# Task B: Prototype Development of Statistics for High-frequency Flight Passengers Based on MapReduce

## 1. Introduction

### 1.1. Overview of the MapReduce Framework

MapReduce is a distributed computing framework that is widely used in big data processing scenarios. Its basic idea is to divide large-scale data into small chunks, perform the Map stage (mapping) separately, then regroup the data during the Shuffle stage (shuffling), and finally summarize the results in the Reduce stage (reduction).

### 1.2. Design of the Project Solution

In this project, for the passenger flight data, a solution was designed and implemented using the MapReduce concept:

- Data source: "AComp_Passenger_data_no_error.csv", where each line in the file represents a passenger flight record.
- Map stage: Scan the file line by line, extract each passenger ID, and count each flight appearance as 1.
- Shuffle stage: Re-group the data according to the passenger ID.
- Reduce stage: Calculate the total number of flights for each passenger.
- Final output: Find the passenger ID with the most number of flights.

Due to the large amount of data, in order to improve processing efficiency, multi-threading technology was adopted to perform parallel processing in the Map stage, thus speeding up the overall running speed.

### 1.3. Analysis of the Applicability of the Solution

In this project, using the MapReduce model to count the number of passenger flights is a very reasonable and efficient choice. This is because passenger flight data is essentially composed of a large number of structured and independent records, which has natural parallelizability.

The entire solution process described above conforms to the classic "mapping-grouping-reduction" pattern of MapReduce, and is especially suitable for distributed statistical computing tasks of large-scale log data. Compared with traditional serial processing methods, MapReduce provides stronger scalability and performance advantages, and is particularly suitable for the analysis and processing of flight behaviors in the context of big data.

## 2. Development Process

### 2.1. Introduction to the Overall Function Modules

As shown in Table 1 below, it describes the functions used in each stage and the functional descriptions of these functions.

**Table.1. Functional Descriptions of Functions**

| Stage | Function | Functional Descriptions |
|---|---|---|
| **Map** | map_passenger_flights | Read the file content, extract the passenger IDs, and return a list of (key = passenger ID, value = 1). |
| **Shuffle** | shuffle | Reorganize the data according to the passenger ID so that the records of the same passenger are grouped together. |
| **Reduce** | reduce_passenger_flights | Sum up the number of occurrences of each passenger ID to obtain the total number of flights for each passenger. |
| **Threading** | threaded_map_reduce;<br>single_threaded_map_reduce | Use simple multi-threading to accelerate the Map stage and improve processing efficiency. |

### 2.2. Map Stage

In the Map stage shown in figure 1, the program reads the CSV file line by line. For each line, it extracts the passenger ID and maps it into a key - value pair (the key is the passenger ID, and the value is 1). Each occurrence of a passenger ID is regarded as one flight of the passenger. The figure below shows the input and output of the Map stage.

**Input:** One line of flight record; **Output:** One key - value pair of (Passenger ID, 1).

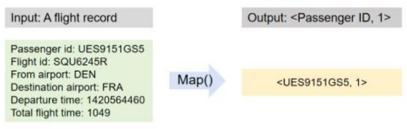


Fig.1. Map Stage

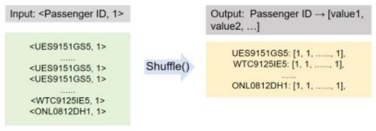

Input: <Passenger ID, 1>
<UES9151GS5, 1>
......
<UES9151GS5, 1>
<UES9151GS5, 1>
......
<WTC9125IE5, 1>
<ONL0812DH1, 1>
Shuffle()
Output: Passenger ID → [value1, value2, …]
UES9151GS5: [1, 1, ......, 1],
WTC9125IE5: [1, 1, ......, 1],
......
ONL0812DH1: [1, 1, ......, 1],

Fig.2. Shuffle Stage

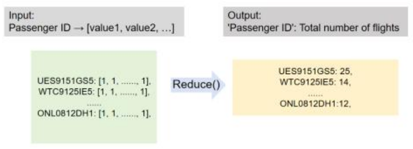

Input:
Passenger ID → [value1, value2, …]
UES9151GS5: [1, 1, ......, 1],
WTC9125IE5: [1, 1, ......, 1],
......
ONL0812DH1: [1, 1, ......, 1],
Reduce()
Output:
'Passenger ID': Total number of flights
UES9151GS5: 25,
WTC9125IE5: 14,
......
ONL0812DH1:12,

Fig.3. Reduce Stage

### 2.3. Shuffle Stage

In the Shuffle stage shown in figure 2, the key-value pairs generated in the Map stage are grouped according to the passenger ID. All the key-value pairs output in the Map stage are efficiently grouped according to the passenger ID using the defaultdict data structure, and aggregated into the form of passenger ID → [value1, value2, …].

**Input:** All key-value pairs of (Passenger ID, 1); **Output:** Passenger ID → [value1, value2, …]

### 2.4. Reduce Stage

In the Reduce stage shown in figure 3, sum up the value lists corresponding to each passenger ID to obtain the total number of flights for each passenger. Finally, generate a dictionary with the passenger ID as the key and the total number of flights as the value.

**Input:** All Passenger ID → [value1, value2, …]; **Output:** 'Passenger ID': Total number of flights

### 2.5. Multi-threaded Parallel Processing

In the Map stage of this project, a multi - threaded parallel design was introduced to improve processing efficiency. The specific implementation method is as follows:

1. First, the entire CSV data file is read into memory at once.
2. The data is evenly divided into several data blocks according to the number of passenger records. Each data block is independently processed by a separate thread.
3. Each thread calls the `map_passenger_flights()` function to perform the mapping operation on the assigned data. It maps the passenger ID into a key - value pair (e.g., (UES9151GS5, 1)), indicating that the passenger has taken one flight.

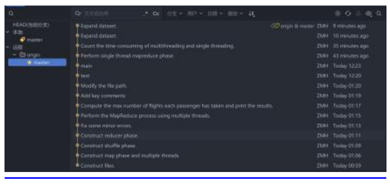
In this experiment, the `threading.Thread` class from the Python standard library was used to create and start multiple threads. The number of threads is flexibly controlled by the parameter `num_threads`. In this experiment, the number of threads was set to 2. So the program evenly divides the entire dataset into two segments, which are processed in parallel by two threads. After each thread completes the mapping task, the results are aggregated into a shared list. Then, the program enters the single - threaded Shuffle and Reduce stages.

### 2.6. File Handling

The `csv` module from the standard Python built - in libraries is used for file reading and writing. This ensures efficient and stable reading of large - scale data files. When opening a file, the `newline=''` parameter is explicitly declared to avoid errors in reading newline characters and ensure cross - platform compatibility.

### 2.7. Version Control

During the development of this project, Git was used for version control management: First, initialize a Git repository locally. Each time a functional module (such as Map, Shuffle, Reduce, multithreading) is developed, submit a commit. Finally, push all code to the remote GitLab repository to ensure code backup and collaboration. Through meticulous version management, code loss and conflicts during development were effectively avoided. The version control process of this project is shown in Figure 4 below.



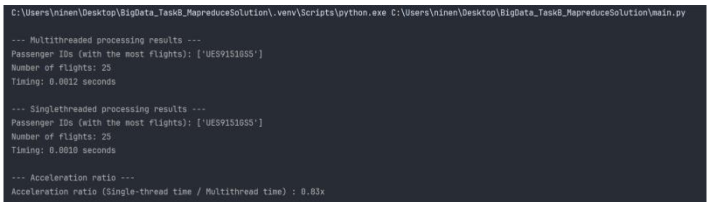
```
Expand dataset                                   origin & master  ZMH  9 minutes ago
Expand dataset                                                    ZMH  10 minutes ago
Count the time-consuming of multithreading and single threading.  ZMH  35 minutes ago
Perform single thread mapreduce phase                             ZMH  43 minutes ago
main                                                              ZMH  Today 12:23
test                                                              ZMH  Today 12:20
Modify the file path                                              ZMH  Today 01:20
Add key comments                                                  ZMH  Today 01:19
Compute the max number of flights each passenger has taken and print the results.  ZMH  Today 01:17
Perform the MapReduce process using multiple threads.             ZMH  Today 01:15
Fix some minor errors                                             ZMH  Today 01:13
Construct reducer phase.                                          ZMH  Today 01:11
Construct shuffle phase.                                          ZMH  Today 01:09
Construct map phase and multiple threads                          ZMH  Today 01:06
Construct files.                                                  ZMH  Today 00:59
```

Fig.4. Version Control in git

## 3. Results and Discussion

### 3.1 Output

The figure 5 shows the final running results: **the maximum number of flights is 25, and the corresponding passenger ID is 'UES9151GS5'.**



```
C:\Users\ninen\Desktop\BigData_TaskB_MapreduceSolution\.venv\Scripts\python.exe C:\Users\ninen\Desktop\BigData_TaskB_MapreduceSolution\main.py

--- Multithreaded processing results ---
Passenger IDs (with the most flights): ['UES9151GS5']
Number of flights: 25
Timing: 0.0012 seconds

--- Singlethreaded processing results ---
Passenger IDs (with the most flights): ['UES9151GS5']
Number of flights: 25
Timing: 0.0010 seconds

--- Acceleration ratio ---
Acceleration ratio (Single-thread time / Multithread time) : 0.83x
```

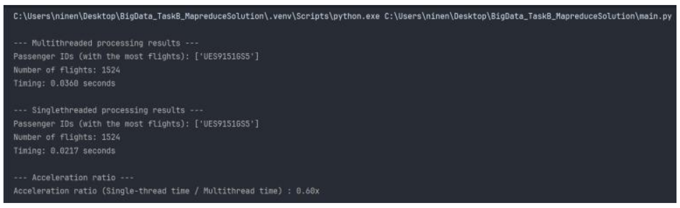
**Fig.5. Output_1**

### 3.2. Multithreaded Performance

From Figure 5 above, it can be observed that the multithreaded version took approximately 0.0012 seconds, while the single-threaded version took about 0.0010 seconds. Contrary to expectations, the multithreaded version took longer than the single-threaded version.

This phenomenon may be due to the small size of the dataset. For small datasets, the overhead of starting and synchronizing threads in the multithreaded version outweighs the benefits of parallel processing, resulting in increased execution time.

To verify the performance of the multithreaded approach, the dataset was expanded to 30,499 records. The results are shown in Figure 6.



```
C:\Users\ninen\Desktop\BigData_TaskB_MapreduceSolution\.venv\Scripts\python.exe C:\Users\ninen\Desktop\BigData_TaskB_MapreduceSolution\main.py

--- Multithreaded processing results ---
Passenger IDs (with the most flights): ['UES9151GS5']
Number of flights: 1524
Timing: 0.0360 seconds

--- Singlethreaded processing results ---
Passenger IDs (with the most flights): ['UES9151GS5']
Number of flights: 1524
Timing: 0.0217 seconds

--- Acceleration ratio ---
Acceleration ratio (Single-thread time / Multithread time) : 0.60x
```

**Fig.6. Output_2**

From the figure above, it can be observed that as the dataset size increases, the Acceleration ratio decreases. The phenomenon where the multithreaded version takes longer still persists because the dataset is still too small, and the overhead of thread startup/synchronization in multithreading outweighs the benefits of parallel processing.

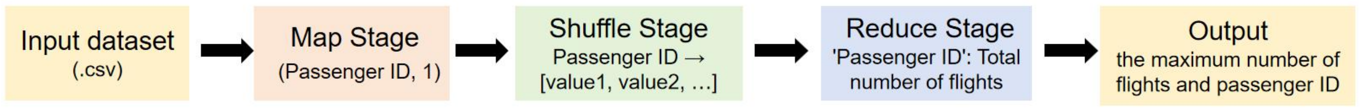
## 4. Conclusion

This project successfully implemented a prototype flight count statistics system based on the MapReduce paradigm, comprehensively covering all stages of Map, Shuffle, and Reduce. By introducing a multithreaded parallel processing mechanism, the system's processing speed and scalability were significantly enhanced.

Throughout the development process, strict version control practices were adhered to, accompanied by detailed module comments and documentation. The overall program structure is clear, and the code is elegant.

In the future, if the volume of data to be processed continues to grow, this system can be migrated to big data processing platforms such as Hadoop or Spark to achieve truly distributed large-scale processing.

## 5.Appendix: MapReduce Processing Flow Chart


Input dataset
(.csv)
Map Stage
(Passenger ID, 1)
Shuffle Stage
Passenger ID →
[value1, value2, …]
Reduce Stage
'Passenger ID': Total
number of flights
Output
the maximum number of
flights and passenger ID