Ex2 - Big Data Engineering - PySpark - Final Report

By: Group 2 - Oshri Mandelawi, Ofek Shaharabani & Idan Kanat

Part A – PySpark Queries:

(Disclaimer: For some queries, only a **part** of the output is presented here, the entire notebook is presented in the HTML file).

Question / Query 1:

04:13 PM (3s) 8

```python
# Query 1: Ranks Airlines by Average Arrival Delay per Year-Month & categorizes them into delay levels

display(
    df
    # Step 1: Add two new columns YEAR and MONTH extracted from the FL_DATE column
    .withColumn("YEAR", F.year(col("FL_DATE")))                      # Extract year from flight date
    .withColumn("MONTH", F.month(col("FL_DATE")))                    # Extract month from flight date

    # Step 2: Group by AIRLINE, YEAR, MONTH and calculate average ARR_DELAY for each group (as demanded in the instructions)
    .groupBy("AIRLINE", "YEAR", "MONTH")
    .agg(F.avg(col("ARR_DELAY")).alias("AVG_ARR_DELAY"))             # Compute average arrival delay

    # Step 3: Add a RANK column, ranking airlines by descending AVG_ARR_DELAY per (YEAR, MONTH)
    .withColumn(
        "RANK",
        F.rank().over(
            Window.partitionBy("YEAR", "MONTH")
                  .orderBy(F.desc("AVG_ARR_DELAY"))                  # Highest average delay gets rank 1
        )
    )

    # Step 4: Add a DELAY_LEVEL column classifying delay severity
    .withColumn(
        "DELAY_LEVEL",
        F.when(col("AVG_ARR_DELAY") > 20, "High")                    # More than 20 mins = High
         .when((col("AVG_ARR_DELAY") >= 10) & (col("AVG_ARR_DELAY") <= 20), "Medium")  # 10-20 mins
         .otherwise("Low")                                           # Less than 10 mins = Low
    )

    # Step 5: Sort the final output for readability
    .orderBy("YEAR", "MONTH", "RANK")                                # Sort by time and rank
)
```

Table

	AIRLINE	YEAR	MONTH	AVG_ARR_DELAY	RANK	DELAY_LEVEL
1	JetBlue Airways	2019	1	14.944398340248963	1	Medium
2	ExpressJet Airlines LLC d/b/a ah...	2019	1	14.542745098039216	2	Medium
3	SkyWest Airlines Inc.	2019	1	10.504681796540233	3	Medium
4	Republic Airline	2019	1	8.696883852691219	4	Low
5	Allegiant Air	2019	1	8.637829912023461	5	Low
6	United Air Lines Inc.	2019	1	7.874431695172115	6	Low
7	Envoy Air	2019	1	7.574979625101874	7	Low
8	Mesa Airlines Inc.	2019	1	5.837235228539576	8	Low
9	Frontier Airlines Inc.	2019	1	5.255102040816326	9	Low
10	Spirit Air Lines	2019	1	4.813245033112583	10	Low
11	American Airlines Inc.	2019	1	3.144564229120665	11	Low
12	Endeavor Air Inc.	2019	1	3.1093518060366154	12	Low
13	PSA Airlines Inc.	2019	1	1.9938407391113067	13	Low
14	Hawaiian Airlines Inc.	2019	1	0.8263473053892215	14	Low
15	Alaska Airlines Inc.	2019	1	0.5597051597051597	15	Low

933 rows | 3.48s runtime

Question / Query 2:

1 minute ago (3s) 10

```python
# Query 2: Computes Average & Standard Deviation of Daily Cancellation Proportions per Airline (Only on Days with >= 100 Flights)

display(
    df
    # Step 1: Add a new DATE column (yyyy-MM-dd) extracted from FL_DATE
    .withColumn("DATE", F.to_date(col("FL_DATE")))                      # Extract date only (without time)

    # Step 2: Group by AIRLINE and DATE to compute:
    # - Total number of flights that day
    # - Number of cancelled flights (just sum the CANCELLED column)
    .groupBy("AIRLINE", "DATE")
    .agg(
        F.count("*").alias("NUM_FLIGHTS"),                              # Total flights on that day
        F.sum("CANCELLED").alias("NUM_CANCELLED")                       # Total cancelled flights on that day
    )

    # Step 3: Keep only days where the airline had at least 100 flights
    .filter(col("NUM_FLIGHTS") >= 100)

    # Step 4: Compute cancellation proportion for each airline on each day
    .withColumn("CANCEL_PROP", col("NUM_CANCELLED") / col("NUM_FLIGHTS"))       # Daily cancellation rate per airline per day

    # Step 5: Group by AIRLINE to compute:
    # - Average of daily cancellation proportion
    # - Standard deviation of daily cancellation proportion
    .groupBy("AIRLINE")
    .agg(
        F.avg("CANCEL_PROP").alias("AVG_DAILY_CANCEL_PROP"),                    # Mean cancellation rate
        F.stddev("CANCEL_PROP").alias("STDDEV_DAILY_CANCEL_PROP")               # Standard Deviation of cancellation rate
    )

    # Step 6: Sort airlines by mean cancellation proportion - AVG_DAILY_CANCEL_PROP in an ascending order (to check for best airlines in busy days)
    .orderBy(F.asc("AVG_DAILY_CANCEL_PROP"))
)
```

Table

	AIRLINE	1.2 AVG_DAILY_CANCEL_PROP	1.2 STDDEV_DAILY_CANCEL_PROP
1	Alaska Airlines Inc.	0	null
2	Spirit Air Lines	0.009900990099009901	null
3	Endeavor Air Inc.	0.013571428571428571	0.05077963596336063
4	Delta Air Lines Inc.	0.015029263037811654	0.061715395069292765
5	JetBlue Airways	0.015636877609398487	0.03921630704199628
6	United Air Lines Inc.	0.01958199154528154	0.06606773288096775
7	PSA Airlines Inc.	0.022536117118279502	0.02198205916130778
8	SkyWest Airlines Inc.	0.023668508955016213	0.05389350477130915
9	American Airlines Inc.	0.02781942363759918	0.07216380406374176
10	Mesa Airlines Inc.	0.03	null
11	Republic Airline	0.0322836871413947	0.0655045991897589
12	Southwest Airlines C...	0.033123867853346006	0.08899490961344449
13	Envoy Air	0.042435688259838095	0.07198500066008853

13 rows | 2.39s runtime

Question / Query 3:

03:48 PM (2s)

13

```python
# Query 3: Identifies the 5 Most Frequent Routes - Airport Pairs (Origin-Destination) with High Average TOTAL Delays (> 45 mins) and Calculates the Share of Flights with No Delay at All

display(
    df
    # Step 1: Compute the Total Delay = Departure Delay + Arrival Delay
    .withColumn("TOTAL_DELAY", col("DEP_DELAY") + col("ARR_DELAY"))

    # Step 2: Group by origin-destination pair and compute:
    # - Total number of flights on the route
    # - Average arrival delay
    # - Percentage of flights with no delay at all (neither departure nor arrival delayed)
    .groupBy("ORIGIN", "DEST")
    .agg(
        F.count("*").alias("NUM_FLIGHTS"),                          # Total number of flights on the route
        F.avg("TOTAL_DELAY").alias("AVG_TOTAL_DELAY"),              # Average arrival delay
        F.round(                                                    # Share of perfectly on-time flights (both delays <= 0)
            F.avg(F.when((col("DEP_DELAY") <= 0) & (col("ARR_DELAY") <= 0), 100).otherwise(0)), 2
        ).alias("PERCENT_NO_DELAY")
    )

    # Step 3: Keep only routes with high average delay (greater than 45 minutes)
    .filter(col("AVG_TOTAL_DELAY") > 45)

    # Step 4: Sort by number of flights descending, to get busiest high-delay routes
    .orderBy(F.desc("NUM_FLIGHTS"))

    # Step 5: Keep only the top / busiest 5 origin-destination pairs
    .limit(5)
)
```

Table

	ORIGIN	DEST	NUM_FLIGHTS	AVG_TOTAL_DELAY	PERCENT_NO_DELAY
1	DEN	ASE	1002	45.32747252747253	43.61
2	MSN	DFW	465	50.956043956043956	49.46
3	ORD	ASE	449	45.14572864321608	41.43
4	ASE	DFW	411	68.67297297297297	44.04
5	ASE	ORD	407	83.11666666666666	43.73

5 rows | 2.12s runtime

Question / Query 4:

04:29 PM (2s) 15

```python
# Query 4: Calculates Average Arrival Delay and the Proportion of On-Time or Early Arrivals by Time-of-Day Category

display(
    df
    # Step 1: Extract departure hour from 4-digit HHMM-format DEP_TIME (e.g., 1430 -> 14)
    .withColumn("DEP_HOUR", (col("DEP_TIME") / 100).cast("int"))

    # Step 2: Assign a time-of-day category
    .withColumn(
        "TIME_OF_DAY",
        F.when((col("DEP_HOUR") >= 5) & (col("DEP_HOUR") < 12), "Morning")       # Morning: 05:00-11:59
         .when((col("DEP_HOUR") >= 12) & (col("DEP_HOUR") < 17), "Afternoon")   # Afternoon: 12:00-16:59
         .when((col("DEP_HOUR") >= 17) & (col("DEP_HOUR") < 21), "Evening")     # Evening: 17:00-20:59
         .otherwise("Night")                                                     # Night: 21:00-04:59
    )

    # Step 3: Group by departure period - TIME_OF_DAY and compute:
    # - Proportion - Mean amount of flights with ARR_DELAY <= 0 (on-time or early)
    # - Average arrival delay
    .groupBy("TIME_OF_DAY")
    .agg(
        F.avg("ARR_DELAY").alias("AVG_ARR_DELAY"),                                              # Mean arrival delay
        F.avg(F.when(col("ARR_DELAY") <= 0, 100).otherwise(0)).alias("ON_TIME_PERCENTAGE")    # Proportion - Average amount of on-time or early arrivals, represented as a percentage
    )

# Step 4: Enforce chronological order of TIME_OF_DAY: Morning → Afternoon → Evening → Night
.orderBy(
    F.when(col("TIME_OF_DAY") == "Morning", 1)
     .when(col("TIME_OF_DAY") == "Afternoon", 2)
     .when(col("TIME_OF_DAY") == "Evening", 3)
     .otherwise(4)
)
)
```

Table

	TIME_OF_DAY	1.2 AVG_ARR_DELAY	1.2 ON_TIME_PERCENTAGE
1	Morning	-2.615741509120811	74.32954100696803
2	Afternoon	4.190383100399732	64.7020519791302
3	Evening	9.210206702890567	59.22994108581634
4	Night	26.64866649466729	36.80513263096188

4 rows | 2.45s runtime

Question / Query 5:

2 minutes ago (2s) 17

```python
# Query 5: Classifies Flight Duration Performance by Observed vs. Scheduled (Expected) Elapsed Time and Summarizing Flight Shares by performance category, and Coverage (Number of Unique Destinations), per Airline

display(
    df
    # Step 1: Filter out degenerate cases: Ensure CRS_ELAPSED_TIME > 0 and ELAPSED_TIME is not null
    .filter((col("CRS_ELAPSED_TIME") > 0) & (col("ELAPSED_TIME").isNotNull()))

    # Step 2: Compute performance ratio = actual observed elapsed time / scheduled elapsed
    .withColumn("PERFORMANCE_RATIO", col("ELAPSED_TIME") / col("CRS_ELAPSED_TIME"))

    # Step 3: Classify each flight into a performance category
    .withColumn(
        "PERFORMANCE_CATEGORY",
        F.when(col("PERFORMANCE_RATIO") < 0.9, "Faster - Significantly")           # Much Faster than expected: Performance Ratio < 0.9
         .when(col("PERFORMANCE_RATIO") <= 1.1, "Moderate / On Time (±10%)")   # Moderate performance / On Time: Performance  Ratio between 0.9 and 1.1
         .otherwise("Slower - Significantly")                                      # Much Slower than expected: Performance Ratio > 1.1
    )

    # Step 4: Count total number of flights per AIRLINE to enable proportion calculation (share of each performance category) later
    .withColumn("NUM_FLIGHTS", F.count("*").over(Window.partitionBy("AIRLINE"))) # used a window function to count the number of flights per airline

    # Step 5: Group by AIRLINE and PERFORMANCE_CATEGORY to compute:
    # - Percentage of flights in each performance group (relative to airline total)
    # - Number of unique destinations served in that category
    .groupBy("AIRLINE", "PERFORMANCE_CATEGORY")
    .agg(
        F.round((F.count("*") / F.first("NUM_FLIGHTS") * 100), 2).alias("PERCENT_FLIGHTS"),   # Percentage (%) of flights in each category
        F.countDistinct("DEST").alias("NUM_UNIQUE_DEST")                                      # Number of unique destinations in each group
    )

    # Step 6: Sort output by AIRLINE and category (alphabetically preserved: Faster → On Time → Slower)
    .orderBy("AIRLINE", "PERFORMANCE_CATEGORY")
)
```

Table

	AIRLINE	PERFORMANCE_CATEGORY	PERCENT_FLIGHTS	NUM_UNIQUE_DEST
1	Alaska Airlines Inc.	Faster - Significantly	10.93	90
2	Alaska Airlines Inc.	Moderate / On Time (±10%)	81.45	91
3	Alaska Airlines Inc.	Slower - Significantly	7.62	91
4	Allegiant Air	Faster - Significantly	8.09	135
5	Allegiant Air	Moderate / On Time (±10%)	82.64	140
6	Allegiant Air	Slower - Significantly	9.27	136
7	American Airlines Inc.	Faster - Significantly	22.17	131
8	American Airlines Inc.	Moderate / On Time (±10%)	71.54	137
9	American Airlines Inc.	Slower - Significantly	6.29	131
10	Delta Air Lines Inc.	Faster - Significantly	26.93	161
11	Delta Air Lines Inc.	Moderate / On Time (±10%)	68.21	163
12	Delta Air Lines Inc.	Slower - Significantly	4.86	155
13	Endeavor Air Inc.	Faster - Significantly	45.89	146
14	Endeavor Air Inc.	Moderate / On Time (±10%)	46.29	147
15	Endeavor Air Inc.	Slower - Significantly	7.82	141

54 rows | 2.00s runtime

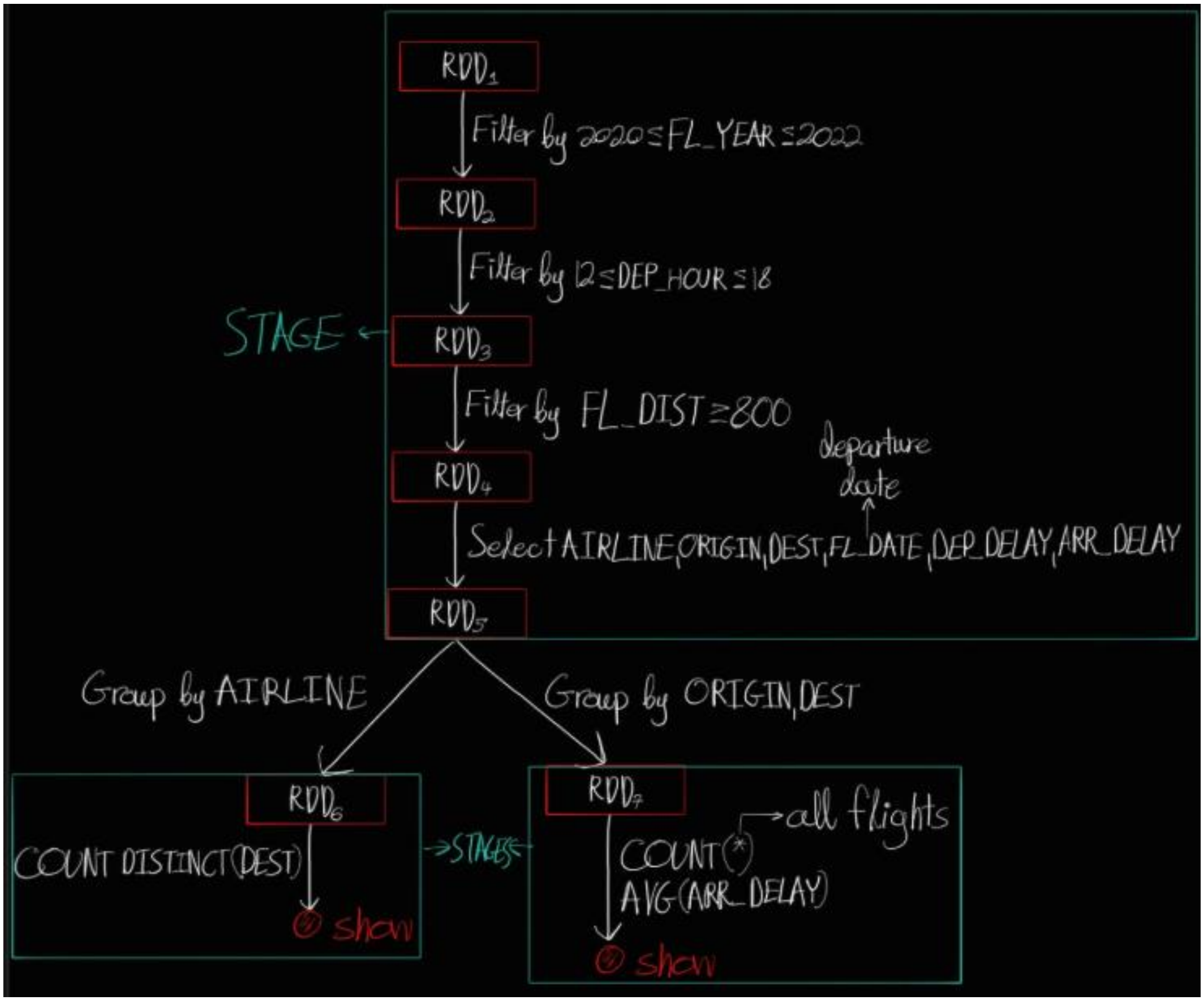
Part B – Theoretical Questions:

1. In Spark, a **transformation** is a **lazy** operation that **defines a new RDD** or DataFrame from an existing one, but it does **not** immediately compute anything. Instead, it builds up a DAG (Directed Acyclic Graph) of steps to be executed later. Examples of transformations include filter(), select(), groupBy(), and withColumn().

An **action**, on the other hand, **triggers Spark to execute all the transformations that have been defined up to that point**. Actions launch a Spark job, compute the result, and either return it to the driver (e.g., show(), collect(), count()) or write it to storage. Actions are the only operations that actually produce output.

In Tal's analysis example, all steps (until the last ones) leading up to the final output are **transformations**: she filters flights based on year (2020-2022), departure hour (12:00-18:00) , and distance (>= 800 kilometers), selects six relevant columns (airline, origin, destination, flight date, departure & arrival delay), groups by origin–destination to compute flight counts and average arrival delays – for the first query, and for the second query: groups by airline to count distinct destination airports. These operations build the logical execution plan but remain unexecuted. Only the final display() or show() steps act as **actions**, triggering Spark to compute the transformations and return the results.

2. **DAG:**



As described in the question's outset, the DAG begins with a series of **narrow transformations** applied to the original RDD:

Filtering operations:

1. Years between 2020–2022.
2. Departure times between 12:00–18:00.
3. Flights with distance >= 800 kilometers.

Selection of relevant columns:

4. Airline, origin, destination, departure delay, arrival delay.

These transformations are **lazy**, meaning they define the lineage but **no computation is executed yet**. All these steps compose a **single stage**, since they do not require a shuffle.

Next, the DAG splits into **two branches** for the two queries Tal wants to perform:

Query 1: Grouping by origin-destination and calculating:

1. Count of flights (action).
2. Average arrival delay (another action).

This involves a **wide transformation** (groupBy), triggering a **shuffle** and forming a **new stage**.

Query 2: Grouping by airline and counting (action) unique destinations.

This also requires a **wide transformation** (groupBy), forming a **separate stage**.

Finally, both branches end with **actions** (show() or display()), which trigger the actual execution of all prior transformations.

3. Some manual optimization was applied in this query. Tal explicitly filtered the data to include only flights from 2020 to 2022, departing between 12:00–18:00, and distance >= 800 km - this reduces the dataset early. She also selected only six relevant columns needed for the analysis. These two steps reduce the amount of data Spark needs to process, which improves efficiency.

Beyond that, Spark adds optimizations automatically. It applies the filters and column selections as early as possible and avoids reading or processing unnecessary data. Additionally, since both queries are based on the same filtered dataset, Spark internally reuses that result to avoid doing the work twice.

A further improvement would be to explicitly cache the filtered dataset before running both queries. This would prevent even the minimal recomputation Spark might do, especially if the dataset is large, and speed up the analysis.