**Name:** Khoa Cu Dang (Cody) Cao

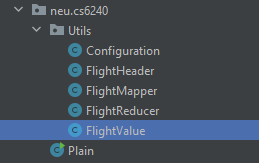
**Course:** CS6620 Spring 2024

# Homework 3 Report

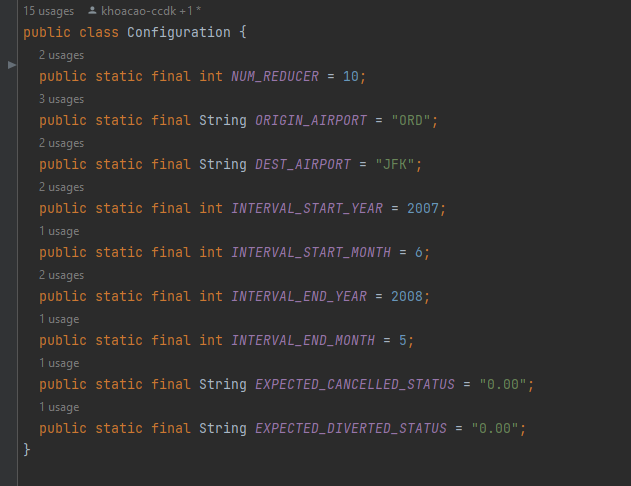
## Source Code

### Plain MapReduce

Like Homework 3, the Plain program for Homework 3 has its code separated into utility classes.



Configuration: This class stores all required constant values such as original airport, destination airports, date range, etc.



FlightHeader: this class stores the index of the parameters within a CSV line of flight record that is needed for the purpose of this program



FlightMapper: The map() function is called for each line of the data. It then performs the necessary filters.

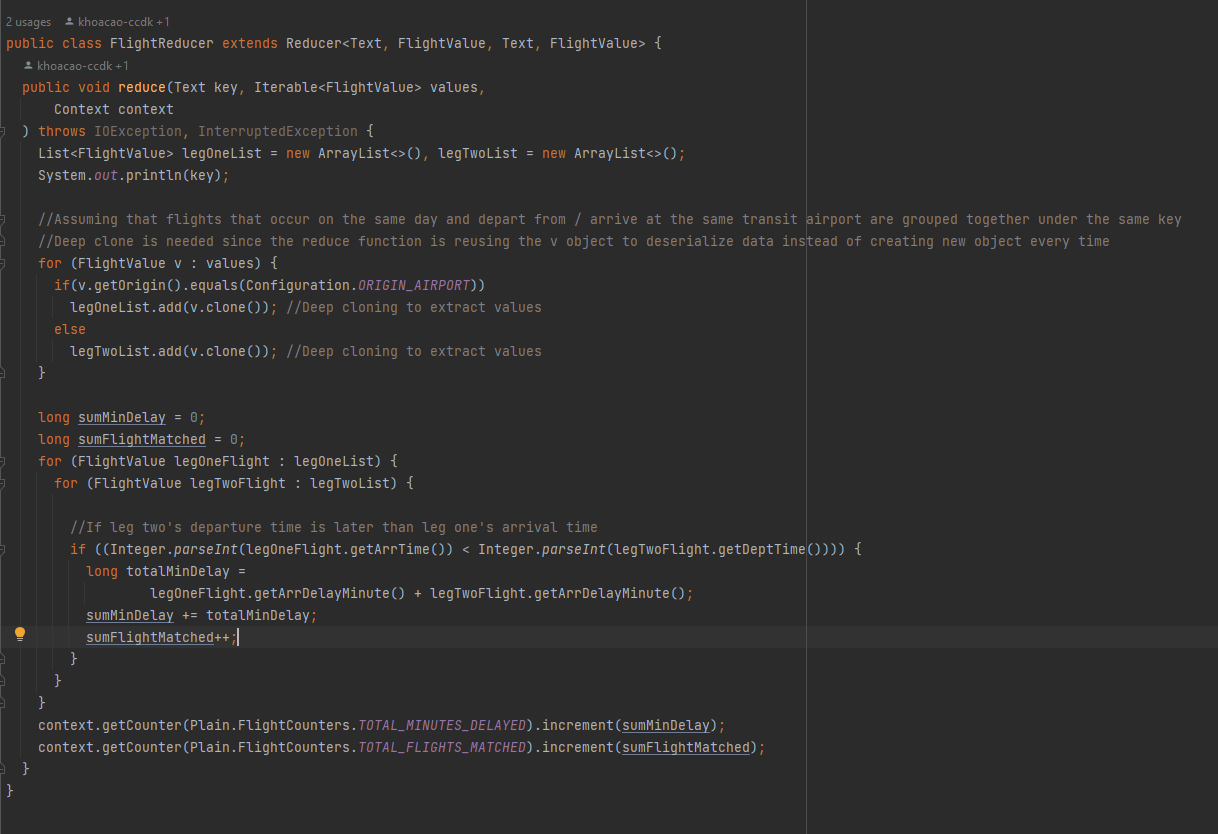
* Not cancelled/diverted.
* Within the specified date range.
* Originates from the ORD airport or arrives at JFK airport.

After filtering, the record is emitted as a key-value pair, with the key being a combination of flight date and the transit airport, while the value being a FlightValue object that contains necessary values.



FlightReducer: The logics of the reducer comes with an expectation that all values with the same key are matched together, which means each reduce call should have access to every flight on the same date that has the same transit airport.

The calculated results are then used to update the global counters so that there is a centralized counter value.



Main class: This class sets up the mapreduce program, as well as writing the result values to the output file after the program’s execution.

A flaw with this approach is that I could not figure out a way to write to a file in S3. Thus, I used System.out.println(), which would then write results to the stdout file.

A screenshot of a computer program

Description automatically generated

**Pseudo code**

Mapper:

1. Read each line from input.
2. Parse the line using a CSV parser to get individual fields.
3. Extract origin, destination, year, month, cancelled status, diverted status, and flight date from the parsed tokens.
4. Check if the flight is not cancelled and not diverted. If cancelled or diverted, skip the record.
5. Check if the flight falls within the defined time range. If not, skip the record.
6. Determine the transit airport based on origin and destination.
7. Extract departure time, arrival time, and arrival delay minutes.
8. Emit key-value pairs where the key is a combination of flight date and transit airport, and the value is a FlightValue object containing origin, destination, departure time, arrival time, and arrival delay minutes.

Reducer:

1. Receive key-value pairs grouped by the transit airport and flight date.
2. Separate flights into two lists: legOneList for flights departing from ORD and legTwoList for flights arriving at JFK.
3. Iterate over each flight in legOneList and legTwoList.
4. Check if the departure time of leg two's flight is later than the arrival time of leg one's flight.
5. If the condition is met, calculate the total delay minutes as the sum of arrival delay minutes of leg one and leg two's flights.
6. Increment global counters for total minutes delayed and total flights matched.

After the program execution, writes the global counter values to the stdout file of the Step on EMR.

### Pig FilterFirst

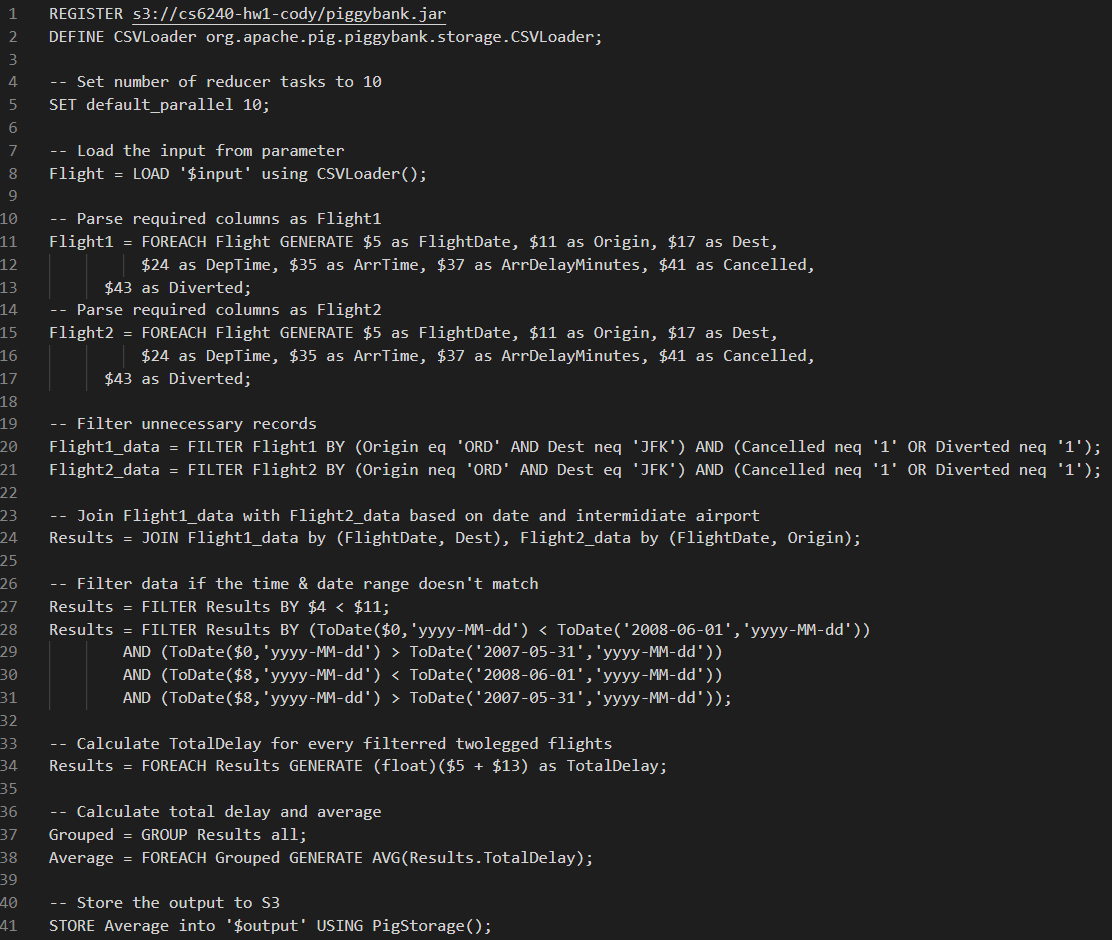
A screenshot of a computer program

Description automatically generated

**Pseudocode:**

1. Define a CSVLoader for loading CSV files.
2. Set the default number of reducer tasks to 10.
3. Load data from the input parameter into Flight using the CSVLoader.
4. Parse required columns from Flight into Flight1\_data and Flight2\_data.
5. Filter the data in Flight1\_data and Flight\_2 data (not cancelled/diverted, within date range, correct departure/arrival airport)
6. Join Flight1\_data with Flight2\_data based on date and transit airport
7. Filter Results to keep only relevant flights based on date.
8. Filter Results to keep only records that are valid (leg 2 deptTime > leg 1 arrTime)
9. Filter Results to keep records that are within the date range
10. Calculate the total delay for each filtered flight in Results.
11. Group Results to calculate the average total delay.
12. Store the average total delay to the output location specified.

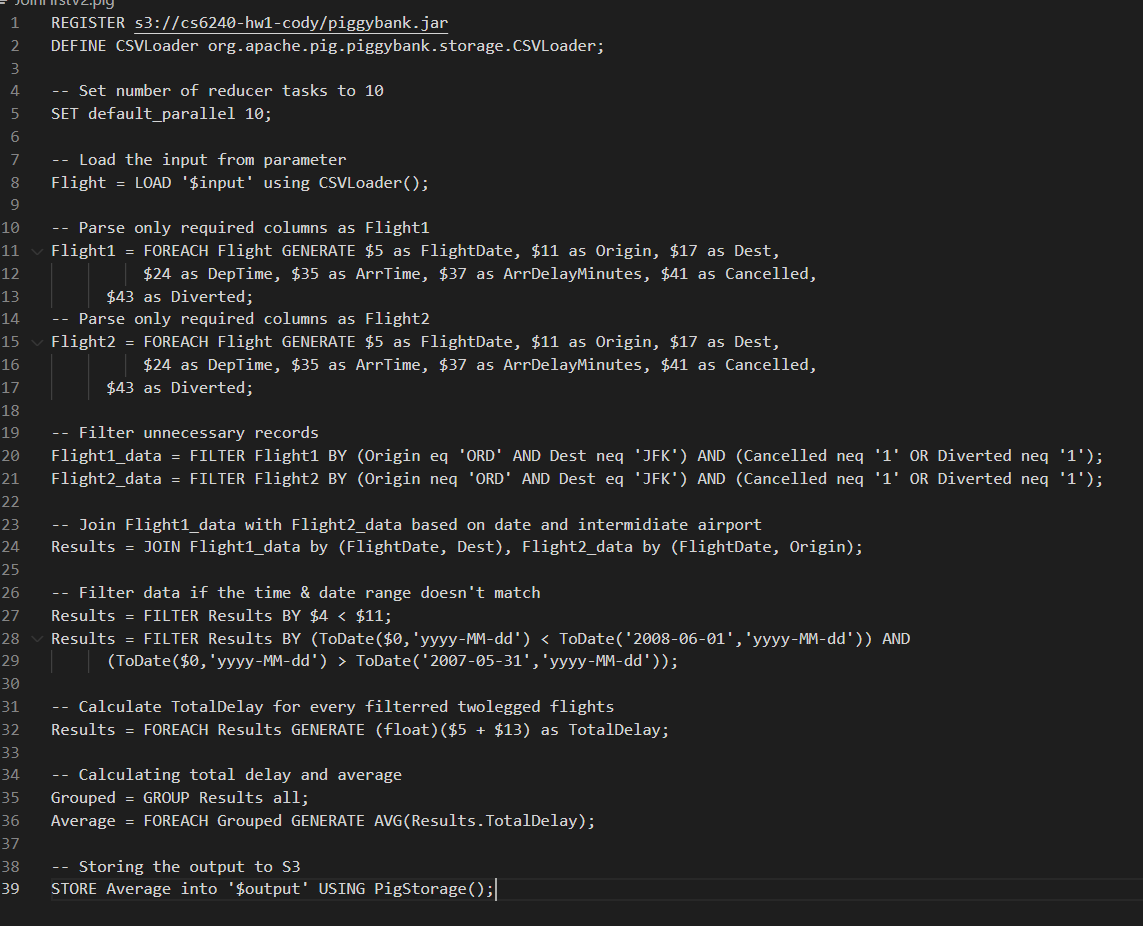
### Pig JoinFirst V1



**Pseudocode:**

1. Define a CSVLoader to handle CSV file loading.
2. Set the default number of reducer tasks to 10.
3. Load the input data from the input parameter using the defined CSVLoader into Flight.
4. Parse the required columns from `Flight` into `Flight1` and `Flight2`.
5. Filter out unnecessary records from `Flight1` and `Flight2` (only keep non-cancelled and non-diverted flights).
6. Join `Flight1` with `Flight2` based on the date and transit airport.
7. Filter Results to keep only records that are valid (leg 2 deptTime > leg 1 arrTime)
8. Further filter the joined data to ensure flight date of BOTH leg 1 and leg 2 are within the desired date range.
9. Calculate the total delay for each filtered flight leg.
10. Group the results to calculate the average total delay across all flight legs.
11. Store the average total delay to the output location specified.

### Pig JoinFirst V2



**Pseudocode:**

1. Define a CSVLoader to handle loading CSV files.
2. Set the default number of reducer tasks to 10.
3. Load the input data from input using the defined CSVLoader to `Flight`.
4. Extract only the required columns from `Flight` into `Flight1` and `Flight2`.
5. Filter out unnecessary records from `Flight1` and `Flight2` (only keep non-cancelled and non-diverted flights).
6. Join `Flight1` with `Flight2` based on the date and transit airport.
7. Further filter the joined data to ensure it falls within the desired time and date range.
8. Further filter the joined data to ensure flight date of leg 1 is within the desired date range.
9. Group the results to calculate the average total delay across all filtered flights.
10. Store the calculated average total delay to the specified output location.

### Output Results Plain

A screen shot of a computer

Description automatically generated

### Filter First



### JoinFirst V1

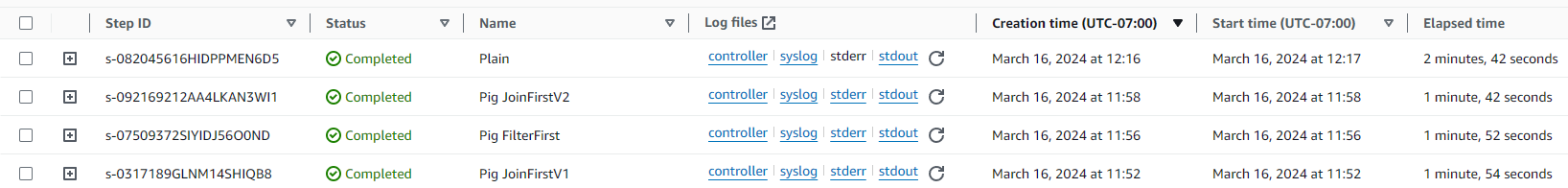


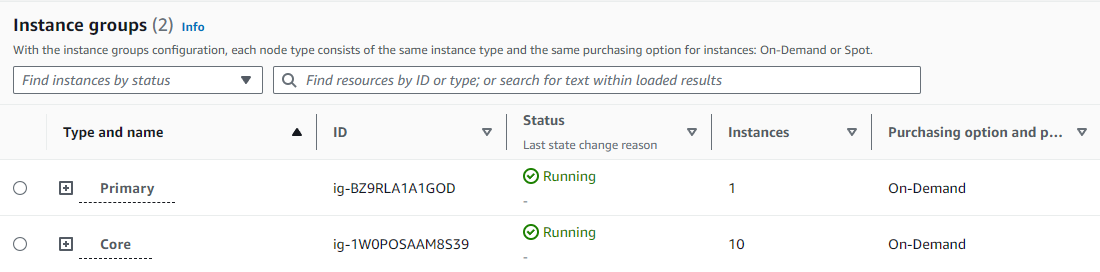
### JoinFirst V2



## Performance Comparison

### Proof of AWS EMR execution





### Runtime

A screenshot of a table

Description automatically generated

### Plain Java vs Pig

The actual runtime difference between Pig and Plain Java program meets my expectations due to the following reasons:

* The nature of Pig allows for more Abstraction and under the hood optimization, which helps tremendously when it comes to data processing.
* My Plain Java approach is not as optimized as Pig’s approach of handling data.
* Pig has PigOptimizer which can rewrite the scripts into more optimized execution plan using safe operations.

### Difference between Pig Scripts

#### Filter First vs Join First

The performance difference between these two approaches is dependent on the nature of the dataset. In this case, we see that the Join First approach being faster. This can potentially be explained by:

* Data produced after join is significantly smaller and easier to perform as a filter.
* The given date range and other conditions covers a large subset of data. Thus, the output of filtering still gives a large data set to be joined and further filtered.

#### Two Join First approaches

The difference between these two approaches seems to be more obvious, as the additional filter condition that requires checking both legs’ flight date requires more execution time, thus resulting in different run time results.