Naya College
Cloud Data Engineering

Apache Spark
Lab3: Data Analysis & Completion

Contents

Description

In this Lab, you will perform a comparison between two datasets: **flights** and **flights_raw**. The aim is to identify matched and unmatched data between them. This is crucial for data cleaning, validation, and reconciliation.

Techniques Covered:

1. **Data Loading**: Read parquet files into Spark DataFrames.
2. **Data De-duplication**: Eliminate duplicate entries within each dataset.
3. **Data Comparison**: Use set operations to find matched and unmatched records between two datasets.
4. **Data Annotation**: Mark the source of the unmatched records.
5. **Data Storage**: Store the final results back into S3 in Parquet format.
6. **Data Generation**: Creating a DataFrame of sequence of dates.
7. **Column Operations**: Using withColumn to add new columns.
8. **Date Functions**: Utilizing PySpark's built-in date functions like dayofweek and dayofmonth.
9. **Aggregation**: Using groupBy and agg for aggregation operations.
10. **Joining DataFrames**: Using join to combine different DataFrames based on common columns.
11. **Data Writing**: Writing DataFrames back to S3 in Parquet format.
12. **Schema inspection**
13. **Sorting and ordering data**
14. **String concatenation in dataframe**

Task1: Data Comparison

Background:

This exercise guides you through the process of comparing two datasets: **flights** and **flights raw**. After deduplication, you will create two datasets to capture matched and unmatched data between the two sources. The matched dataset will contain records present in both datasets, while the unmatched dataset will specify which records are unique to each source.

Instructions:

Setting Up Your PyCharm Workspace:

- **Open PyCharm:** Launch the PyCharm IDE on your machine.
- **Folder Creation**: Right-click in the project pane and select **New > Directory**. Name it **exercises_three**.
- **New Python Script:** Inside the exercises_three folder, right-click and opt for **New > Python File**. Name this script **compare_raw.py**.

SparkSession Initialization:

- As you begin your script, you must first initialize the SparkSession.
- While initializing, **set** the configuration **spark.driver.memory to 4g**. This is crucial to ensure that the driver has enough memory to execute operations without any memory-related issues.

Loading Data:

- Flights Data: Load the flights data stored in Parquet format from the path: **'s3a://spark/data/source/flights/'**
- Flights Raw Data: Similarly, load the flights raw data from: **'s3a://spark/data/source/flights_raw/'**

Data Deduplication:

- Ensure that each dataset is free from duplicate records by using the **dropDuplicates()** method.

Data Comparison:

- Matched Data: To find data present in both datasets, use the **intersect()** method. This will give you a new dataset with records that are common to both.

- Unmatched Data: To get records that are unique to each dataset, use the **subtract()** method. This will subtract the common records from each dataset, giving you the unmatched records.
    - For clarity, add a new column, **source_of_data**, to each unmatched dataset. This column will specify whether the unmatched record originates from **flights** or **flights_raw**.

Saving the Results:

- Matched Data: Save the matched data to **'s3a://spark/data/stg/flight_matched/'** in Parquet format.
- Unmatched Data: Likewise, save the unmatched data to **'s3a://spark/data/stg/flight_unmatched/'**.

Terminating the Session:

- Ensure to stop the SparkSession after all operations are completed.

After following these instructions, you will successfully compare the two datasets and store the results in the specified directories.

Task2: Add Missing Dates

Objective: In this exercise, you will enhance the flights dataset by adding a full date field. This full date represents the most recent occurrence when the **day of month** and **day of week** matched a specific record, starting from December 31, 2020, and going backwards.

Instructions:

Setting Up Your PyCharm Workspace:

- **Open PyCharm:** Launch the PyCharm IDE on your machine.
- **Folder Creation:** Right-click in the project pane and choose **New > Directory**. Name this directory **exercises_three**.
- **New Python Script:** Inside the **exercises_three** folder, right-click and opt for **New > Python File**. Name this script **add_missing_dates.py**.

SparkSession Initialization:

- At the beginning of your script, initialize the SparkSession, which provides an entry point to use Spark's functionality.

Function to Generate Date DataFrame:

- Create a function, **get_dates_df**, that returns a DataFrame with all dates in the year 2020.
- Use the **sequence** function to generate this list of dates.

Function Guide: def get_dates_df():

Purpose: This function is designed to generate a DataFrame containing all dates within the year 2020.

Instructions:

1. **Initialize a Dummy DataFrame:**
    - Begin by creating a DataFrame with a single row and a dummy column. The purpose of this dummy DataFrame is to utilize Spark's **select** and **explode** operations on the upcoming steps.

```
dummy_df = spark.createDataFrame([Row(dummy='x')])
```

2. **Generate a Date Sequence for the Year 2020:**
    - The aim is to generate a list containing all the dates from January 1, 2020, to December 31, 2020.
    - To do this, leverage Spark's **sequence** function. This function can generate a range of dates between two provided date boundaries.

```
date_sequence = F.sequence(F.lit("2020-01-01").cast(T.DateType()), F.lit("2020-12-31").cast(T.DateType()))
```

3. **Explode the Date Sequence:**

- The generated date sequence from the previous step is essentially a list. To convert each date within this list into individual rows within a DataFrame, use the **explode** function.

```
exploded_dates = F.explode(date_sequence)
```

4. **Select and Alias the Exploded Dates:**
   - Now, select the exploded dates from the dummy DataFrame. While doing so, rename the column containing these dates to "flight_date".

```
in_dates_df = dummy_df.select(exploded_dates.alias("flight_date"))
```

5. **Return the Resultant DataFrame:**
   - The function should now return **in_dates_df**, which is a DataFrame where each row corresponds to a date within the year 2020

```
return in_dates_df
```

By following the above instructions, the **get_dates_df()** function should successfully produce a DataFrame with dates for each day in 2020.

Loading the Flights Matched Data:

- Load the flights matched data from the Parquet file located at: '**s3a://spark/data/stg/flight_matched/**'.

Generate Date Information:

- Invoke the **get_dates_df** function to retrieve the DataFrame with dates from 2020.
- Add two new columns to this DataFrame: **day_of_week** and **day_of_month**. These columns are calculated using the **dayofweek** and **dayofmonth** functions respectively on the **flight_date** column.

Determine the Maximum (Latest) Date:

- For each combination of **day_of_week** and **day_of_month**, determine the latest (maximum) date that combination occurred in 2020.
- Use the **groupBy** and **agg** functions to achieve this, saving the result in **max_date_df**.

Enrich the Flights DataFrame:

- Join the original **flights_df** with **max_date_df** using the **day_of_week** and **day_of_month** columns. This will add the **flight_date** column to each record in the **flights_df**, indicating the latest date in 2020 that the **day_of_month** and **day_of_week** combination occurred.

Save the Enriched Flights DataFrame:

- Write the **enriched_flights_df** to the specified directory: '**s3a://spark/data/transformed/flights/**' in Parquet format.

Terminating the Session:

- To release resources and ensure a clean shutdown, stop the SparkSession after all operations are completed.

After following these instructions, you'll have a dataset with records from the flights matched data, enhanced with the latest date from 2020 that matches the **day_of_month** and **day_of_week** of each record.

Task3: Aggregation Analysis

Folder Name: exercises_three
File Name: aggregations_analysis

Objective:
Gain insights from the flights dataset concerning:

- Top 10 airports by departures.
- Top 10 airports by arrivals.
- Top 10 most frequented flight routes.

Instructions:

Setup Your IDE:

- Create a new Python script named aggregations_analysis in the folder exercises_three.

Initialize Your Spark Environment:

- Begin by setting up the Spark session, specifying the app name as ex3_aggregations.
- Configure the Spark session for local execution and allocate appropriate resources.

Load Data:

- Read in the flights and airports datasets stored as Parquet files from the specified S3 paths.

```
flights_df = spark.read.parquet('s3a://spark/data/transformed/flights/')
airports_df = spark.read.parquet(' s3a://spark/data/source/airports/')
```

Inspect Data:

- Examine the schemas of the flights and airports DataFrames to understand their structures using **printSchema()** method.
- Familiarizing yourself with the data columns and types will make the subsequent steps smoother.

Top 10 Airports by Departures:

- Group the **flights** dataset by the **origin_airport_id** to aggregate the number of departures.
- Rename this grouping as **airport_id.**
- Count the number of departures and name this count as **number_of_departures**.
- To translate airport IDs to **airport** names, join this aggregated data with the **airports** dataset. Match **the airport_id** of the aggregated data with that of the airports dataset.
- Sort this data in descending order based on the number of departures and display the top 10 results.

Top 10 Airports by Arrivals:

- This time, group the **flights** dataset by the **dest_airport_id** to count the number of arrivals.
- Rename the column to **airport_id**.
- Count the number of arrivals for each airport.
- Again, join this with the **airports** dataset to get the airport names.
- Display the top 10 airports with the most arrivals.

Top 10 Flight Routes:

- Group the **flights** dataset by both the **origin_airport_id** and **dest_airport_id** to represent unique flight routes.
- Rename these columns to **source_airport_id** and **dest_airport_id**, respectively.
- Count the number of times each route is taken.
- For each source and destination ID, join with the **airports** dataset to get the corresponding airport names.
- Combine the source and destination airport names to form a single track representation like "Source Airport -> Destination Airport".
- Display the top 10 most frequent flight routes.

Clean Up:

- After extracting the insights, ensure you close the Spark session to release any allocated resources.
- Save and backup your code to avoid any loss of work.

Document Your Findings:

- Alongside your code, maintain a document or markdown file detailing the insights you've uncovered. This will be beneficial for presentations or to provide a quick summary without diving into the code.

Version Control (Optional):

- If you're using Git or a similar version control system, commit your changes with a descriptive message. This ensures you can track modifications and collaborate with others.

By following this guide, you'll successfully complete the analysis while maintaining a structured approach.