Assignment_1

September 5, 2021

Name: Nabilah Anuwar

Student ID: 31282016

1 Table of contents

[1]:
```
from pyspark.sql import Row
from pyspark.sql.functions import UserDefinedFunction
from pyspark.sql.functions import col, when
import pyspark.sql.functions as F
from pyspark.sql.types import *
```

2 1 Working with RDD

2.1 1.1 Data Preparation and Loading

2.1.1 1.1.1 Create SparkSession and SparkContext

Section ??

[2]:
```
from pyspark import SparkConf
master = "local[*]"
app_name = "31282016"
spark_conf = SparkConf().setMaster(master).setAppName(app_name)

from pyspark import SparkContext
from pyspark.sql import SparkSession

spark = SparkSession.builder.config(conf = spark_conf).getOrCreate()
sc = spark.sparkContext
sc.setLogLevel("ERROR")
```

2.1.2 1.1.2 Import CSV files and Make RDD for each file

Section ??

[3]:
```
flights_rdd0 = sc.textFile("flight-delays/flight*.csv")
airports_rdd0 = sc.textFile("flight-delays/airports.csv")
```

1.1.2.1 Flights RDD Section ??

Function to convert certain columns to their correct format

[4]:
```
def convert_them(rdd):
    new_rdd = []
    for i in rdd:
        new_row = []
        for f in list(range(len(i))):
            if i[f] in ('', None, " ", "") and (f in␣
 ↪[0,1,2,3,5,11,12,15,16,17,19,22]):
                new_row.append(None)
            elif f in [0,1,2,3,5]:
                new_row.append(int(i[f]))
            elif f in [11,12,15,16,17,19,22]:
```

```
                new_row.append(float(i[f]))
            else:
                new_row.append(i[f])
        new_rdd.append(new_row)
    return new_rdd
```

Split the words by commas and making it into a list for each line

[5]:
```
flights_rdd1 = flights_rdd0.map(lambda line: line.split(','))
header_flights = flights_rdd1.first()
flights_rdd1 = flights_rdd1.filter(lambda row: row != header_flights)
```

Create a schema and creating a DataFrame with that schema and our previous rdd. The columns YEAR, MONTH, DAY, DAY_OF_WEEK, and FLIGHT_NUMBER is Integer type based one the question. Columns such as DEPARTURE_DELAY, TAXI_OUT, ELAPSED_TIME, AIR_TIME, DISTANCE, TAXI_IN, and ARRIVAL_DELAY are Float type based on the request of the question.

[6]:
```
flights_list = convert_them(flights_rdd1.collect())
```

[7]:
```
schema_flights = StructType([
    StructField("YEAR", IntegerType()),
    StructField("MONTH", IntegerType()),
    StructField("DAY", IntegerType()),
    StructField("DAY_OF_WEEK", IntegerType()),
    StructField("AIRLINE", StringType()),
    StructField("FLIGHT_NUMBER", IntegerType()),
    StructField("TAIL_NUMBER", StringType()),
    StructField("ORIGIN_AIRPORT", StringType()),
    StructField("DESTINATION_AIRPORT", StringType()),
    StructField("SCHEDULED_DEPARTURE", StringType()),
    StructField("DEPARTURE_TIME", StringType()),
    StructField("DEPARTURE_DELAY", FloatType()),
    StructField("TAXI_OUT", FloatType()),
    StructField("WHEELS_OFF", StringType()),
    StructField("SCHEDULED_TIME", StringType()),
    StructField("ELAPSED_TIME", FloatType()),
    StructField("AIR_TIME", FloatType()),
    StructField("DISTANCE", FloatType()),
    StructField("WHEELS_ON", StringType()),
    StructField("TAXI_IN", FloatType()),
    StructField("SCHEDULED_ARRIVAL", StringType()),
    StructField("ARRIVAL_TIME", StringType()),
    StructField("ARRIVAL_DELAY", FloatType()),
    StructField("DIVERTED", StringType()),
    StructField("CANCELLED", StringType()),
    StructField("CANCELLATION_REASON", StringType()),
    StructField("AIR_SYSTEM_DELAY", StringType()),
```

```
    StructField("SECURITY_DELAY", StringType()),
    StructField("AIRLINE_DELAY", StringType()),
    StructField("LATE_AIRCRAFT_DELAY", StringType()),
    StructField("WEATHER_DELAY", StringType())
])
```

[8]:
```
flights_df = spark.createDataFrame(flights_list, schema_flights)
flights_rdd = flights_df.rdd
```

[9]:
```
flights_rdd.take(1)
# 05081995
```

[9]:
```
[Row(YEAR=2015, MONTH=6, DAY=26, DAY_OF_WEEK=5, AIRLINE='EV',
FLIGHT_NUMBER=4951, TAIL_NUMBER='N707EV', ORIGIN_AIRPORT='BHM',
DESTINATION_AIRPORT='LGA', SCHEDULED_DEPARTURE='630', DEPARTURE_TIME='629',
DEPARTURE_DELAY=-1.0, TAXI_OUT=13.0, WHEELS_OFF='642', SCHEDULED_TIME='155',
ELAPSED_TIME=141.0, AIR_TIME=113.0, DISTANCE=866.0, WHEELS_ON='935',
TAXI_IN=15.0, SCHEDULED_ARRIVAL='1005', ARRIVAL_TIME='950', ARRIVAL_DELAY=-15.0,
DIVERTED='0', CANCELLED='0', CANCELLATION_REASON='', AIR_SYSTEM_DELAY='',
SECURITY_DELAY='', AIRLINE_DELAY='', LATE_AIRCRAFT_DELAY='', WEATHER_DELAY='')]
```

1.1.2.2 Airports RDD Section ??

Split the words by commas and making it into a list for each line

[10]:
```
airports_rdd1 = airports_rdd0.map(lambda line: line.split(','))
header_airport = airports_rdd1.first()
airports_rdd1 = airports_rdd1.filter(lambda row: row != header_airport)
```

Create a schema and make a DataFrame from the `airports_rdd1` then convert it to rdd. There is no specification on the data type thus we make all a String type.

[11]:
```
schema_airport = StructType([
    StructField("IATA_CODE", StringType()),
    StructField("AIRPORT", StringType()),
    StructField("CITY", StringType()),
    StructField("STATE", StringType()),
    StructField("COUNTRY", StringType()),
    StructField("ALATITUDA", StringType()),
    StructField("ALONGITUDE", StringType())
])
```

[12]:
```
airports_df = spark.createDataFrame(airports_rdd1, schema_airport)
airports_rdd = airports_df.rdd
```

[13]:
```
airports_rdd.take(2)
# 05081995
```

[13]:
```
[Row(IATA_CODE='ABE', AIRPORT='Lehigh Valley International Airport',
CITY='Allentown', STATE='PA', COUNTRY='USA', ALATITUDA='40.65236',
ALONGITUDE='-75.44040'),
 Row(IATA_CODE='ABI', AIRPORT='Abilene Regional Airport', CITY='Abilene',
STATE='TX', COUNTRY='USA', ALATITUDA='32.41132', ALONGITUDE='-99.68190')]
```

2.1.3 1.1.3 Show RDD number of columns, and number of records

Section **??**

Number of Columns

[14]:
```
print("Number of Columns")
print("flights_rdd:", flights_rdd.map(lambda x: len(x)).take(1)[0])
print("airports_rdd:", airports_rdd.map(lambda x: len(x)).take(1)[0])
```

```
Number of Columns
flights_rdd: 31
airports_rdd: 7
```

Number of Records

[15]:
```
print("Number of Records")
trec_flights = flights_rdd.cache().count()
trec_airports = airports_rdd.cache().count()
print("flights_rdd:", trec_flights)
print("airports_rdd:", trec_airports)
```

```
Number of Records
flights_rdd: 582184
airports_rdd: 322
```

Number of Partitions

[16]:
```
print("Number of Partitions")
print("flights_rdd:", flights_rdd.getNumPartitions())
print("airports_rdd:", airports_rdd.getNumPartitions())
```

```
Number of Partitions
flights_rdd: 2
airports_rdd: 2
```

2.2 1.2 Dataset Partitioning

1. First we need to check which rows have missing values within `ARRIVAL_DELAY`

[17]:
```
print(flights_rdd.filter(lambda x: x[22] == None).cache().count())
```

```
10455
```

2. ARRIVAL_DELAY is a result from subtracting ARRIVAL_TIME and SCHEDULED_ARRIVAL. We want to see which of these is the cause of ARRIVAL_DELAY being missing

[18]:
```
mAD_AT = flights_rdd.filter(lambda x: x[22] == None and x[21] == "").cache().
 ↪count()
mAD_SA = flights_rdd.filter(lambda x: x[22] == None and x[20] == "").cache().
 ↪count()
print("Missing ARRIVAL_DELAY and ARRIVAL_TIME:", mAD_AT)
print("Missing ARRIVAL_DELAY and SCHEDULED_ARRIVAL:", mAD_SA)
```

```
Missing ARRIVAL_DELAY and ARRIVAL_TIME: 9257
Missing ARRIVAL_DELAY and SCHEDULED_ARRIVAL: 0
```

- From this, we find that there is a difference in number between point 1 and 2 which means that some values of ARRIVAL_DELAY is not calculated properly.
- Another finding is that the SCHEDULED_TIME seems to not be the cause of the missing value.

Before continuing we will need to calculate the missing value from the first finding

[19]:
```
import math

# function to make sure string length is 4 and add 0s
def add_zeros(a):
    if len(str(a)) < 4:
        zeros = "0"*(4-len(str(a)))
        result = zeros+str(a)
    else:
        result = str(a)
    return result

# get minutes from difference of time
def get_mins(a,b):
    a = add_zeros(a)
    b = add_zeros(b)
    ah = int(a[:-2])
    am = int(a[-2:])

    bh = int(b[:-2])
    bm = int(b[-2:])

    m1 = (ah*60)+am
    m2 = (bh*60)+bm

    diff = float(m1-m2)

    return diff
```

[20]:
```
flights_rdda = flights_rdd.filter(lambda x: x[22] == None and x[21] != "" and␣
↪x[20] != "") # not yet calculated
flights_rddb = flights_rdd.filter(lambda x: x[22] == None and x[21] == "") #␣
↪missing ARRIVAL_TIME
flights_rddc = flights_rdd.filter(lambda x: x[22] != None) # not missing
```

Here we assume that all the values are appropriate for `flights_rdda`. The values on `ARRIVAL_TIME` and `SCHEDULED_TIME` is HHmm. However, if the hours are not present it will not be written so 0s at the front will not be written. Example,

[21]:
```
flights_rdda = flights_rdda.map(lambda x: Row(*x[0:22], get_mins(x[21],x[20]),␣
↪*x[23:]))
flights_rdda = spark.createDataFrame(flights_rdda, schema_flights).rdd
```

3. `ARRIVAL_TIME` is calculated from `WHEELS_ON` and `TAXI_IN`. We can try to find how many from `flights_rddb` that have these two values

[22]:
```
print(flights_rddb.filter(lambda x: x[18] != "" and x[19] != None).cache().
↪count())
```

```
0
```

We can see from the result above that there are no data from `flights_rddb` that has valid values of `WHEELS_ON` and `TAXI_IN`. We will keep the `ARRIVAL_DELAY` column as None and remove these row from the maximum and minimum calculation. We will not use the value 0 as it means there is no delay in the flight.

[23]:
```
print(f"Percentage of data that we lost removing flights_rddb: {round((mAD_AT/
↪trec_flights)*100,2)}%")
```

```
Percentage of data that we lost removing flights_rddb: 1.59%
```

Thus we can now combine those with non empty values in `ARRIVAL_DELAY` column

[24]:
```
frdd_clean = flights_rdda.union(flights_rddc)
```

2.2.1 1.2.1 Obtain the maximum arrival delay

Section **??**

[25]:
```
q121 = frdd_clean.max(lambda x: x[22])[22]
print(f"Maximum arrival delay is {q121} minutes")
```

```
Maximum arrival delay is 1665.0 minutes
```

2.2.2 1.2.2 Obtain the minimum arrival delay

Section **??**

[26]:
```
q122 = frdd_clean.min(lambda x: x[22])[22]
print(f"Minimum arrival delay is {q122} minutes")
```

```
Minimum arrival delay is -1371.0 minutes
```

2.2.3 1.2.3 Define hash partitioning function

Section **??**

Hash Partitioning is not like Range Partitioning. Its main goal is to try to distribute the rows evenly among the partitions. Thus, it is unlikely that max and min can fully determine the distribution within the partition.

- make `ARRIVAL_DELAY` the key
- get max and min of `ARRIVAL_DELAY` to determine the distribution within the partition by having a range function

1. Get ideal number of partition, the ideal amount of partition is total cores times 4

[27]:
```
total_cores = sc.defaultParallelism
part_no = total_cores*4
```

[28]:
```
print(f"The number of partitions that will be used is: {part_no}")
```

```
The number of partitions that will be used is: 8
```

2. Define Hash Function

[29]:
```
def hash_function(key):
    print(key)
    the_range = []
    gaps = (q121-q122)/part_no
    low = q122
    for i in list(range(part_no)):
        the_range.append(low+(gaps*i))

    for r in list(range(part_no)):
        if float(key) <= the_range[r]:
            return int(the_range[r]-float(key))
    return 0
```

2. Create Hash Partitioning Function

[30]:
```
def partByhash(the_rdd):
    # make ARRIVAL_DELAY the key
    data = the_rdd.map(lambda x: (x[22],x))

    result = data.partitionBy(part_no, hash_function)

    return result
```

[31]:
```
frdd_hash = partByhash(frdd_clean)
```

2.2.4 1.2.4 Display the records in each partition

Section **??**

[32]:
```
def print_partitions(data):
    numPartitions = data.getNumPartitions()
    partitions = data.glom().collect()

    print(f"####### NUMBER OF PARTITIONS: {numPartitions}")
    for index, partition in enumerate(partitions):
        # show partition if it is not empty
        if len(partition) > 0:
            print(f"Partition {index}: {len(partition)} records")
```

[33]:
```
print_partitions(frdd_hash)
```

```
####### NUMBER OF PARTITIONS: 8
Partition 0: 71783 records
Partition 1: 71288 records
Partition 2: 71902 records
Partition 3: 71740 records
Partition 4: 71494 records
Partition 5: 71499 records
Partition 6: 71520 records
Partition 7: 71701 records
```

Here the hash function try to distribute the data based on the maximum and minimum values of the `ARRIVAL_DELAY` column. I try to make the function to group them based on the range list that was created then get the integer difference between the number within the range and the key itself.

Effect of number of partitions in processing speed Number of partition need to be just enough for us to use partition ideally. Too less means we are not using all the available cores, which means we might put too much data in the few partitions where we can give them lighter load. If we have too much partitions, the cores might not be able to parallel process them as there are more partitions than the available cores. Thus, we determine the number of partitions based on the core. Each core can handle around 3 to 4 partitions. Thus the ideal number of partition is the number of cores x 3 or 4, in this case we times it by 4.

Effect of hash functions in processing speed As mentioned before, hash partitioning main goal is to evenly distribute the data in all partitions. If we use range partitioning it might be useful for time bound data, however I believe this data will not be used mainly from its timed data. Hash partitioning helps by putting even load to all partitions, thus all would have almost the same amount of time to process.

Normal Partitioning vs Hash Partitioning Though they both aim to partition entries evenly, hash partitioning is better as it mixes the entries. Thus one partition will have different types of entries while normal partition will have similar data types in the partition.

2.3 1.3 Query RDD

2.3.1 1.3.1 Collect a total number of flights for each month

Section **??**

We use the full uncleaned data as it is more suitable to represent the total number of flights each month

[34]:
```
q131 = flights_rdd.map(lambda x: (x[1],1)).groupByKey().mapValues(len).
↪sortByKey()
q131.collect()
```

[34]:
```
[(1, 47136),
 (2, 42798),
 (3, 50816),
 (4, 48810),
 (5, 49691),
 (6, 50256),
 (7, 52065),
 (8, 50524),
 (9, 46733),
 (10, 48680),
 (11, 46809),
 (12, 47866)]
```

2.3.2 1.3.2 Collect the average delay for each month

Section **??**

We use cleaned data as the unavailable rows does now have valid data type for calculations

[35]:
```
from statistics import mean
q132 = frdd_clean.map(lambda x: (x[1],x[22])).groupByKey().mapValues(mean).
↪map(lambda x: (x[0],round(x[1],2))).sortByKey()
q132.collect()
```

[35]:
```
[(1, 5.86),
 (2, 8.07),
 (3, 5.09),
 (4, 3.29),
 (5, 4.69),
 (6, 9.95),
 (7, 6.8),
 (8, 4.79),
 (9, -0.84),
 (10, -0.63),
 (11, 0.91),
 (12, 6.07)]
```

3 2 Working with DataFrame

3.1 2.1. Data Preparation and Loading

3.1.1 2.1.1 Define dataframes and loading scheme

Section ??

[36]:
```
flightsDf = spark.read.load("flight-delays/flight*.csv", format = "csv", sep =␣
↪",", inferSchema = True, header = True)
airportsDf = spark.read.load("flight-delays/airports.csv", format = "csv", sep␣
↪= ",", inferSchema = True, header = True)
```

3.1.2 2.1.2 Display the schema of the final two dataframes

Section ??

[37]:
```
flightsDf.printSchema()
```

```
root
 |-- YEAR: integer (nullable = true)
 |-- MONTH: integer (nullable = true)
 |-- DAY: integer (nullable = true)
 |-- DAY_OF_WEEK: integer (nullable = true)
 |-- AIRLINE: string (nullable = true)
 |-- FLIGHT_NUMBER: integer (nullable = true)
 |-- TAIL_NUMBER: string (nullable = true)
 |-- ORIGIN_AIRPORT: string (nullable = true)
 |-- DESTINATION_AIRPORT: string (nullable = true)
 |-- SCHEDULED_DEPARTURE: integer (nullable = true)
 |-- DEPARTURE_TIME: integer (nullable = true)
 |-- DEPARTURE_DELAY: integer (nullable = true)
 |-- TAXI_OUT: integer (nullable = true)
 |-- WHEELS_OFF: integer (nullable = true)
 |-- SCHEDULED_TIME: integer (nullable = true)
 |-- ELAPSED_TIME: integer (nullable = true)
 |-- AIR_TIME: integer (nullable = true)
 |-- DISTANCE: integer (nullable = true)
 |-- WHEELS_ON: integer (nullable = true)
 |-- TAXI_IN: integer (nullable = true)
 |-- SCHEDULED_ARRIVAL: integer (nullable = true)
 |-- ARRIVAL_TIME: integer (nullable = true)
 |-- ARRIVAL_DELAY: integer (nullable = true)
 |-- DIVERTED: integer (nullable = true)
 |-- CANCELLED: integer (nullable = true)
 |-- CANCELLATION_REASON: string (nullable = true)
 |-- AIR_SYSTEM_DELAY: integer (nullable = true)
 |-- SECURITY_DELAY: integer (nullable = true)
 |-- AIRLINE_DELAY: integer (nullable = true)
```

```
 |-- LATE_AIRCRAFT_DELAY: integer (nullable = true)
 |-- WEATHER_DELAY: integer (nullable = true)
```

[38]:
```
airportsDf.printSchema()
```

```
root
 |-- IATA_CODE: string (nullable = true)
 |-- AIRPORT: string (nullable = true)
 |-- CITY: string (nullable = true)
 |-- STATE: string (nullable = true)
 |-- COUNTRY: string (nullable = true)
 |-- LATITUDE: double (nullable = true)
 |-- LONGITUDE: double (nullable = true)
```

Learning from Section 1, we found that `ARRIVAL_DELAY` had some missing values, yet some was possible to be calculated. So we will go through that first.

[39]:
```
x = flightsDf.filter(col("ARRIVAL_DELAY").isNull()).count()
print(f"The number of nulls without cleaning is {x} entries")
```

```
The number of nulls without cleaning is 10455 entries
```

[40]:
```
# get minutes from difference of time but integer result
def get_mins2(a,b):
    #different in this part as DF convert the ones with 0s to None rather than␣
 ↪as a string
    if a == None:
        a = "0"
    if b == None:
        b = "0"
    a = add_zeros(a)
    b = add_zeros(b)
    ah = int(a[:-2])
    am = int(a[-2:])

    bh = int(b[:-2])
    bm = int(b[-2:])

    m1 = (ah*60)+am
    m2 = (bh*60)+bm

    diff = float(m1-m2)
    # the result should be in integer not float
    return int(diff)

# create UDF
```

```
a_udf = UserDefinedFunction(get_mins2, IntegerType())
```

[41]:
```
flightsDf = flightsDf.withColumn("ARRIVAL_DELAY", when((col("ARRIVAL_DELAY").
↪isNull()) & (col("ARRIVAL_TIME").isNull() == False) & (col("SCHEDULED_TIME").
↪isNull() == False), a_udf("ARRIVAL_TIME","SCHEDULED_TIME")).
↪otherwise(col("ARRIVAL_DELAY")))
```

[42]:
```
y = flightsDf.filter(col("ARRIVAL_DELAY").isNull()).count()
print(f"The number of nulls with cleaning is {y} entries")
```

```
The number of nulls with cleaning is 9257 entries
```

The numbers match to our previous finding in RDD section, so we can continue

3.2 2.2. Query Analysis

3.2.1 2.2.1 January flight events with ANC airport

Section **??**

[43]:
```
janFlightEventsAncDf = flightsDf.filter(col("MONTH")==1).filter(col("YEAR")==␣
↪2015).filter(col("ORIGIN_AIRPORT")=="ANC").
↪select("MONTH","ORIGIN_AIRPORT","DESTINATION_AIRPORT","DISTANCE","ARRIVAL_DELAY")
janFlightEventsAncDf.show()
```

```
+-----+--------------+-------------------+--------+-------------+
|MONTH|ORIGIN_AIRPORT|DESTINATION_AIRPORT|DISTANCE|ARRIVAL_DELAY|
+-----+--------------+-------------------+--------+-------------+
|    1|           ANC|                SEA|    1448|          -13|
|    1|           ANC|                SEA|    1448|           -4|
|    1|           ANC|                JNU|     571|           17|
|    1|           ANC|                CDV|     160|           20|
|    1|           ANC|                BET|     399|          -20|
|    1|           ANC|                SEA|    1448|          -15|
|    1|           ANC|                SEA|    1448|          -11|
|    1|           ANC|                ADQ|     253|          -16|
|    1|           ANC|                SEA|    1448|           17|
|    1|           ANC|                BET|     399|           -9|
|    1|           ANC|                SEA|    1448|           15|
|    1|           ANC|                FAI|     261|           -6|
|    1|           ANC|                JNU|     571|            2|
|    1|           ANC|                JNU|     571|           -3|
|    1|           ANC|                PDX|    1542|          -21|
|    1|           ANC|                SEA|    1448|           -5|
|    1|           ANC|                SEA|    1448|          -15|
|    1|           ANC|                PDX|    1542|          -13|
|    1|           ANC|                SFO|    2018|           20|
|    1|           ANC|                FAI|     261|           56|
+-----+--------------+-------------------+--------+-------------+
```

```
only showing top 20 rows
```

3.2.2 2.2.2 Average Arrival Delay From Origin to Destination

Section ??

Results in `AVERAGE_DELAY` will be rounded to 2 decimal places.

[44]:
```python
janFlightsEventsAncAvgDf = janFlightEventsAncDf.groupBy(col("ORIGIN_AIRPORT"),␣
↪col("DESTINATION_AIRPORT")).agg(F.round(F.mean("ARRIVAL_DELAY"),2).
↪alias("AVERAGE_DELAY")).orderBy(col("AVERAGE_DELAY"))
janFlightsEventsAncAvgDf.show()
```

```
+--------------+-------------------+-------------+
|ORIGIN_AIRPORT|DESTINATION_AIRPORT|AVERAGE_DELAY|
+--------------+-------------------+-------------+
|           ANC|                ADK|        -27.0|
|           ANC|                HNL|        -20.0|
|           ANC|                MSP|       -19.25|
|           ANC|                BET|        -9.09|
|           ANC|                SEA|        -6.49|
|           ANC|                BRW|        -4.33|
|           ANC|                OME|         -3.0|
|           ANC|                ADQ|        -2.67|
|           ANC|                CDV|          1.0|
|           ANC|                OTZ|         1.25|
|           ANC|                PHX|          2.0|
|           ANC|                DEN|         3.33|
|           ANC|                PDX|          3.5|
|           ANC|                JNU|          5.0|
|           ANC|                LAS|          9.0|
|           ANC|                SCC|        16.67|
|           ANC|                SFO|         20.0|
|           ANC|                FAI|         25.0|
+--------------+-------------------+-------------+
```

3.2.3 2.2.3 Join Query with Airports DataFrame

Section ??

Here we have `janFlightsEventsAncAvgDf` `ORIGIN_AIRPORT` equal to `airportsDf` `IATA_CODE`

[45]:
```python
joinedSqlDf = janFlightsEventsAncAvgDf.join(airportsDf, airportsDf.IATA_CODE ==␣
↪janFlightsEventsAncAvgDf.ORIGIN_AIRPORT, how = "inner")
joinedSqlDf.show()
```

```
+--------------+-------------------+-------------+---------+--------------------
+---------+-----+-------+--------+----------+
```

```
|ORIGIN_AIRPORT|DESTINATION_AIRPORT|AVERAGE_DELAY|IATA_CODE|
AIRPORT|     CITY|STATE|COUNTRY|LATITUDE| LONGITUDE|
+--------------+-------------------+-------------+---------+--------------------
+---------+-----+-------+--------+----------+
|           ANC|                BRW|        -4.33|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                ADK|        -27.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                OME|         -3.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                JNU|          5.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                LAS|          9.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                SCC|        16.67|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                CDV|          1.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                DEN|         3.33|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                OTZ|         1.25|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                SFO|         20.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                FAI|         25.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                ADQ|        -2.67|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                PDX|          3.5|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                PHX|          2.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                HNL|        -20.0|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                SEA|        -6.49|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                MSP|       -19.25|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
|           ANC|                BET|        -9.09|      ANC|Ted Stevens
Ancho…|Anchorage|   AK|    USA|61.17432|-149.99619|
+--------------+-------------------+-------------+---------+--------------------
+---------+-----+-------+--------+----------+
```

3.3 2.3. Analysis

3.3.1 2.3.1 Relationship between day of week with mean arrival delay, total time delay, and count flights

Section ??

[46]:
```
flightsDf.createOrReplaceTempView("sql_flights")
q231 = spark.sql('''
    SELECT DAY_OF_WEEK, AVG(ARRIVAL_DELAY) AS MeanArrivalDelay,␣
 ↪SUM(ARRIVAL_DELAY) AS TotalTimeDelay, COUNT(*) AS NumOfFlights
    FROM sql_flights f
    WHERE YEAR = 2015
    GROUP BY DAY_OF_WEEK
    ORDER BY NumOfFlights DESC
    ''')
q231.show()
```

```
+-----------+-----------------+--------------+------------+
|DAY_OF_WEEK| MeanArrivalDelay|TotalTimeDelay|NumOfFlights|
+-----------+-----------------+--------------+------------+
|          4|7.312340849113385|        631757|       87683|
|          1|7.586891745907381|        639097|       86317|
|          5|6.328193109913288|        540048|       86253|
|          3|5.640166175478464|        476532|       85607|
|          2|6.234578920705377|        516884|       84449|
|          7|5.993143328171055|        479859|       81422|
|          6|3.754716438099898|        260919|       70453|
+-----------+-----------------+--------------+------------+
```

From here we find that the fourth day of the week or Thursday is when the highest `NumOfFlights` happen. Interestingly, though Thursday have the most flights, Monday or 1st day of the week is the one that have the highest `TotalTimeDelay` and with it having the highest `MeanArrivalDelay`

3.3.2 2.3.2 Display mean arrival delay each month

Section ??

No requirement showing if we need to find about those especially in the year 2015.

[47]:
```
q232 = spark.sql('''
    SELECT MONTH, AVG(ARRIVAL_DELAY) AS MeanArrivalDelay, SUM(ARRIVAL_DELAY) AS␣
 ↪TotalTimeDelay, COUNT(*) AS NumOfFlights
    FROM sql_flights f
    GROUP BY MONTH
    ORDER BY MeanArrivalDelay ASC
    ''')
q232.show()
```

```
+-----+------------------+-------------+------------+
|MONTH|  MeanArrivalDelay|TotalTimeDelay|NumOfFlights|
+-----+------------------+-------------+------------+
|    9| 0.393758328676439|        18320|       46733|
|   10|0.5218585441404233|        25271|       48680|
|   11|1.9537055047656098|        90396|       46809|
|    4|  5.18668404609687|       250688|       48810|
|    3| 6.196238345516422|       307699|       50816|
|    1| 6.754906653901388|       310442|       47136|
|    5| 7.012317025998087|       344438|       49691|
|    8| 7.126482292479053|       356374|       50524|
|   12| 7.848228338083287|       369008|       47866|
|    7| 9.019146459747818|       464937|       52065|
|    2| 9.404563857195436|       383283|       42798|
|    6|12.660014602092966|       624240|       50256|
+-----+------------------+-------------+------------+
```

The 9th month or September have the lowest `MeanArrivalDelay` compares to other months.

3.3.3 2.3.3 Relationship between mean departure delay and mean arrival delay

Section **??**

[48]:
```
q233 = spark.sql('''
    SELECT MONTH, AVG(DEPARTURE_DELAY) AS MeanDeptDelay, AVG(ARRIVAL_DELAY) AS␣
↪MeanArrivalDelay
    FROM sql_flights f
    GROUP BY MONTH
    ORDER BY MeanDeptDelay DESC
    ''')
q233.show()
```

```
+-----+------------------+------------------+
|MONTH|     MeanDeptDelay|  MeanArrivalDelay|
+-----+------------------+------------------+
|    6|  13.9730063585922|12.660014602092966|
|   12|11.821651454043728| 7.848228338083287|
|    7|11.708608758020432| 9.019146459747818|
|    2|11.620796080832823| 9.404563857195436|
|    8|10.086906141367324| 7.126482292479053|
|    1|  9.75401499511029| 6.754906653901388|
|    3| 9.718308159530178| 6.196238345516422|
|    5| 9.550310180006102| 7.012317025998087|
|    4| 7.737554783759199|  5.18668404609687|
|   11| 6.630585898709037|1.9537055047656098|
|   10| 5.243436261558784|0.5218585441404233|
|    9| 4.728506981740065| 0.393758328676439|
+-----+------------------+------------------+
```

As the `MeanDeptDelay` decrease the `MeanArrivalDelay` would decrease as well. However, this theory is true if we exclude the 12th and 5th month.

4 3 RDDs vs DataFrame vs Spark SQL

Implement the following queries using RDDs, DataFrames and SparkSQL separately. Log the time taken for each query in each approach using the "%%time" built-in magic command in Jupyter Notebook and discuss the performance difference of these 3 approaches.

Find the MONTH and DAY_OF_WEEK, number of flights, and average delay where TAIL_NUMBER = 'N407AS'. Note number of flights and average delay should be aggregated separately. The average delay should be grouped by both MONTH and DAYS_OF_WEEK.

4.1 3.1 RDD Operation

Section **??**

[49]:
```
q31 = frdd_clean.filter(lambda x: x["TAIL_NUMBER"] == "N407AS")\
.map(lambda x: ((x["MONTH"], x["DAY_OF_WEEK"]),(1,␣
↪x["DEPARTURE_DELAY"],x["ARRIVAL_DELAY"])))\
.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1], x[2]+y[2]))\
.mapValues(lambda x: (x[0],round(x[1]/x[0],2),round(x[2]/x[0],2)))\
.sortByKey()
%timeit q31
```

```
40.3 ns ± 0.947 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[50]:
```
q31.take(20)
```

[50]:
```
[((1, 1), (1, 4.0, -6.0)),
 ((1, 2), (2, 12.5, 17.5)),
 ((1, 3), (1, -7.0, -27.0)),
 ((1, 5), (2, -6.0, -21.0)),
 ((1, 6), (3, 8.67, 4.33)),
 ((2, 1), (2, -4.0, -2.5)),
 ((2, 2), (2, -3.5, -9.5)),
 ((2, 3), (2, -12.5, -11.5)),
 ((2, 4), (2, -8.5, -11.0)),
 ((2, 5), (1, -11.0, -31.0)),
 ((2, 7), (2, -7.0, 6.5)),
 ((3, 1), (1, 40.0, 29.0)),
 ((3, 2), (2, -5.5, -28.0)),
 ((3, 3), (1, 28.0, 3.0)),
 ((3, 4), (1, 1.0, 2.0)),
 ((3, 5), (3, 5.67, 6.67)),
 ((3, 6), (1, -1.0, -3.0)),
 ((4, 1), (1, -1.0, 0.0)),
```

```
 ((4, 2), (1, -2.0, 6.0)),
 ((4, 3), (1, -4.0, -7.0))]
```

4.2 3.2 DataFrame Operation

Section ??

[51]:
```
q32 = flightsDf.filter(col("TAIL_NUMBER")=="N407AS")\
.groupBy(col("MONTH"),col("DAY_OF_WEEK"))\
.agg(F.count("MONTH").alias("NumOfFlights"), F.round(F.
 ↪mean("DEPARTURE_DELAY"),2).alias("MeanDeptDelay"), F.round(F.
 ↪mean("ARRIVAL_DELAY"),2).alias("MeanArrivalDelay"))\
.orderBy("MONTH","DAY_OF_WEEK")
%timeit q32
```

```
20.1 ns ± 0.337 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

[52]:
```
q32.show()
```

```
+-----+-----------+------------+-------------+----------------+
|MONTH|DAY_OF_WEEK|NumOfFlights|MeanDeptDelay|MeanArrivalDelay|
+-----+-----------+------------+-------------+----------------+
|    1|          1|           1|          4.0|            -6.0|
|    1|          2|           2|         12.5|            17.5|
|    1|          3|           1|         -7.0|           -27.0|
|    1|          5|           2|         -6.0|           -21.0|
|    1|          6|           3|         8.67|            4.33|
|    2|          1|           2|         -4.0|            -2.5|
|    2|          2|           2|         -3.5|            -9.5|
|    2|          3|           2|        -12.5|           -11.5|
|    2|          4|           2|         -8.5|           -11.0|
|    2|          5|           1|        -11.0|           -31.0|
|    2|          7|           2|         -7.0|             6.5|
|    3|          1|           1|         40.0|            29.0|
|    3|          2|           2|         -5.5|           -28.0|
|    3|          3|           1|         28.0|             3.0|
|    3|          4|           1|          1.0|             2.0|
|    3|          5|           3|         5.67|            6.67|
|    3|          6|           1|         -1.0|            -3.0|
|    4|          1|           1|         -1.0|             0.0|
|    4|          2|           1|         -2.0|             6.0|
|    4|          3|           1|         -4.0|            -7.0|
+-----+-----------+------------+-------------+----------------+
only showing top 20 rows
```

4.3 3.3 Spark SQL OPERATION

Section ??

[53]:
```
q33 = spark.sql('''
    SELECT MONTH, DAY_OF_WEEK, COUNT(*) AS NumOfFlights, AVG(DEPARTURE_DELAY)␣
 ↪AS MeanDeptDelay, AVG(ARRIVAL_DELAY) AS MeanArrivalDelay
    FROM sql_flights f
    WHERE TAIL_NUMBER == "N407AS"
    GROUP BY MONTH, DAY_OF_WEEK
    ORDER BY MONTH, DAY_OF_WEEK
    ''')
%timeit q33
```

```
18.3 ns ± 0.448 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
```

[54]:
```
q33.show(20)
```

```
+-----+-----------+------------+-----------------+-----------------+
|MONTH|DAY_OF_WEEK|NumOfFlights|    MeanDeptDelay| MeanArrivalDelay|
+-----+-----------+------------+-----------------+-----------------+
|    1|          1|           1|              4.0|             -6.0|
|    1|          2|           2|             12.5|             17.5|
|    1|          3|           1|             -7.0|            -27.0|
|    1|          5|           2|             -6.0|            -21.0|
|    1|          6|           3|8.666666666666666|4.333333333333333|
|    2|          1|           2|             -4.0|             -2.5|
|    2|          2|           2|             -3.5|             -9.5|
|    2|          3|           2|            -12.5|            -11.5|
|    2|          4|           2|             -8.5|            -11.0|
|    2|          5|           1|            -11.0|            -31.0|
|    2|          7|           2|             -7.0|              6.5|
|    3|          1|           1|             40.0|             29.0|
|    3|          2|           2|             -5.5|            -28.0|
|    3|          3|           1|             28.0|              3.0|
|    3|          4|           1|              1.0|              2.0|
|    3|          5|           3|5.666666666666667|6.666666666666667|
|    3|          6|           1|             -1.0|             -3.0|
|    4|          1|           1|             -1.0|              0.0|
|    4|          2|           1|             -2.0|              6.0|
|    4|          3|           1|             -4.0|             -7.0|
+-----+-----------+------------+-----------------+-----------------+
only showing top 20 rows
```

4.4 3.4 Discussion

Section ??

From the queries above we can see that RDD query have the fastest result. This is because RDD is considered raw. It has not been processed fully like DataFrame, which makes it easier for computers to process, though not the same for humans.

SQL and DataFrame has a similar time result. This may be because SQL was derived from the DataFrame. But it is a query directly towards the data. For DataFrame, they are already processed and its more visualised. When we query with DataFrame we are trying to query from the table not the data itself. Which explains the time difference.

RDD may also be faster since we only preprocessed necessary columns while DataFrame and SQL set everything up.

[ ]:

```

```