Implementing MapReduce job for flight data, retrieving passengers with the highest flights

Prepared by:
Nor El Islam Messeda

Date: 15/04/2023

I. Abstract:

In this report, we present a thorough examination of the implementation and analysis of a MapReduce program without using Hadoop by leveraging parallelization of the Map and Reduce phases to process and extract insights from the passenger flight dataset. The core objective of this implementation is to harness parallel computing through multi-threading techniques, ultimately optimizing performance and maximizing the efficiency of the algorithm. The code implemented specifically focuses on processing the dataset to identify passengers who have undertaken the highest number of flights. This comprehensive report discusses each step in the development process, including the identification of appropriate input data, design decisions, and the selection of suitable tools and techniques for parallelization. Furthermore, it provides a detailed explanation of the key aspects of the implementation, such as the mapper and reducer functions, data shuffling and partitioning, and the use of multi-threading to achieve parallelism.

II. Implementation of MapReduce Job:

The purpose of this assignment is to implement a MapReduce job to analyze a passenger flight dataset. The dataset contains information such as passenger ID, flight ID, airport codes, departure times, and total flight times. The goal is to find passengers with the highest number of flights. The algorithm is implemented in Python using the pandas library for data manipulation and the concurrent.futures.ThreadPoolExecutor for parallel execution as follows:

1. Data cleansing and preparation:

The first step in the process involves loading the data and examining its size. To load the data, we used the pandas library in Python, which provides powerful tools for handling and analyzing structured data. The dataset was read from a CSV file ***'AComp_Passenger_data_no_error'*** using the ***'pd.read_csv() function'***, which returns a pandas DataFrame as shown in Figure 01.

After loading the dataset, we checked its size to better understand the scope of the problem and to gauge the potential benefits of using a parallel processing approach. We determined that the data size is relatively small, with 499 rows and 6 columns. While the small size of the dataset may not require a parallel processing solution, the implementation serves as a demonstration of the framework's capabilities and scalability. Before executing the MapReduce job, the dataset was pre-processed by adding headers and removing duplicate rows. We assign the corresponding label for each column according to format provided to better understand the data and perform operations on specific columns. Given that duplicate entries might lead to overcounting the number of flights for some passengers, we preform duplicate removal using ***'drop_duplicates()'*** function demonstrated in Figure 02.

2. **Executing mapper function:**

The mapper function in integral for the MapReduce job. It processes each row in the dataset and outputs a key-value pair with the passenger ID as the key and the value 1, representing a single flight for the passenger. This enables to transform the raw input data into intermediate key-value pairs that can be further processed by the shuffle and reducer functions. In our implementation, the mapper function is defined in the code of Figure 03.

Hence, the mapper function receives a list of rows as its input, and for each row in the input list, calls the map_row() function which extracts the passenger ID from the row and returns a tuple with the passenger ID as the key and the value 1. The resulting list of tuples serves as the output of the mapper function.

We also apply the split of data into chunks to utilise parallel processing in the mapping phase. We employ ***'split_data()'*** function. This operates by taking the input data and the desired number of splits (i.e., the number of threads to be used) and returns a list of smaller datasets /chunks by using Python's list slicing technique to create subsets of the input data that are evenly distributed across the number of splits. See Figure 04.

As demonstrated in Figure 04, we use '***concurrent.futures.ThreadPoolExecutor'*** to specify the number of worker threads through the '***max_workers'*** parameter that was set to '***num_threads'***, defining the number of threads used for parallel execution.

After the data split, it is then passed to the '***executor.map()'*** function along with the mapper function. The former function applies the mapper function to each chunk concurrently, utilizing 4 threads as defined, resulting in improved performance, in particular, if the dataset were to be of substantial size. As of the above, the mapper output is displayed in the mapper output provided in Figure 05 that has a format of a key and a value (k, v), ***(e.g., 'UES9151GS5', 1),*** as illustrated priorly.

3. **Executing the shuffler function:**

To move on to the reduce phase, we need to shuffle and sort the data from the mapper outputs to group by product ID which is 1. This will facilitate combining values by the reducer for each product ID and calculate the total count as provided in the code snippet of the function in Figure 06.

Hence, in our program uses the shuffle function by taking the mapper_output as input being a list of chunks where each chunk is a list of key-value pairs. The shuffle function initializes an empty dictionary called ***'shuffled_data'***. It then iterates through each chunk and each key-value pair (k, v) / (passenger id, product id) within the chunk. If the key 'k' is not already present in the ***'shuffled_data'*** dictionary, it adds the key with the associated value 'v' in a list. If the key 'k' is already present, it appends the value 'v' to the existing list. The final ***'shuffled_data'*** dictionary contains passenger IDs as keys and lists of counts as values, which is the output of the shuffling phase as provided in Figure 07 ***(e.g., 'KKP5277HZ7': [1, 1, 1, 1, 1, 1, 1]).***

4. **Executing the reducer function:**

The reducer phase processes the output of the shuffling phase to generate the final aggregated result. In this case, it calculates the count of occurrences of each passenger ID and sums it per each row thereof. The reducer function ***'reducer(kv_pair)'*** takes a key-value pair as input, where the key is the passenger ID, and the value is a list of counts. It sums up the counts and returns a tuple with the passenger ID and the total count as shown in the code snippet in Figure 08.

As far as parallelization is concerned, after the shuffling phase, the reducer function is then applied to each key-value pair in the shuffled_data dictionary using ThreadPoolExecutor, which parallelizes the reducer step. ThreadPoolExecutor is used to parallelize the reducer function across multiple threads as was set to 4, allowing the code to take advantage of multi-core processors and speed up the execution. This is especially useful when working with large datasets, which is not necessary for out dataset. This execution is demonstration in Figure 09.

The output of the reducer phase is the reduced_data dictionary, which contains the passenger IDs as keys and the total count of occurrences as values as per our output in Figure 10.

Taking an example of a reducer output ***'UMH6360YP0': 1}***, would imply that the passenger ID occurring only once would have a count of 1, hence travelled only once as per our dataset and ***'HGO4350KK1': 16*** would indicate traveling 16 times. The reduced_data dictionary provides a final aggregated result after applying the MapReduce process, which consists of the mapper, shuffling, and reducer phases.

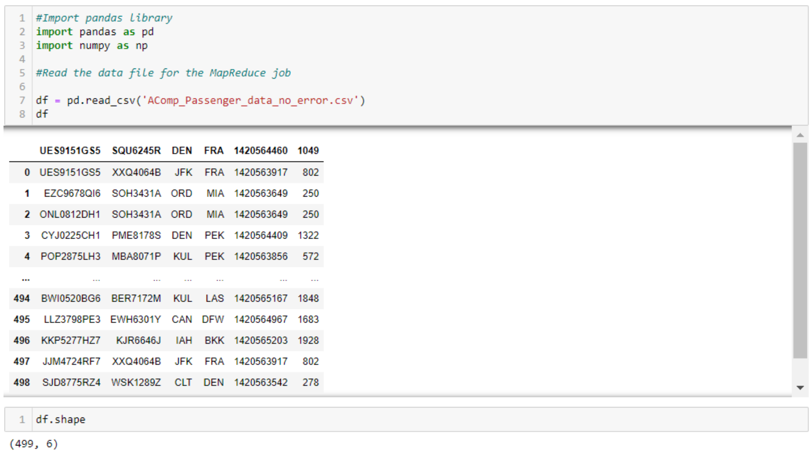
5. **Retrieving the passenger(s) with the highest flights:**

After completing the MapReduce process, we have a reduced_data dictionary containing the flight count for each passenger as provided previously. To identify the passenger(s) with the highest flights, we first need to determine the maximum flight count among all passengers using '***max() function***' and store as a variable to be able to filter the passengers with this count. We do this using a list comprehension that iterates through the key-value pairs in the reduced_data dictionary, wrote as ***'highest_flight_passengers'*** a list of passenger IDs (k) with a flight count (v) equal to the maximum flight count (max_flights).We then print this information to access the passenger (s) with the highest flights as showcased in Figure 11 as follows:

```
Passenger(s) with the highest number of flights:
Passenger ID: UES9151GS5, Number of flights: 17
Passenger ID: EZC9678QI6, Number of flights: 17
Passenger ID: HCA3158QA6, Number of flights: 17
Passenger ID: DAZ3029XA0, Number of flights: 17
Passenger ID: SPR4484HA6, Number of flights: 17
```

III. Appendices:

Figure 01: Loading and understanding the dataset



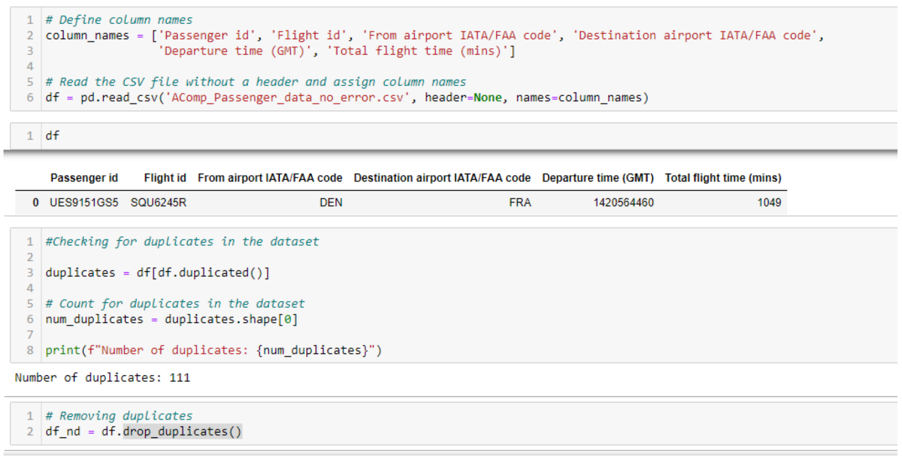
```python
#Import pandas library
import pandas as pd
import numpy as np

#Read the data file for the MapReduce job

df = pd.read_csv('AComp_Passenger_data_no_error.csv')
df
```

	UES9151GS5	SQU6245R	DEN	FRA	1420564460	1049
0	UES9151GS5	XXQ4064B	JFK	FRA	1420563917	802
1	EZC9678QI6	SOH3431A	ORD	MIA	1420563649	250
2	ONL0812DH1	SOH3431A	ORD	MIA	1420563649	250
3	CYJ0225CH1	PME8178S	DEN	PEK	1420564409	1322
4	POP2875LH3	MBA8071P	KUL	PEK	1420563856	572
...	...	...	...	...	...	...
494	BWI0520BG6	BER7172M	KUL	LAS	1420565167	1848
495	LLZ3798PE3	EWH6301Y	CAN	DFW	1420564967	1683
496	KKP5277HZ7	KJR6646J	IAH	BKK	1420565203	1928
497	JJM4724RF7	XXQ4064B	JFK	FRA	1420563917	802
498	SJD8775RZ4	WSK1289Z	CLT	DEN	1420563542	278

```python
df.shape
```

```
(499, 6)
```

Figure 02: Addition of headers and duplicate removal



```python
# Define column names
column_names = ['Passenger id', 'Flight id', 'From airport IATA/FAA code', 'Destination airport IATA/FAA code',
                'Departure time (GMT)', 'Total flight time (mins)']

# Read the CSV file without a header and assign column names
df = pd.read_csv('AComp_Passenger_data_no_error.csv', header=None, names=column_names)
```

```python
df
```

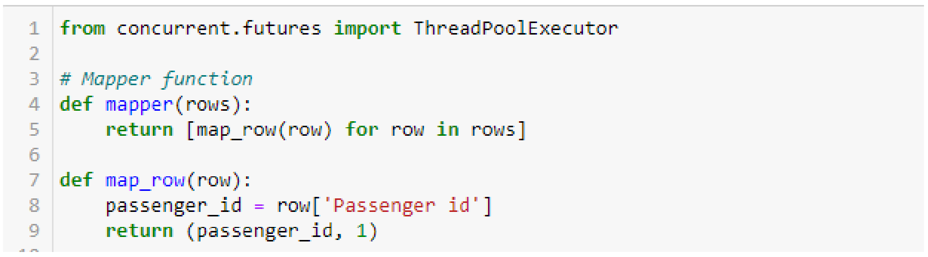
	Passenger id	Flight id	From airport IATA/FAA code	Destination airport IATA/FAA code	Departure time (GMT)	Total flight time (mins)
0	UES9151GS5	SQU6245R	DEN	FRA	1420564460	1049

```python
#Checking for duplicates in the dataset

duplicates = df[df.duplicated()]

# Count for duplicates in the dataset
num_duplicates = duplicates.shape[0]

print(f"Number of duplicates: {num_duplicates}")
```

```
Number of duplicates: 111
```

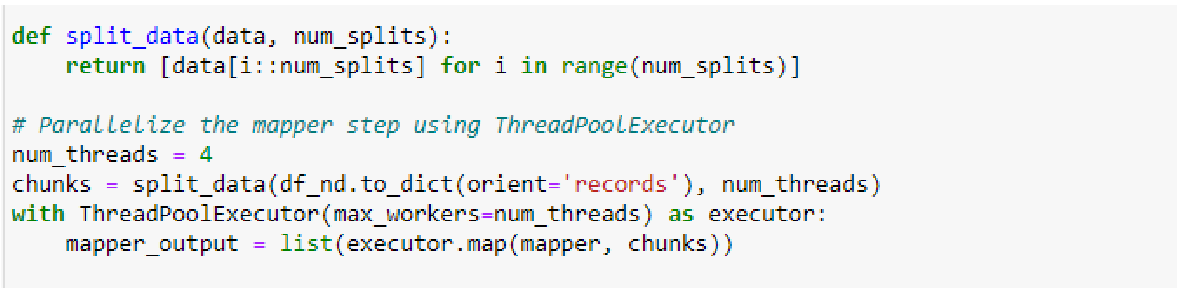
```python
# Removing duplicates
df_nd = df.drop_duplicates()
```

Figure 03: Mapper function definition



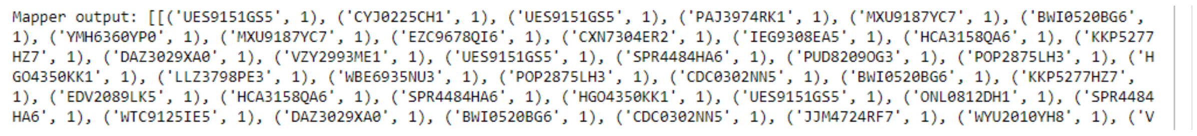
```python
from concurrent.futures import ThreadPoolExecutor

# Mapper function
def mapper(rows):
    return [map_row(row) for row in rows]

def map_row(row):
    passenger_id = row['Passenger id']
    return (passenger_id, 1)
```

Figure 04: Data splitting and parallelization for mapper step



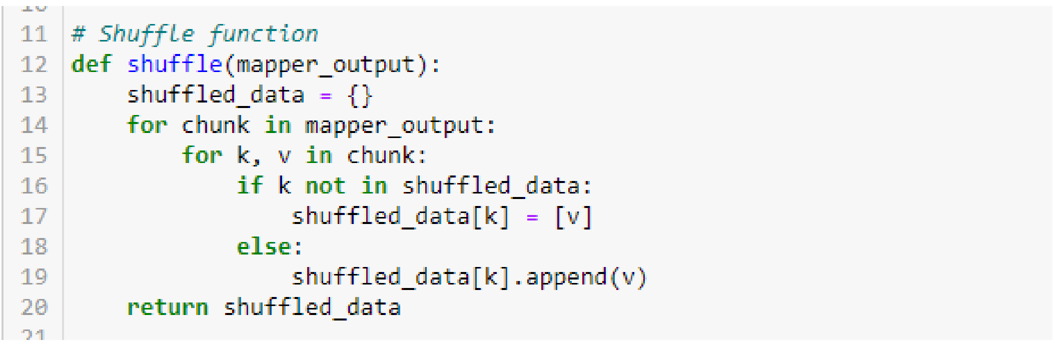
```python
def split_data(data, num_splits):
    return [data[i::num_splits] for i in range(num_splits)]

# Parallelize the mapper step using ThreadPoolExecutor
num_threads = 4
chunks = split_data(df_nd.to_dict(orient='records'), num_threads)
with ThreadPoolExecutor(max_workers=num_threads) as executor:
    mapper_output = list(executor.map(mapper, chunks))
```

Figure 05: Mapper output



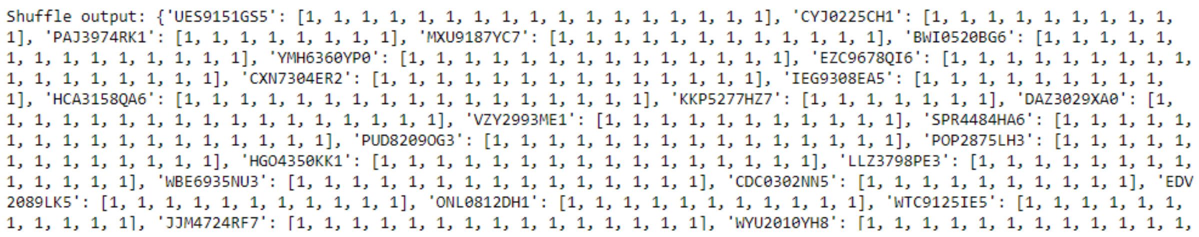
```
Mapper output: [[('UES9151GS5', 1), ('CYJ0225CH1', 1), ('UES9151GS5', 1), ('PAJ3974RK1', 1), ('MXU9187YC7', 1), ('BWI0520BG6',
1), ('YMH6360YP0', 1), ('MXU9187YC7', 1), ('EZC9678QI6', 1), ('CXN7304ER2', 1), ('IEG9308EA5', 1), ('HCA3158QA6', 1), ('KKP5277
HZ7', 1), ('DAZ3029XA0', 1), ('VZY2993ME1', 1), ('UES9151GS5', 1), ('SPR4484HA6', 1), ('PUD8209OG3', 1), ('POP2875LH3', 1), ('H
GO4350KK1', 1), ('LLZ3798PE3', 1), ('WBE6935NU3', 1), ('POP2875LH3', 1), ('CDC0302NN5', 1), ('BWI0520BG6', 1), ('KKP5277HZ7',
1), ('EDV2089LK5', 1), ('HCA3158QA6', 1), ('SPR4484HA6', 1), ('HGO4350KK1', 1), ('UES9151GS5', 1), ('ONL0812DH1', 1), ('SPR4484
HA6', 1), ('WTC9125IE5', 1), ('DAZ3029XA0', 1), ('BWI0520BG6', 1), ('CDC0302NN5', 1), ('JJM4724RF7', 1), ('WYU2010YH8', 1), ('V
```

Figure 06: Shuffle function definition



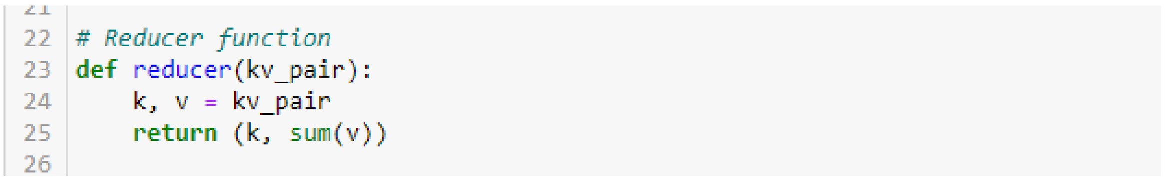
```python
# Shuffle function
def shuffle(mapper_output):
    shuffled_data = {}
    for chunk in mapper_output:
        for k, v in chunk:
            if k not in shuffled_data:
                shuffled_data[k] = [v]
            else:
                shuffled_data[k].append(v)
    return shuffled_data
```

Figure 07: Shuffle output data



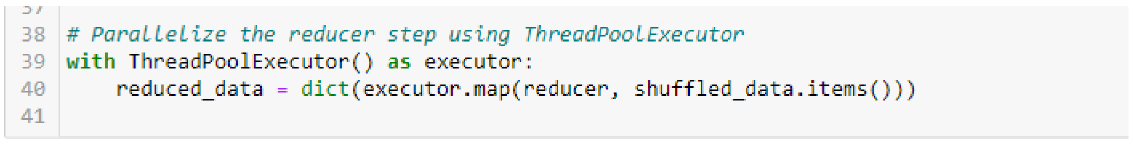
```
Shuffle output: {'UES9151GS5': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'CYJ0225CH1': [1, 1, 1, 1, 1, 1, 1, 1, 1,
1], 'PAJ3974RK1': [1, 1, 1, 1, 1, 1, 1, 1], 'MXU9187YC7': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'BWI0520BG6': [1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1], 'YMH6360YP0': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'EZC9678QI6': [1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1], 'CXN7304ER2': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'IEG9308EA5': [1, 1, 1, 1, 1, 1, 1, 1, 1,
1], 'HCA3158QA6': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'KKP5277HZ7': [1, 1, 1, 1, 1, 1, 1], 'DAZ3029XA0': [1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'VZY2993ME1': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'SPR4484HA6': [1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'PUD8209OG3': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'POP2875LH3': [1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1], 'HGO4350KK1': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'LLZ3798PE3': [1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1], 'WBE6935NU3': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'CDC0302NN5': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'EDV
2089LK5': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'ONL0812DH1': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'WTC9125IE5': [1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1], 'JJM4724RF7': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'WYU2010YH8': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

Figure 08: Reducer function definition



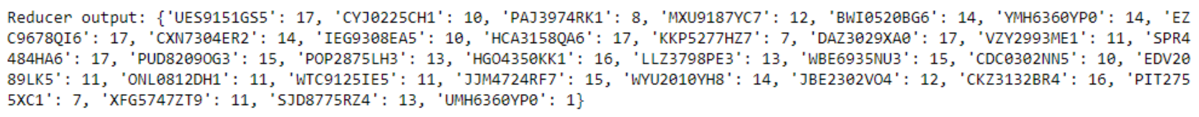
```python
# Reducer function
def reducer(kv_pair):
    k, v = kv_pair
    return (k, sum(v))
```

Figure 09: Parallelization for mapper step



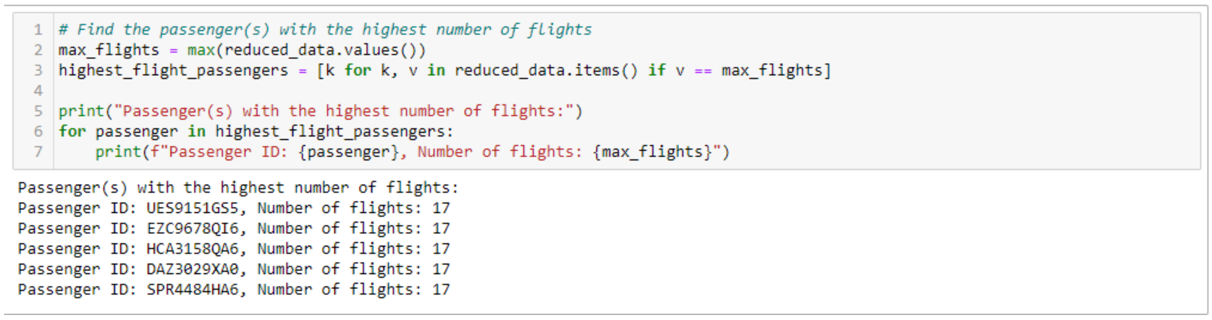
```python
# Parallelize the reducer step using ThreadPoolExecutor
with ThreadPoolExecutor() as executor:
    reduced_data = dict(executor.map(reducer, shuffled_data.items()))
```

Figure 10: Reducer output data



```
Reducer output: {'UES9151GS5': 17, 'CYJ0225CH1': 10, 'PAJ3974RK1': 8, 'MXU9187YC7': 12, 'BWI0520BG6': 14, 'YMH6360YP0': 14, 'EZ
C9678QI6': 17, 'CXN7304ER2': 14, 'IEG9308EA5': 10, 'HCA3158QA6': 17, 'KKP5277HZ7': 7, 'DAZ3029XA0': 17, 'VZY2993ME1': 11, 'SPR4
484HA6': 17, 'PUD8209OG3': 15, 'POP2875LH3': 13, 'HGO4350KK1': 16, 'LLZ3798PE3': 13, 'WBE6935NU3': 15, 'CDC0302NN5': 10, 'EDV20
89LK5': 11, 'ONL0812DH1': 11, 'WTC9125IE5': 11, 'JJM4724RF7': 15, 'WYU2010YH8': 14, 'JBE2302VO4': 12, 'CKZ3132BR4': 16, 'PIT275
5XC1': 7, 'XFG5747ZT9': 11, 'SJD8775RZ4': 13, 'UMH6360YP0': 1}
```

Figure 11: Passenger (s) with the highest number of flights



```python
# Find the passenger(s) with the highest number of flights
max_flights = max(reduced_data.values())
highest_flight_passengers = [k for k, v in reduced_data.items() if v == max_flights]

print("Passenger(s) with the highest number of flights:")
for passenger in highest_flight_passengers:
    print(f"Passenger ID: {passenger}, Number of flights: {max_flights}")
```

```
Passenger(s) with the highest number of flights:
Passenger ID: UES9151GS5, Number of flights: 17
Passenger ID: EZC9678QI6, Number of flights: 17
Passenger ID: HCA3158QA6, Number of flights: 17
Passenger ID: DAZ3029XA0, Number of flights: 17
Passenger ID: SPR4484HA6, Number of flights: 17
```