Flight Delay Trends

DAT500 project in Spring of 2023

Bhakti Prabhu
University of Stavanger, Norway

Stephan Frederik Werner Brandasu
University of Stavanger, Norway

ABSTRACT

In this paper, we aimed to analyse flight delay and cancellation trends using big data technologies such as Hadoop and Spark. By leveraging data from the U.S. Department of Transportation Statistics, we aimed to identify patterns in the data to answer questions such as which months or weeks have the most delayed or cancelled flights and which carriers are more likely to experience delays.

The paper focuses mainly on how we utilized Hadoop and Spark technologies, for building the application and the Spark optimisation techniques used to improve the performance of the application. We discuss the different performance problems we faced while executing a Spark application and the measures we took to mitigate some of those issues. Specifically, we discuss the problems of data skew, spill, serialisation and shuffle. For each of these issues, we suggest techniques for optimizing Spark applications, such as data partitioning, optimizing code and choosing appropriate aggregation methods while transforming the data.

Overall, our paper demonstrates the practical use of big data technologies to extract insights from large datasets and highlights the challenges and solutions associated with processing large datasets and optimizing the execution process. The findings of our use case could be useful for travellers, airlines, airport authorities, and other stakeholders in the aviation industry to better understand the causes of flight delays and cancellations and to take appropriate measures to improve their performance.

KEYWORDS

flight delay, spark, pyspark, hadoop, spark optimisations

1 INTRODUCTION

1.1 Use Case

Did you ever experience being stranded in an airport because of flight delays or cancellations, and pondered on the possibility of being able to predict it with the help of more data? The use case that we chose for our project poses a solution for this problem. The U.S. Department of Transportation Statistics (DTS) tracks the on-time performance of domestic flights operated by large air carriers[4]. Summary information on the number of on-time, delayed, cancelled, and diverted flights is published in the DTS's monthly Air Travel Consumer Report. We have taken our dataset from their official website. The dataset consists of the flight delay and cancellation information for the years from 2018 to 2022.

The goal of our use case is to analyse the trends in delays, to get an understanding of how the flight on-time performance varies in a particular time period. This analysis gives answer to various questions like, during which months in a year flights get most delayed, which day or week in a month has more possibility of flights getting delayed or cancelled, which flight carrier gets delayed more often compared to others and many more similar questions. This information can be useful for both a traveller and the airline carrier. Having access to this information a traveller can ensure that they do not miss any important meetings or events, while at the same time an airline carrier can know when it needs to increase their work force and use this opportunity to improve over their competitors.

Project in Computer Science (DAT500), IDE, UiS
2023.

1.2 Background

The goal of this project is to learn and get understanding of Hadoop and Spark Technologies which are frameworks for handling data intensive systems or nowadays commonly known as Big Data. Apache Hadoop is a collection of open-source modules and utilities intended to make the process of storing, managing and analysing big data easier while Apache Spark is an open-source data processing engine built for efficient, large-scale data analysis. Apache Spark can be run either standalone or as a software package on top of Apache Hadoop.

For our project we used Hadoop for converting our unstructured data which was in Text format to structured Comma Separated Values (.CSV) files. We used MapReduce in Hadoop to achieve this. Next, we used Spark to read this structured file from Hadoop Distributed File System (HDFS) and do the required processing on the dataset in line with the goal of our use case. One of the main aims of our project was to understand spark optimization techniques. We learnt about the five most common performance issues that can be encountered while executing a spark application, those issues are spill, skew, shuffle, storage and serialisation. From those five we selected the 3 which we believed would help the most at making our application run faster.

1.3 optimisations

Among these, we observed that our spark application can be improved by solving problems related to Spill and Skew (Inter-related in our case), Shuffle and Serialization.

Spill is the term used to refer to the act of moving an RDD from RAM to disk, and later back into RAM again. This occurs when a given partition is simply too large to fit into RAM. In this case, Spark is forced into potentially expensive disk reads and writes to free up local RAM. All of this just to avoid the dreaded OOM (Out of Memory) Error. One of the reasons of Spill could be the Skew in the dataset. Hence, skew can induce spill. We observed that there was skew in our dataset, there was a lot more information on some carriers compared to very little information on other carriers. We also observed in the Spark UI that there was Spill in some of the stages of execution of the application processes.

Shuffle is a side effect of wide transformation caused by operations like join() and group by() on the data. Shuffling refers to transfer of data over the network which adds as overhead time to the actual processing time. Since we are using multiple of these operations in our application, we chose to attempt to minimise this problem.

Serialization occurs when the code has to be serialised, this means that the code is sent to the executors and then de-serialised before it can be executed. Python code will take an even harder hit due to it having to be pickled and the spark must initiate an instance of the python interpreter in every single executer. meanwhile Spark SQL and Dataframe instructions are compact and optimized for distribution instructiosn from the driver to each executor. Our code was using a few User Defined Functions (UDF's) so this was the last of our 3 optimisations.

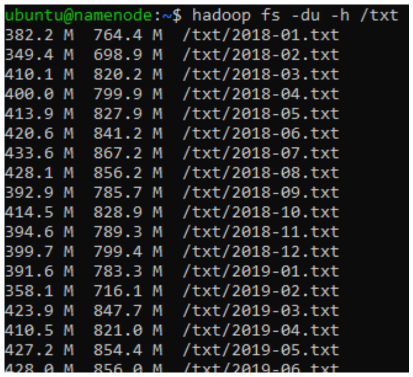
2 METHODS

2.1 Cluster Configuration

The resources for implementing and executing the application was provided to us by the University of Stavanger. We had 1 Name node (Master) with 4GB RAM and 2 CPU cores, and 3 Data nodes (workers) with 8GM RAM, 4 CPU cores and 80GB of disk space on each node, which totals out to 24GB RAM, 12 cores and 240GB disk space for parallel processing on data stored in the HDFS using YARN as the resource manager. Our application is built on Apache Hadoop version 3.2.1 and Spark version 3.2.4. Our Spark application is coded in Python language using the Pyspark(3.2.4) APIs.

2.2 The Dataset

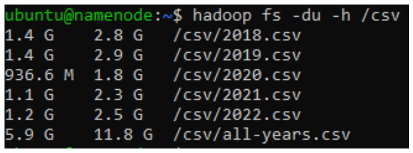
We obtained our dataset from the website of the US Department of Transportation Statistics. The data about the On-Time Performance of Flights was available on the site in monthly format, meaning the data was downloaded in a month by month format. We decided to work with 5 years spanning from 2018-2022. The data in its unsorted text form came out to be a total of 20.7GB. How the files were named can be seen in figure 2.1.



```
ubuntu@namenode:~$ hadoop fs -du -h /txt
382.2 M  764.4 M  /txt/2018-01.txt
349.4 M  698.9 M  /txt/2018-02.txt
410.1 M  820.2 M  /txt/2018-03.txt
400.0 M  799.9 M  /txt/2018-04.txt
413.9 M  827.9 M  /txt/2018-05.txt
420.6 M  841.2 M  /txt/2018-06.txt
433.6 M  867.2 M  /txt/2018-07.txt
428.1 M  856.2 M  /txt/2018-08.txt
392.9 M  785.7 M  /txt/2018-09.txt
414.5 M  828.9 M  /txt/2018-10.txt
394.6 M  789.3 M  /txt/2018-11.txt
399.7 M  799.4 M  /txt/2018-12.txt
391.6 M  783.3 M  /txt/2019-01.txt
358.1 M  716.1 M  /txt/2019-02.txt
423.9 M  847.7 M  /txt/2019-03.txt
410.5 M  821.0 M  /txt/2019-04.txt
427.2 M  854.4 M  /txt/2019-05.txt
428.0 M  856.0 M  /txt/2019-06.txt
```

Figure 2.1: Truncated view of the file system showing the text files

After converting the dataset from text format to CSV using a Map job in Hadoop, the dataset size reduced to 5.9GB having all flight performance attributes in 38 columns. We stored our dataset in CSV format in two ways. First with year by year CSV files, and second all 5 years data in a single CSV file. The idea behind this was that we could easily test the upsert functionality of the delta table using the year by year files while we could test the performance of the application using the larger combined file. The file naming format and sizes can be seen in figure 2.2.



```
ubuntu@namenode:~$ hadoop fs -du -h /csv
1.4 G    2.8 G   /csv/2018.csv
1.4 G    2.9 G   /csv/2019.csv
936.6 M  1.8 G   /csv/2020.csv
1.1 G    2.3 G   /csv/2021.csv
1.2 G    2.5 G   /csv/2022.csv
5.9 G    11.8 G  /csv/all-years.csv
```

Figure 2.2: view of the file system showing the csv files

2.3 Mapping

The dataset from the start is unstructured as a text document where each field is listed sequentially in a [key, value] format as shown in figure 2.3. This meant that to ingest the data and structure it as CSV, we went through each row of the text file, cleaning the data of any commas that might exist and extracting the value into a list element. Once the last key for a single row of data was found the list would be printed and we could move on to the next row. [1]

```
YEAR 2022
QUARTER 4
MONTH 12
DAY_OF_MONTH 1
DAY_OF_WEEK 4
FL_DATE 12/1/2022 12:00:00 AM
OP_UNIQUE_CARRIER 9E
TAIL_NUM N131EV
OP_CARRIER_FL_NUM 4736
ORIGIN_AIRPORT_ID 12323
ORIGIN_AIRPORT_SEQ_ID 1232305
ORIGIN_CITY_MARKET_ID 32323
ORIGIN ILM
ORIGIN_CITY_NAME Wilmington, NC
ORIGIN_STATE_NM North Carolina
DEST_AIRPORT_ID 10397
DEST_AIRPORT_SEQ_ID 1039707
DEST_CITY_MARKET_ID 30397
DEST ATL
DEST_CITY_NAME Atlanta, GA
DEST_STATE_NM Georgia
DEP_DELAY 14.00
DEP_DELAY_NEW 14.00
DEP_DEL15 0.00
ARR_DELAY -2.00
ARR_DELAY_NEW 0.00
ARR_DEL15 0.00
CANCELLED 0.00
CANCELLATION_CODE
DIVERTED 0.00
CARRIER_DELAY
WEATHER_DELAY
NAS_DELAY
SECURITY_DELAY
LATE_AIRCRAFT_DELAY
```

Figure 2.3: The original dataset in text format with a key, value structure

As can be seen in figure 2.3 not all columns of a row would have data in them. In this case we would return an empty string into the list so that the CSV would stay consistent. Additionally we stripped any commas which might exist in the data to avoid any potential issues in the final CSV.

In figure 2.4 it can be seen how we went through each line of the text file, stripping it of commas, then checking if it is the final column of a data row.

```python
import sys

row = []

for line in sys.stdin:
  line = line.strip().replace(',','').split()

  if line[0] == "LATE_AIRCRAFT_DELAY":
    try:
      data = line[1]
    except IndexError:
      data = ' '
    row.append(data)
    print(','.join(row))
    row = []
  else:
    try:
      data = ' '.join(line[1:])
    except IndexError:
      data = ' '
    row.append(data)
```

Figure 2.4: The Hadoop map script

Regardless of if the line was the last column or not we had to make sure that the application wouldn't error out due to not finding any value for the key so we inserted either the value or effectively a null into the list before either continuing to build the list, or printing it.

Below in figure 2.5 an example of how we read in the text files can be seen. Here we read in all the text files for the year 2022 and merged them into a single output file so that we would be working with less files in Pyspark.

```
mapred streaming -files flight_mapper.py \
-input /txt/2022-12.txt,/txt/2022-11.txt,/txt/2022-10.txt,/txt/2022-09.txt,/txt
       /2022-08.txt,/txt/2022-07.txt,/txt/2022-06.txt,/txt/2022-05.txt,/txt
       /2022-04.txt,/txt/2022-03.txt,/txt/2022-02.txt,/txt/2022-01.txt \
-output /csv/2022.csv \
-mapper "flight_mapper.py"
```

Figure 2.5: the Mapred Command

2.3.1 The Data Struct. When reading data into Pyspark we wanted to make sure that it existed in a consistent and logical structure. To do this we made 3 Structs for each type of table which we would read in, these Structs can be seen in fig 2.6. The $flgithschema$ was made to read in all of the data which was created by the Hadoop mapping job. Each field was given the correct type whether that be a string, integer or a float. They were also given their original names from the text file so that we knew what was what.

```python
flightSchema = StructType() \
 .add('YEAR', 'integer')\
 .add('QUARTER', 'integer')\
 .add('MONTH', 'integer')\
 .add('DAY_OF_MONTH', 'integer')\
 .add('DAY_OF_WEEK', 'integer')\
 .add('FL_DATE', 'string')\
 .add('OP_UNIQUE_CARRIER', 'string')\
 .add('TAIL_NUM', 'string')\
 .add('OP_CARRIER_FL_NUM', 'integer')\
 .add('ORIGIN_AIRPORT_ID', 'integer')\
 .add('ORIGIN_AIRPORT_SEQ_ID', 'integer')\
 .add('ORIGIN_CITY_MARKET_ID', 'integer')\
 .add('ORIGIN', 'string')\
 .add('ORIGIN_CITY_NAME', 'string')\
 .add('ORIGIN_STATE_NM', 'string')\
 .add('DEST_AIRPORT_ID', 'integer')\
 .add('DEST_AIRPORT_SEQ_ID', 'integer')\
 .add('DEST_CITY_MARKET_ID', 'integer')\
 .add('DEST', 'string')\
 .add('DEST_CITY_NAME', 'string')\
 .add('DEST_STATE_NM', 'string')\
 .add('DEP_DELAY', 'float')\
 .add('DEP_DELAY_NEW', 'float')\
 .add('DEP_DEL15', 'float')\
 .add('ARR_DELAY', 'float')\
 .add('ARR_DELAY_NEW', 'float')\
 .add('ARR_DEL15', 'float')\
 .add('CANCELLED', 'float')\
 .add('CANCELLATION_CODE', 'float')\
 .add('DIVERTED', 'float')\
 .add('AIR_TIME', 'float')\
 .add('DISTANCE', 'float')\
 .add('DISTANCE_GROUP', 'float')\
 .add('CARRIER_DELAY', 'float')\
 .add('WEATHER_DELAY', 'float')\
 .add('NAS_DELAY', 'float')\
 .add('SECURITY_DELAY', 'float')\
 .add('LATE_AIRCRAFT_DELAY', 'float')

numIdSchema = StructType()\
 .add('id', 'integer')\
  .add('val', 'string')

StringIdSchema = StructType()\
 .add('id', 'string')\
  .add('val', 'string')
```

Figure 2.6: the 3 data Structs used in our Pyspark application

Additionally 2 Structs were created for the 2 types of lookup table that existed for our data. The data did not come with human readable names for the airline, destination airport or origin airport so we had to use lookup tables to get something more readable out of our final output. For this we had 2 types of Struct; one where the keys are a number and another where the key is a string.

2.4 Reading and cleaning the data in Spark

To insert the created CSV into Pyspark we used the *spark.read.csv* function together with the *FlightSchema* previously described. We had to take special consideration for dates to be ingested correctly so we used the *to_date* function to convert the date column to be the correct type.

```python
flight_data = spark.read.csv('hdfs://namenode:9000/csv/'+sys.argv[1]+'.csv',
       schema=flightSchema)\
.withColumn('FL_DATE', to_date(to_timestamp('FL_DATE', 'M/d/yyyy h:mm:ss a')))
```

Figure 2.7: Reading the flight data CSV into Pyspark

As can be seen in figure 2.8 after reading the data we would select only the columns we needed to avoid handling an unnecessarily large data-frames in our algorithm. We would also drop any rows which are not useful to us due to certain columns of essential data missing. For us columns that contained essential data were the *year*, *origin_airport_id*, *dest_airport_id* and the *fl_date*.

In addition to dropping rows that were missing these columns we filled in a default 0.0 value in any rows which had null on the *arr_delay_new* column. This was to play it safe with anything potentially ugly happening during the delay calculations.

```
flight_data = flight_data.select( 'year'
      , 'month'
      , 'fl_date'
      , 'op_unique_carrier'
      , 'origin_airport_id'
      , 'dest_airport_id'
      , 'dep_delay_new'
      , 'arr_delay_new'
      , 'cancelled'
      , 'diverted'
      , 'air_time')

def replace_null(value, default):
  if value is None:
    return default
  return value

def drop_null(*cols):
  for col in cols:
    if col is None:
      return False
  return True

replace_null_udf = udf(lambda value, default: replace_null(value, default),
      FloatType())
drop_null_udf = udf(lambda *cols: drop_null(*cols), BooleanType())

flight_data = flight_data.filter(drop_null_udf(*[col(c) for c in ['year', '
      origin_airport_id', 'dest_airport_id', 'fl_date']]))
flight_data = flight_data.withColumn('arr_delay_new', replace_null_udf(col('
      arr_delay_new'), lit(0.0)))
```

Figure 2.8: Cleaning the dataframe from any unnecessary rows

2.5 Manipulating the data

The data manipulation had to be done in 2 steps, this was due to wanting to extract the day at which the highest delay occurred in addition to all the other flight delay statistics. The resulting data columns were grouped by the *year*, *month*, *op_unique_carrier*, *origin_airport_id* and *dest_airport_id* columns. This means that we are aggregating the data for each year on a month by month basis, for each airline, from a unique location, to a unique destination.

The first part of the data manipulation can be seen in figure 2.9. In this group by statement we sort the data by arrival delay in descending order and then grab the row which had the highest arrival delay. From that row we keep the flight date and the actual arrival delay.

```
windowSpec = Window.partitionBy('year'
    , 'month'
    , 'op_unique_carrier'
    , 'origin_airport_id'
    , 'dest_airport_id').orderBy(col('arr_delay_new').desc())

arr_delay_dates = flight_data.withColumn(
  'rank'
 , rank().over(windowSpec)
).filter(
 col('rank') == 1
).groupBy(
  'year'
 , 'month'
 , 'op_unique_carrier'
 , 'origin_airport_id'
 , 'dest_airport_id'
).agg(
  round(max('arr_delay_new'), 2).alias('max_arr_delay')
 , first('fl_date').alias('max_arr_delay_fl_date')
)
```

Figure 2.9: Pyspark WindowPartition to grab the highest delay, and the date it occurred on

The second part of the data manipulation seen in figure 2.10 handles the rest of the delay statistics, here we used the same group as previously described and with that group we now calculate the average arrival delay, median arrival delay, average time recovered which was the departure delay subtracted by the arrival delay, the number of flights diverted, the average time in the air, the total number of flights and the number of cancelled flights.

Each result is rounded to 2 decimals to avoid overly long numbers. The reason why the number of cancelled flights is a summation is due to that the data uses 1 and 0 as a true and false respectively, this meant that we could just sum the value of the column in all the rows to get the number of cancelled flights.

```
flight_data = flight_data.groupBy(
      'year'
    , 'month'
    , 'op_unique_carrier'
    , 'origin_airport_id'
    , 'dest_airport_id').agg(
        round(avg('arr_delay_new'), 2).alias('avg_arr_delay')
      , round(percentile_approx('arr_delay_new', 0.5), 2).alias('med_arr_delay
    ')
      , round(avg(col('dep_delay_new') - col('arr_delay_new')), 2).alias('
    avg_time_recovered')
      , sum('diverted').alias('nr_diverted')
      , round(avg('air_time'), 2).alias('avg_airtime')
      , count('*').alias('flight_count')
      , sum('cancelled').alias('nr_cancelled'))

flight_data = arr_delay_dates.join( flight_data
    , on=['year', 'month', 'op_unique_carrier', 'origin_airport_id', '
      dest_airport_id']
    , how='left')
```

Figure 2.10: The main group by select statement and aggregate functions

After the second group by we joined the 2 data frames together and also joined all the lookup tables so that the final tables resulting tables could be human readable.

```
airports = spark.read.csv('hdfs://namenode:9000/lookup_tables/airport_id.csv',
        schema=numIdSchema)
carriers = spark.read.csv('hdfs://namenode:9000/lookup_tables/unique_carrier.csv
        ', schema=StringIdSchema)

flight_data = flight_data.join(
  carriers.select('id', col('val').alias('airline'))
  , flight_data['op_unique_carrier'] == carriers['id']
  , how="left"
)

flight_data = flight_data.join(
  airports.select('id', col('val').alias('origin_airport'))
  , flight_data['origin_airport_id'] == airports['id']
  , how="left"
)

airports_alias = airports.alias('airports_alias')
flight_data = flight_data.join(
  airports_alias.select('id', col('val').alias('dest_airport'))
  , flight_data['dest_airport_id'] == airports_alias['id']
  , how="left"
)

flight_data = flight_data.drop("id")
```

Figure 2.11: Joining all the lookup tables into the results table

```
if DeltaTable.isDeltaTable(spark, "hdfs://namenode:9000/spark-warehouse/
        flight_data_table"):
  deltaDF = DeltaTable.forPath(spark, "hdfs://namenode:9000/spark-warehouse/
        flight_data_table")
  merge_condition = "existing.year = upsert.year \
                     AND existing.month = upsert.month \
                     AND existing.op_unique_carrier = upsert.op_unique_carrier \
                     AND existing.origin_airport_id = upsert.origin_airport_id \
                     AND existing.dest_airport_id = upsert.dest_airport_id "

  deltaDF.alias('existing') \
         .merge(flight_data.alias('upsert'), merge_condition) \
         .whenMatchedUpdateAll() \
         .whenNotMatchedInsertAll() \
         .execute()
  print("Delta Table called 'flight_data_table' updated.")
else:
  flight_data.write.format("delta").mode("overwrite").saveAsTable("
        flight_data_table")
  print("Delta Table called 'flight_data_table' created.")
```

Figure 2.12: The upsert statement for the deltatable

2.6 Upserting Data into the Delta Table

After the entire process of structuring, cleaning and manipulating the data as per our use case requirement, we stored our final filtered data into a delta table called $flight\_data\_table$. Our final table consists of 14 columns containing the following information: year and month of the flight, name of the airline, origin and destination airport id and name, maximum arrival delay and the date on which it occurred, average and median arrival delays, average time recovered, number of flights diverted, average airtime, total number of flights and number of flights cancelled. By using appropriate columns, we could plot various kinds of trends in the data, for example, average arrival delay of a particular carrier over a period of 1 year showed us how the arrival delay varies according to months.

We also handled the upsert functionality for our delta table. Upsert is the combination of the words update and insert, what it does is either insert data if it does not already exist or update data if it already exists. To avoid duplication of data in our delta table we implemented the upsert functionality by defining a merge condition. Our merge condition consists of the 5 columns used in the group by statements which together form a combined key for our delta table. Whenever new data is to be added to the delta table a check is done on the year, month, carrier, origin airport and destination airport columns. If all these columns match with some rows that already exist in the delta table, then the rows are replaced. If the merge condition is not met, the new records are inserted into the delta table. The code for the merge condition can be seen in figure 2.12.

2.7 Optimisations

When testing the performance impact of the optimisations we would run a 'base' version of the code which doesn't contain the upsert operation for the deltatable. Instead the upsert was replaced by a count operation. The reason for using the count method was that this operation is optimised and will report significantly shorter times as opposed to a for-each operation, which will include serialisation overhead, which would skew the benchmark operations as described in the literature.

When running the benchmarks we would run it on all 5 years of data from 1 big file, this was to attempt to make the operation as long and slow as possible for testing purposes, the full dataset in a single file came out to be 5.9GB. The way the results were collected was by running each configuration at least 4 times and taking the best time from those runs. The reason for using the best time rather than an averaged is that we found the execution to occasionally be very unreliable in terms of performance so we believed that fastest time is most likely the one with the least outside factors affecting it.1

2.7.1 Spill. The first problem that we tried to solve for optimizing the application is Spill. To check whether our application performance is affected by Spill, we checked the Spark UI, and found that one of the stages of execution was taking a much longer time compared to the others. When going into the details of that stage we found that spill had occurred in that stage. According to the literature spill can be caused by mainly two reasons.

The first being that the data we are processing is skewed in nature, that is there is more data on few certain partitions and less data on remaining partitions, which in turn leads lack of memory for those certain partitions and thus, memory being spilled into the disk. To check if our data is skewed, we created a plot of the data and as can be seen in figure 2.13 there is some skewness in the dataset. On checking solutions to mitigate this problem, we found that Spark 3 has new feature called Adaptive Skewed Join where configurations like *spark.sql.adaptive.skewJoin.enabled* and *'spark.sql.adaptive.advisoryPartitionSizeInBytes'* can be used to solve the skew problem. When delving deeper into this we found that this feature is enabled by default in spark and configured automatically to optimize the performance by eliminating the skew[2]. Thus, spark is capable of handling skew on its own to a very good extent.

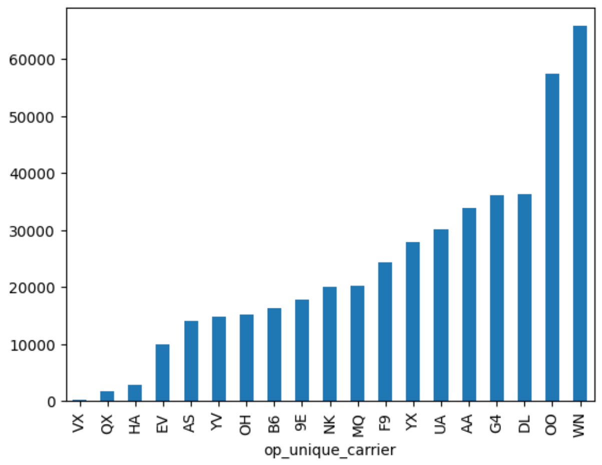
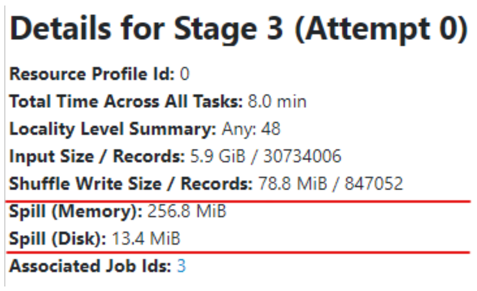


Figure 2.13: Skew Observed in the Data

The other reason for Spill is less partition memory. In this case Spill is caused when there is less memory available for each partition while processing data. While looking into the solutions to mitigate Spill, we found that we can manage the number of partitions that spark uses to handle the data by configuring the parameter *'spark.sql.shuffle.partitions'*. The default value for this is 200. Meaning that spark can create maximum of 200 partitions of our dataset, but it adapts according to the size of the dataset. We observed on the terminal window that for our application spark was using around 55 to 70 partitions. We realized that, since our dataset size is roughly 6GB, and if we want each partition to have minimum 256 MB of memory, we should set the numbers of partitions to 48. Thus, we configured the *'spark.sql.shuffle.partitions'* to 48.


Details for Stage 3 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 8.0 min
Locality Level Summary: Any: 48
Input Size / Records: 5.9 GiB / 30734006
Shuffle Write Size / Records: 78.8 MiB / 847052
Spill (Memory): 256.8 MiB
Spill (Disk): 13.4 MiB
Associated Job Ids: 3

Figure 2.14: Spill Observed in the Data on Spark UI

2.7.2 Serialisation. The data cleaning stage of our application originally was using UDF's which are quite slow compared to the pyspark native functions. In figure 2.15 the new data cleaning functions that replace the ones from figure 2.8 can be seen. These will have the exact same functionality as before but thanks to being native funtions should come with a significant performance increase. As an added bonus these functions also make the code more compact and simple to read.

```
flight_data = flight_data.na.drop(subset=['year', 'origin_airport_id', '
        dest_airport_id', 'fl_date'])
flight_data = flight_data.fillna({'arr_delay_new': 0.0})
```

Figure 2.15: pyspark native data cleaning

2.7.3 Shuffle. Our code uses 4 joins with 3 different tables so making sure that the joins are working as fast as possible is important to our use case . To do this we both tested forcing the code to use broadcast joins and sort merge joins. The broadcast versions of all the joins can be seen in figure 2.16 while the sort merge joins can be seen in figure 2.17. The lookup tables over all we expect to be faster with a broadcast join while for the main table we are uncertain ahead of time which will be faster. Something to keep in mind is that spark will attempt to automatically choose what it believes will be the fastest join so manually forcing a specific type of join could also potentially slow the code down.

```
flight_data = flight_data.join(
  broadcast(carriers.select('id', col('val').alias('airline')))
  , flight_data['op_unique_carrier'] == carriers['id']
  , how='left')

flight_data = flight_data.join(
  broadcast(airports.select('id', col('val').alias('origin_airport')))
  , flight_data['origin_airport_id'] == airports['id']
  , how='left')

airports_alias = airports.alias('airports_alias')
flight_data = flight_data.join(
    broadcast(airports_alias.select('id', col('val').alias('dest_airport')))
  , flight_data['dest_airport_id'] == airports_alias['id']
  , how='left')

flight_data = flight_data.join(
  broadcast(carriers.select('id', col('val').alias('airline')))
  , flight_data['op_unique_carrier'] == carriers['id']
  , how='left')
```

Figure 2.16: broadcast joins

```
flight_data = flight_data.sort('op_unique_carrier')
carriers = carriers.sort('id')
flight_data = flight_data.join(
  carriers.select('id', col('val').alias('airline'))
  , flight_data['op_unique_carrier'] == carriers['id']
  , how="left"
)

flight_data = flight_data.sort('origin_airport_id')
airports = airports.sort('id')
flight_data = flight_data.join(
  airports.select('id', col('val').alias('origin_airport'))
  , flight_data['origin_airport_id'] == airports['id']
  , how="left"
)

airports_alias = airports.alias('airports_alias')
flight_data = flight_data.sort('dest_airport_id')
airports_alias = airports_alias.sort('id')
flight_data = flight_data.join(
  airports_alias.select('id', col('val').alias('dest_airport'))
  , flight_data['dest_airport_id'] == airports_alias['id']
  , how="left"
)
```

Figure 2.17: sort merge joins

When testing the sort merge joins to be certain that pyspark would use them over broadcast joins we also had to make the configuration changes shown in figure 2.18. With these configuration settings we could be certain that the desired joins would be used.

```
spark.sql.join.preferSortMergeJoin                true
spark.sql.adaptive.autoBroadcastJoinThreshold     -1
```

Figure 2.18: spark configurations to force a sort merge join

3 RESULTS

The results from all of our optimisations can be seen in table 3.1. It should be noted that each optimisation builds on top of the last one, this means that after applying an optimisation that optimisation is also present for the next optimisation in the table. The only exception to this are the two shuffle optimisations, since these are affecting the same part of the code these had to replace each other.

Problem to solve	Optimisation	Execution time (min.)
baseline code	-	2.4
Spill	Setting partitions to 48	2
Serialization	Eliminating UDF's	1.6
Shuffle 1	Using Broadcast joins	1.3
Shuffle 2	Using Sort Merge joins	1.4

Table 3.1: the optimisation results

With these results we can see the reducing the spill and getting rid of the UDF's had the largest performance impact. while forcing the code to use one specific type of join made a relatively smaller difference. The difference between the two types of joins is negligible and considering how short the over all run time of the program is it could be argued they are within margin of error of each other.

4 DISCUSSION

The observed results have shown that replacing the UDF's have the larger influence on the execution time. Reducing the allowed number of partitions to 48 also gave some improved results. While tuning the spark.sql.shuffle.partitions parameter we tested with 12, 32 and 64 partition numbers as well. Reducing it to 12 increased the execution time a bit, which seems logical since that would reduce the parallelism factor. Setting the value to 64 performed the same as it did with the default value of 200. The reason for this is our dataset size is too small and spark does self-optimization in many cases and in this case, it was only using 55-70 partitions, even when max value was set as 200. Hence, setting the value to 64 did not show any improvement. The performance for the values 32 and 48 were almost same but we chose 48 as we read that the optimal parallelism level is from 36 to 72 for a dataset with size as small as ours.[3]

The different join techniques did not show any visible improvement in performance. One thing we observed was that Spark was automatically choosing to use broadcast join while executing the application, even when no join was explicitly mentioned. By this we concluded that explicitly mentioning join in the code had no impact on the application performance. In fact spark preferred the broadcast join so strongly for us that settings the configuration to prefer the sort merge join was not enough to get it to use it, we also had to set the broadcast join threshold to -1.

Overall it was difficult to show how much our optimisations actually mattered, this was most likely due to the dataset not being very large and the calculations being relatively simple. But if we had more data, we could experiment with different skew optimization techniques to see how they affect the performance.

A APPENDIX

A link to the GitHub repository with all the code can be found here: https://github.com/sbthepotato/DAT500-Project-23V

REFERENCES

[1] Apache Hadoop 3.2.1. [n. d.]. https://hadoop.apache.org/docs/r3.2.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[2] Apache Spark 3.2.4. [n. d.]. https://spark.apache.org/docs/3.2.4/sql-performance-tuning.html

[3] Sergii Minukhin, Maksym Novikov, Natalia Brynza, and Dmytro Sitnikov. 2020. Experimental research of optimizing the Apache Spark tuning: RDD vs Data Frames. *CEUR-WS* 2608, 31 (2020). https://doi.org/Vol-2608/paper31.pdf

[4] United States Department of Transportation Statistics. [n. d.]. https://www.transtats.bts.gov/Tables.asp