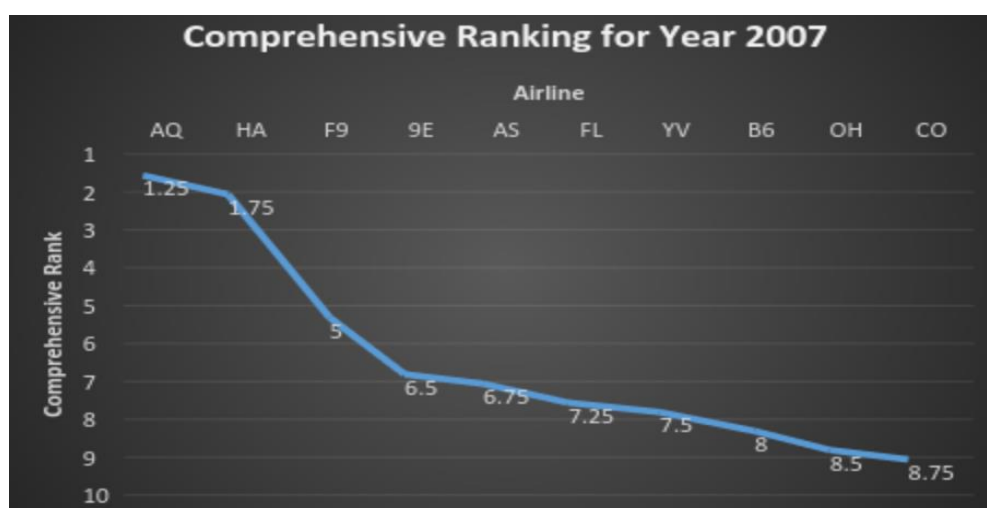
Output Format: <Carrier,Cumulative Rank>

Output:

File	Edit	Format	View	Help
AQ	1.2			
SHA	1.75			
F9	5.0			
9E	6.5			
AS	6.75			
FL	7.25			
YV	7.5			
B6	8.0			
OH	8.5			
CO	8.75			

→ The above image shows the comprehensive rank of each airline based on the individual rankings obtained from the helper tasks explained below.

Graph:



RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
BestAirline	34 seconds	32 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge

Cluster with 11 machines - 1 Master - m3.xlarge
10 Cores - m3.xlarge

→ Since the number of reduce tasks remains as 1 in both the cluster configurations we do not get a marginal difference between the execution time of both the versions. Also, the number of input splits were 4 in both the cases. Thus, this program does not scale well.

- **Helper Task – Largest Number of Air Routes** – Finding the number of air routes for each airline and selecting top 10 such airlines with the largest count.

- A route should be considered as a pair of two different cities. For example, flight from Seattle to Vancouver should be considered as a route. But the flight from Vancouver to Seattle should be considered as a different flight. We count the total number of flights of each airline and report 10 airlines that have largest number of routes in descending order.

PseudoCode:

```
map(key,value){
    emit(carrier, origin+~+dest);
}

Class reducer{

Map<carrier, numberOfAirroutes> map;

reduce(carrier, [origin1+~+dest1, origin2+~+dest2,...]){
    HashSet< origin+~+dest> set;
    For(each value v){
        Set.add(v);
    }
    map.put(carrier,set.size());
}

cleanup(context){
    //sort the map by value

    For(i=1.....10){
        emit(map.key(), map.value());
    }
}
}
```

Reference file: LargestAirRoutes.java

Output Format: <Carrier, Number of Air Routes>

Output:

part-r-00002-2		
1	AQ	34.0
2	HA	40.0
3	F9	111.0
4	AS	169.0
5	B6	232.0
6	CO	330.0
7	FL	333.0
8	UA	411.0
9	MQ	430.0
10	YV	459.0

- The above screenshot shows the Unique Carrier with their respective number of air routes sorted in ascending order.

RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
LongestAirRoute	80 seconds	58 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge

Cluster with 11 machines - 1 Master - m3.xlarge
10 Cores - m3.xlarge

```

Job Counters
  Killed map tasks=1
  Launched map tasks=39
  Launched reduce tasks=1
  Data-local map tasks=39
  Total time spent by all maps in occupied slots (ms)=
33683760
  Total time spent by all reduces in occupied slots
(ms)=3196620
  Total time spent by all map tasks (ms)=748528
  Total time spent by all reduce tasks (ms)=35518
  Total vcore-milliseconds taken by all map tasks=
748528
  Total vcore-milliseconds taken by all reduce tasks=
35518

```

6 machines

```

Job Counters
    Killed map tasks=1
    Launched map tasks=39
    Launched reduce tasks=1
    Data-local map tasks=39
    Total time spent by all maps in occupied slots (ms)=
34853670
    Total time spent by all reduces in occupied slots
(ms)=1956150
    Total time spent by all map tasks (ms)=774526
    Total time spent by all reduce tasks (ms)=21735
    Total vcore-milliseconds taken by all map tasks=
774526
    Total vcore-milliseconds taken by all reduce tasks=
21735
    Total megabyte-milliseconds taken by all map tasks=
1115317440

```

11 machines

The shuffle phase overhead adds up to the extra time in 5 machine cluster resulting in increased time taken in the reduce task.

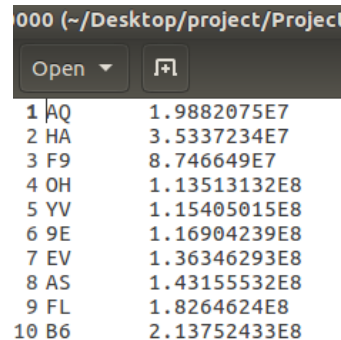
- **Helper Task – Longest Distance** – Finding the total distances traveled of each airline and reporting a 10-element-rank

- We add up all the distance traveled of each airline and report 10 airlines that have largest traveled distance in descending order.
- The pseudo code for this task is like the **Largest Number of Air Routes** helper task discussed above but we consider distance instead of routes.

Reference File: LongestDistance.java

Output Format: <Carrier, Distance>

Output:



```
000 (~/Desktop/project/Project)
Open [icon]
1 AQ      1.9882075E7
2 HA      3.5337234E7
3 F9      8.746649E7
4 OH      1.13513132E8
5 YV      1.15405015E8
6 9E      1.16904239E8
7 EV      1.36346293E8
8 AS      1.43155532E8
9 FL      1.8264624E8
10 B6     2.13752433E8
```

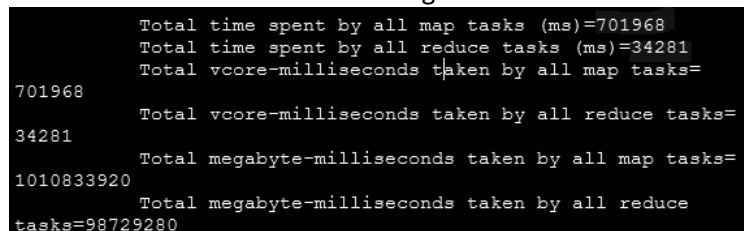
→ The above screenshot shows the distance covered by each carrier sorted in ascending order.

RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
LongestDistance	76 seconds	54 seconds

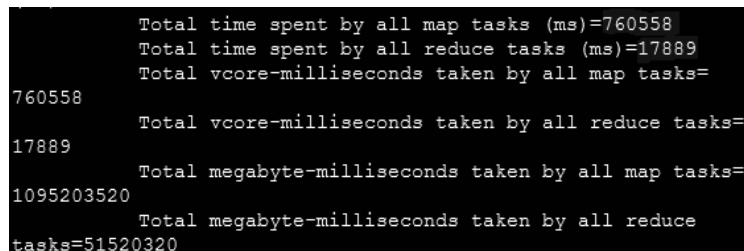
Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge Cluster with 11 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge 10 Cores - m3.xlarge



```
Total time spent by all map tasks (ms)=701968
Total time spent by all reduce tasks (ms)=34281
Total vcore-milliseconds taken by all map tasks=
701968
Total vcore-milliseconds taken by all reduce tasks=
34281
Total megabyte-milliseconds taken by all map tasks=
1010833920
Total megabyte-milliseconds taken by all reduce
tasks=98729280
```

6 machines



```
Total time spent by all map tasks (ms)=760558
Total time spent by all reduce tasks (ms)=17889
Total vcore-milliseconds taken by all map tasks=
760558
Total vcore-milliseconds taken by all reduce tasks=
17889
Total megabyte-milliseconds taken by all map tasks=
1095203520
Total megabyte-milliseconds taken by all reduce
tasks=51520320
```

11 machines

The shuffle phase overhead adds up to the extra time in 5 machine cluster resulting in increased time taken in the reduce task.

- **Helper Task – Lowest Delay Time** – Finding the total delay time of each airline and reporting a 10-element-rank from lowest to highest.
 - Add up all the time gap of flights of each airline and report 10 airlines that have lowest total delaying time in ascending order.
 - The pseudo code for this task is like the **Largest Number of Air Routes** helper task discussed above but we consider arrival delay instead of routes.

Reference File: LowestDelayTime.java

Output Format: <Carrier, Delay>

Output:

	part-r-00000	
1	AQ	188889.0
2	HA	237668.0
3	F9	1120732.0
4	AS	2343686.0
5	9E	3455423.0
6	FL	3484648.0
7	B6	3811938.0
8	OH	4036629.0
9	YV	4645876.0
10	CO	5226285.0
11		

→ The above screenshot shows the delay time of each carrier sorted in ascending order.

RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
LowestDelayTime	84 seconds	58 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge

Cluster with 11 machines - 1 Master - m3.xlarge
10 Cores - m3.xlarge

→ The performance comparison is similar to the above case [LongestDistance].

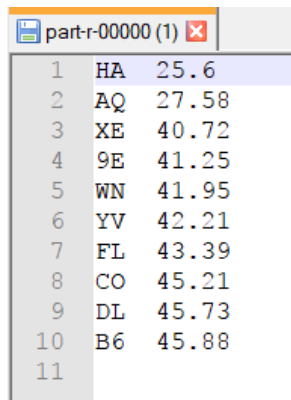
- **Helper Task – Lowest Delay Percentage** – Finding the number of total flights of each airline and the number of delayed flights to get percentages. Results will be a 10-element-list of ranks from lowest percentage to highest.

- First, we count the total number of flights of each airline and the number of delayed flights among the total flights. Then, we divided the number of delayed flights by the total number of flights to get percentages. To report the results, we sorted the list of results in ascending order and reported the first 10 of them.
- The pseudo code for this task is like the **Largest Number of Air Routes** helper task discussed above but we consider arrival delay instead of routes.

Reference File: LowestDelayPercentage.java

Output Format:<Carrier,DelayPercentage>

Output:



1	HA	25.6
2	AQ	27.58
3	XE	40.72
4	9E	41.25
5	WN	41.95
6	YV	42.21
7	FL	43.39
8	CO	45.21
9	DL	45.73
10	B6	45.88
11		

→ The above screenshot shows each carrier with delay percentage sorted in ascending order.

RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
LowestDelayPercentage	82 seconds	64 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge Cluster with 11 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge 10 Cores - m3.xlarge

→ The performance comparison is similar to the above case [LongestDistance].

- **Analysis Task – StatisticalDelayCalculation** – “Finding the average delay based on metrics like - carrier, month, hour.”

1. **Main Task – CarrierAvgDelay** – “Finding the average delay for each unique airline carrier”.

PseudoCode:

```
map(key,value){
    emit(carrier, delay);
}

reduce(carrier, [delay1,delay2,...]){
    count = 0;
    delaySum =0.0;
    for(each delay d){
        count++;
        delaySum += d;
    }

    avgDelay = delaySum / count;

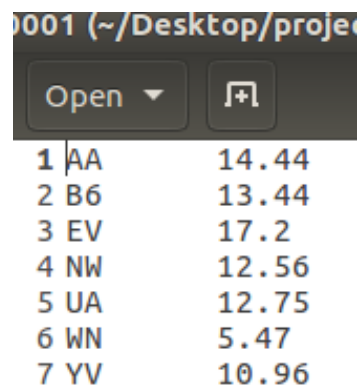
    emit(carrier, avgDelay);
}
```

- **Helper Task - CarrierAvgArrDelay** - Calculating the average delay time for each airline carrier based on arrival delay minutes.

Reference file: CarrierArrDelay.java

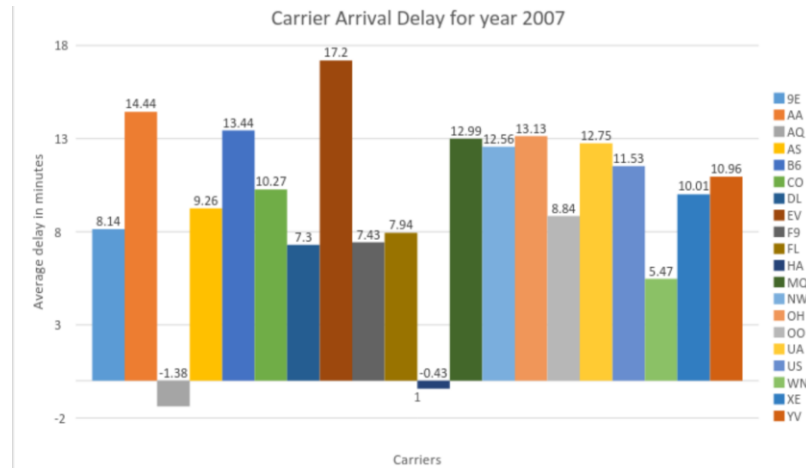
Output Format: <Carrier, Delay>

Output:



1	AA	14.44
2	B6	13.44
3	EV	17.2
4	NW	12.56
5	UA	12.75
6	WN	5.47
7	YV	10.96

Graph:



RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
CarrierArrAvgDelay	90 seconds	62 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge Cluster with 11 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge 10 Cores - m3.xlarge

```
Total time spent by all maps in occupied slots (ms)=33328665
Total time spent by all reduces in occupied slots (ms)=13509000
Total time spent by all map tasks (ms)=740637
Total time spent by all reduce tasks (ms)=150100
Total vcore-milliseconds taken by all map tasks=740637
Total vcore-milliseconds taken by all reduce tasks=150100
Total megabyte-milliseconds taken by all map tasks=1066517280
Total megabyte-milliseconds taken by all reduce tasks=432288000
```

6 machines

```
Total time spent by all maps in occupied slots (ms)=38321550
Total time spent by all reduces in occupied slots (ms)=12658410
Total time spent by all map tasks (ms)=851590
Total time spent by all reduce tasks (ms)=140649
Total vcore-milliseconds taken by all map tasks=851590
Total vcore-milliseconds taken by all reduce tasks=140649
Total megabyte-milliseconds taken by all map tasks=1226289600
Total megabyte-milliseconds taken by all reduce tasks=405069120
```

11 machines

- The time taken by cluster of 6 machines is more since there are only 5 reduce tasks. Thus, the reduce phase is slower in comparison to that of cluster of 11 machines which has 10 reduce tasks. The above screenshots show the time spent by each cluster in respective reduce phase.

- **Helper Task - CarrierAvgDeptDelay** - Calculating the average delay time for each airline carrier based on departure delay minutes.

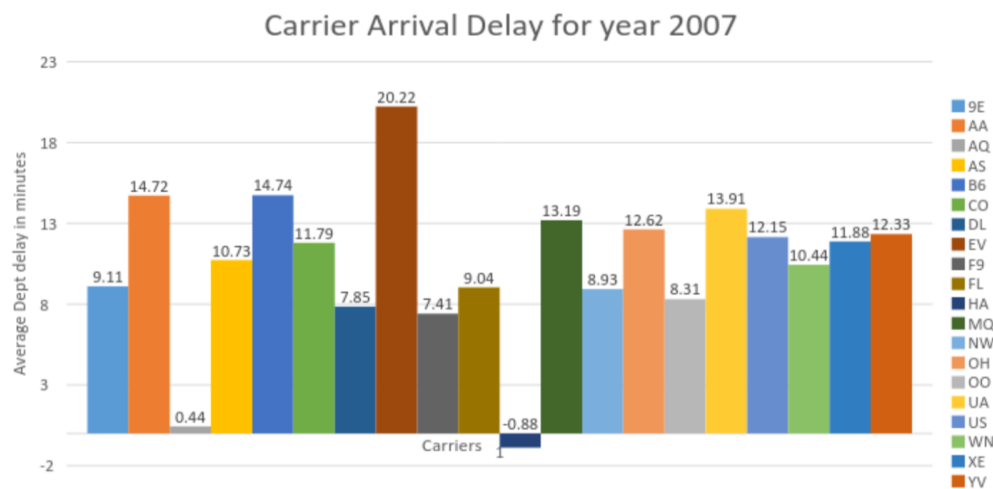
Reference file: CarrierDeptDelay.java

Output Format: <Carrier, Number of Air Routes>

Output:

File (Edit View Search)		
Open ▾		
1	AA	14.72
2	B6	14.74
3	EV	20.22
4	NW	8.93
5	UA	13.91
6	WN	10.44
7	YV	12.33

Graph:



RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
CarrierDeptAvgDelay	82 seconds	64 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge Cluster with 11 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge 10 Cores - m3.xlarge

```
Total time spent by all maps in occupied slots (ms)=31711140
Total time spent by all reduces in occupied slots (ms)=11816550
Total time spent by all map tasks (ms)=704692
Total time spent by all reduce tasks (ms)=131295
Total vcore-milliseconds taken by all map tasks=704692
Total vcore-milliseconds taken by all reduce tasks=131295
Total megabyte-milliseconds taken by all map tasks=1014756480
Total megabyte-milliseconds taken by all reduce tasks=378129600
```

6 machines

```
Total time spent by all maps in occupied slots (ms)=37091340
Total time spent by all reduces in occupied slots (ms)=17107290
Total time spent by all map tasks (ms)=824252
Total time spent by all reduce tasks (ms)=190081
Total vcore-milliseconds taken by all map tasks=824252
Total vcore-milliseconds taken by all reduce tasks=190081
Total megabyte-milliseconds taken by all map tasks=1186922880
Total megabyte-milliseconds taken by all reduce tasks=547433280
```

11 machines

- The time taken by cluster of 6 machines is more since there are only 5 reduce tasks. Thus, the reduce phase is slower in comparison to that of cluster of 11 machines which has 10 reduce tasks. The above screenshots show the time spent by each cluster in respective reduce phase.

2. **Main Task – HourlyAvgDelay** – “Finding the average delay for each hour of the day”.

PseudoCode:

```
map(key,value){
    emit(hour, delay);
}

reduce(carrier, [delay1,delay2,...]){
    count = 0;
    delaySum =0.0;
    for(each delay d){
        count++;
        delaySum += d;
    }

    avgDelay = delaySum / count;

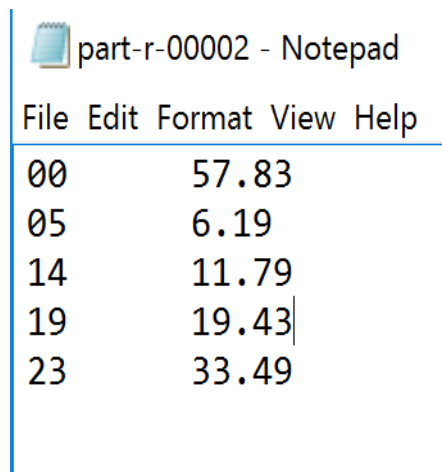
    emit(hour, avgDelay);
}
```

- **Helper Task - HourlyAvgArrDelay** - Calculating the average delay time for each hour of the day based on arrival delay minutes.

Reference file: HourlyArrDelay.java

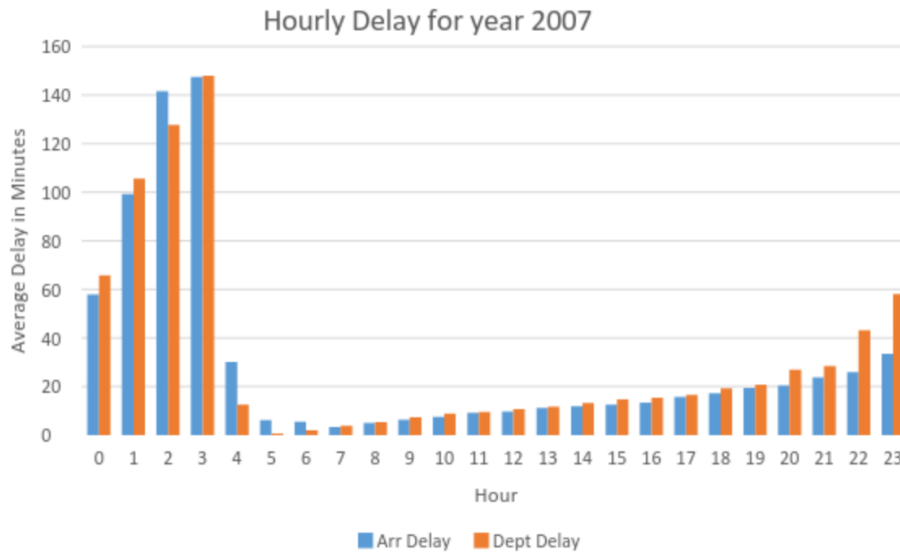
Output Format: <Hour, Delay>

Output:



part-r-00002 - Notepad	
File	Edit Format View Help
00	57.83
05	6.19
14	11.79
19	19.43
23	33.49

Graph:



RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
HourlyArrAvgDelay	100 seconds	58 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge Cluster with 11 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge 10 Cores - m3.xlarge

```
Total time spent by all maps in occupied slots (ms)=37967400
Total time spent by all reduces in occupied slots (ms)=22123170
Total time spent by all map tasks (ms)=843720
Total time spent by all reduce tasks (ms)=245813
Total vcore-milliseconds taken by all map tasks=843720
Total vcore-milliseconds taken by all reduce tasks=245813
Total megabyte-milliseconds taken by all map tasks=1214956800
Total megabyte-milliseconds taken by all reduce tasks=707941440
```

6 machines

```
Total time spent by all maps in occupied slots (ms)=36009855
Total time spent by all reduces in occupied slots (ms)=11874150
Total time spent by all map tasks (ms)=800219
Total time spent by all reduce tasks (ms)=131935
Total vcore-milliseconds taken by all map tasks=800219
Total vcore-milliseconds taken by all reduce tasks=131935
Total megabyte-milliseconds taken by all map tasks=1152315360
Total megabyte-milliseconds taken by all reduce tasks=379972800
```

11 machines

→ The performance comparison is same as the previous tasks. (CarrierDelay tasks).

3. **Main Task – MonthlyAvgDelay** – “Finding the average delay for each month of the year”.

PseudoCode:

```
map(key,value){
    emit(month, delay);
}

reduce(carrier, [delay1,delay2,...]){
    count = 0;
    delaySum =0.0;
    for(each delay d){
        count++;
        delaySum += d;
    }

    avgDelay = delaySum / count;

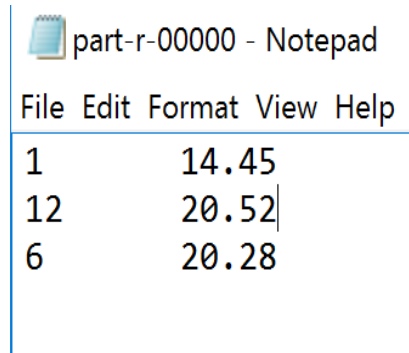
    emit(month, avgDelay);
}
```

- **Helper Task - MonthlyArrAvgDelay** - Calculating the average delay time for each month of the year 2007 based on arrival delay minutes.

Reference file: MonthlyArrDelay.java

Output Format: <Month, Delay>

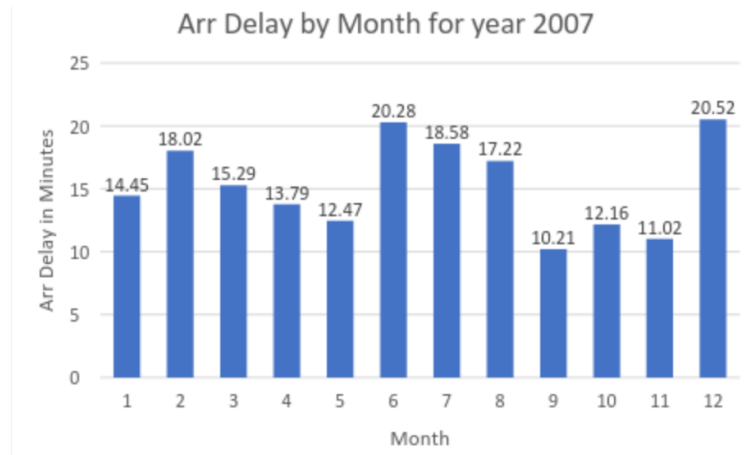
Output:



File Edit Format View Help	
1	14.45
12	20.52
6	20.28

→ The above screenshot shows months January, December and June with arrival delay.

Graph:



RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
MonthlyArrAvgDelay	84 seconds	62 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge
5 Cores - m3.xlarge

Cluster with 11 machines - 1 Master - m3.xlarge
10 Cores - m3.xlarge

```
Launched reduce tasks=5
Data-local map tasks=39
Total time spent by all maps in occupied slots (ms)=31857930
Total time spent by all reduces in occupied slots (ms)=13744710
Total time spent by all map tasks (ms)=707954
Total time spent by all reduce tasks (ms)=152719
Total vcore-milliseconds taken by all map tasks=707954
Total vcore-milliseconds taken by all reduce tasks=152719
Total megabyte-milliseconds taken by all map tasks=1019453760
Total megabyte-milliseconds taken by all reduce tasks=439830720
```

6 machines

```
Launched reduce tasks=10
Data-local map tasks=40
Total time spent by all maps in occupied slots (ms)=35752410
Total time spent by all reduces in occupied slots (ms)=17812620
Total time spent by all map tasks (ms)=794498
Total time spent by all reduce tasks (ms)=197918
Total vcore-milliseconds taken by all map tasks=794498
Total vcore-milliseconds taken by all reduce tasks=197918
Total megabyte-milliseconds taken by all map tasks=1144077120
Total megabyte-milliseconds taken by all reduce tasks=570003840
```

11 machines

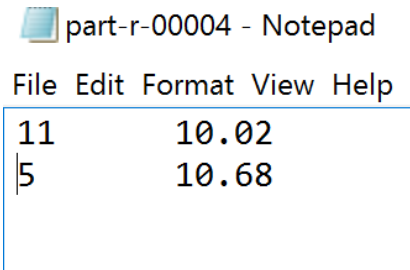
→ The performance comparison is same as the previous tasks. (CarrierDelay tasks).

- **Helper Task - MonthlyDeptAvgDelay** - Calculating the average delay time for each month of the year 2007 based on departure delay minutes.

Reference file: MonthlyDeptDelay.java

Output Format: <Month, Delay>

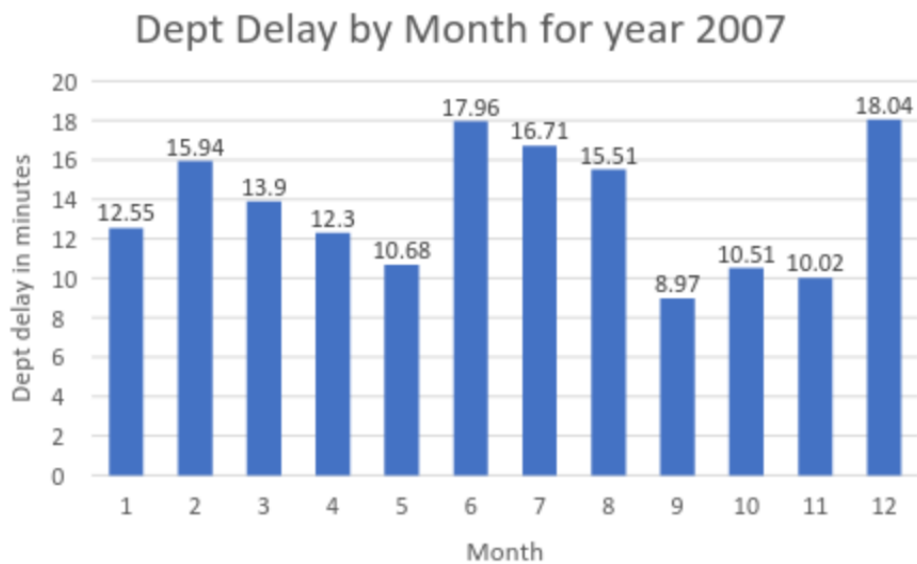
Output:



```
part-r-00004 - Notepad
File Edit Format View Help
11 10.02
5 10.68
```

→ The above screenshot shows months May and November with departure delay.

Graph:



RunTime Analysis:

Task	Time taken in cluster of 6 machines	Time taken in cluster of 11 machines
MonthlyDeptAvgDelay	78 seconds	60 seconds

Performance Analysis:

Cluster with 6 machines - 1 Master - m3.xlarge 5 Cores - m3.xlarge	Cluster with 11 machines - 1 Master - m3.xlarge 10 Cores - m3.xlarge
---	---

```
Launched reduce tasks=5
Data-local map tasks=39
Total time spent by all maps in occupied slots (ms)=30414330
Total time spent by all reduces in occupied slots (ms)=10356120
Total time spent by all map tasks (ms)=675874
Total time spent by all reduce tasks (ms)=115068
Total vcore-milliseconds taken by all map tasks=675874
Total vcore-milliseconds taken by all reduce tasks=115068
Total megabyte-milliseconds taken by all map tasks=973258560
Total megabyte-milliseconds taken by all reduce tasks=331395840
```

6 machines

```
Launched reduce tasks=10
Data-local map tasks=40
Total time spent by all maps in occupied slots (ms)=35701470
Total time spent by all reduces in occupied slots (ms)=14662620
Total time spent by all map tasks (ms)=793366
Total time spent by all reduce tasks (ms)=162918
Total vcore-milliseconds taken by all map tasks=793366
Total vcore-milliseconds taken by all reduce tasks=162918
Total megabyte-milliseconds taken by all map tasks=1142447040
Total megabyte-milliseconds taken by all reduce tasks=469203840
```


11 machines

The performance comparison is same as the previous tasks. (CarrierDelay tasks).

Setup challenges:

- The Transportation dataset snapshot for the linux drive was corrupted and we were unable to mount the drive to EC2 instance. So, we had to mount the dataset snapshot of windows drive (NTFS format) to the EC2 instance. The windows drive doesn't provide permissions to extract the the data present in the ZIP format.

Details



Submitted By: [Santiago@AWS](#)

US Snapshot ID: snap-e1608d88
(Linux/Unix):

**US Snapshot ID: snap-37668b5e
(Windows):**

Size: 15GB

Source: The Bureau of Transportation Services

Created On: April 1, 2009 11:25 PM GMT

Last Updated: June 4, 2009 8:26 PM GMT

- To resolve this, first we had to copy the zip files to one of the local mounted drives of the EC2 instance and then change the owner of those files to hadoop user. Then we extract the data and then move the data (45GB + 17GB) into the HDFS so that we can access it in our MapReduce program. We had to repeat the same process for both the dataset we are using in this project.
- Also, our dataset is present in North Virginia region and we don't have m3x.large machines in this region. We had to use the m1.large machines to the EMR job. Only 5 instances of m1.large machines are allowed to be used in the North Virginia region. So, we had to run our job with only 5 machines for our BusiestAirport. We can't copy this huge data to S3 bucket because for size restrictions.
- For the other tasks, since we are considering only 2007 data, we could copy the data from EBS volume to S3 bucket and then run the EMR jobs in oregon region with 11 machines. This limitation of machines resulted in additional time for data transfer between EBS, S3 and HDFS.

Conclusion:

1. We conclude that - “The busiest airport in US for the year range 2000-2008 is Hartsfield–Jackson Atlanta International Airport, Georgia”.
2. We obtained top 10 best airline for the year 2007 based on a comprehensive ranking system of each airline based on categories - largest number of air routes, longest distance covered, lowest delay time and lowest delay percentage for each airline.
3. We found the average delay for each airline carrier based on metrics like - carrier, month and hour.
4. Major Challenges -
 - a. Cluster Configuration - We ran our EMR cluster under region N.Virginia since the snapshot was available from this region but we could not use m3.xlarge machines. So we had to work with maximum of 5 instances of m1.large machines only.
 - b. Transport dataset’s snapshot data of linux drive was corrupt. So we took extra care for moving the windows version of the dataset to HDFS.
 - c. We had to decide on a load balancing technique to improve the performance of the PassengerCount analysis task when executing with 10 reduce tasks.
5. We decided to work with HBase because HBase is scalable. The basic unit of horizontal scalability in HBase is Region. Regions are a subset of the table’s data and they are essentially a contiguous, sorted range of rows that are stored together.
6. In future extension of our project we would like to predict flight departure delay based on weather specific data using machine learning models. The weather data set provided by **FAA Airline Operations Performance Data** (<https://aspm.faa.gov/>) contains hourly weather and operational data for 77 US airports which could be used for better predictability.
7. We would also like to attempt to use tweets (posts on Twitter) to predict delays. This is built on the premise that people who are experiencing flight delays are bored and unhappy, and bored and unhappy people tend to complain on Twitter. By analyzing the volume of tweets that mention a given airport, we believe that it may be possible to use these tweets as an indicator for delays that may otherwise not yet be published.

The dataset available (<https://drive.google.com/open?id=0B4nR50njtkBYNXM1QUZqQVd3cU0>) can be specifically used on predicting departure delays from LaGuardia Airport, in New York City.

Submitted By:

Survi Satpathy

Prachi Sharma

Prakash Somasundaram