**Assignment 8 Celebal Technologies**

**Solution1:**

**Step 1: Load NYC Taxi Data into PySpark DataFrame**

Assuming the data is in CSV format stored in Blob Storage, you would first need to mount the Blob Storage to your Databricks workspace or directly access it using the storage account credentials.

Python code

# Mount Blob Storage (if not mounted)

# dbutils.fs.mount(source = "wasbs://<container-name>@<storage-account-name>.blob.core.windows.net",

# mount\_point = "/mnt/<mount-name>",

# extra\_configs = {"<conf-key>":dbutils.secrets.get(scope = "<scope-name>", key = "<key-name>")})

# Read CSV into PySpark DataFrame

file\_path = "/mnt/<mount-name>/path/to/your/nyc\_taxi\_data.csv"

df = spark.read.option("header", "true").csv(file\_path)

**Step 2: Perform Queries Using PySpark**

Now, let’s execute each query as specified:

**Query 1: Add a column named "Revenue"**

python code

from pyspark.sql.functions import col, expr

df = df.withColumn("Revenue",

col("Fare\_amount") + col("Extra") + col("MTA\_tax") +

col("Improvement\_surcharge") + col("Tip\_amount") +

col("Tolls\_amount") + col("Total\_amount"))

**Query 2: Increase count of total passengers by area**

Python code

passengers\_by\_area = df.groupBy("pickup\_area").agg({"passenger\_count": "sum"})

**Query 3: Realtime average fare/total earning amount by 2 vendors**

Python code

avg\_fare\_by\_vendor = df.groupBy("vendor\_id").agg({"fare\_amount": "avg", "total\_amount": "sum"})

**Query 4: Moving count of payments made by each payment mode**

Python code

from pyspark.sql.window import Window

from pyspark.sql.functions import count

windowSpec = Window.partitionBy("payment\_mode").orderBy("timestamp").rangeBetween(-3600, 0)

moving\_count\_by\_payment = df.withColumn("moving\_count", count("payment\_mode").over(windowSpec))

**Query 5: Highest two gaining vendors on a particular date with no of passengers and total distance by cab**

python code

from pyspark.sql.functions import rank, desc, sum

windowSpec = Window.partitionBy("date", "vendor\_id").orderBy(desc("total\_amount"))

top\_gaining\_vendors = df.withColumn("rank", rank().over(windowSpec)) \

.filter(col("rank") <= 2) \

.groupBy("date", "vendor\_id") \

.agg({"passenger\_count": "sum", "trip\_distance": "sum"})

**Query 6: Most number of passengers between a route of two locations**

python code

max\_passengers\_route = df.groupBy("pickup\_location", "dropoff\_location") \

.agg({"passenger\_count": "max"})

**Query 7: Top pickup locations with most passengers in last 5/10 seconds**

Python code

from pyspark.sql.functions import window

df\_with\_timestamp = df.withColumn("timestamp", expr("to\_timestamp(pickup\_datetime)"))

windowSpec = Window.partitionBy("pickup\_location") \

.orderBy(desc("timestamp")) \

.rangeBetween(-10, 0)

top\_pickup\_locations = df\_with\_timestamp.withColumn("pickup\_count", count("passenger\_count").over(windowSpec)) \

.filter(col("pickup\_count") > 0)

**Additional Considerations:**

* **Performance Optimization**: Use appropriate partitioning and caching techniques for large datasets.
* **Data Integrity**: Ensure data quality checks and handle any missing or erroneous data appropriately.
* **Cost Management**: Monitor and optimize costs, especially for cloud services like Databricks and Blob Storage.

**Solution 2:**

To accomplish the tasks of loading a dataset into Databricks File System (DBFS), flattening JSON fields within the dataset, and then writing the flattened data as an external Parquet table, you can follow these steps using PySpark on Databricks:

**Step 1: Load Dataset into DBFS**

Assuming you have a dataset stored in DBFS or you want to upload a file from your local machine to DBFS:

Python code

# Assuming you want to upload a local file to DBFS

# Replace '/local/path/to/your/dataset.json' with your actual local file path

dbutils.fs.cp('file:/local/path/to/your/dataset.json', 'dbfs:/datasets/dataset.json')

**Step 2: Read and Flatten JSON Fields**

Python code

# Read JSON file into PySpark DataFrame

df = spark.read.json('dbfs:/datasets/dataset.json')

# Display schema to understand the structure of JSON

df.printSchema()

# Flatten nested JSON fields

from pyspark.sql.functions import col

# Define function to flatten nested structs

def flatten\_df(df):

complex\_fields = [field.name for field in df.schema.fields if field.dataType.typeName() == 'struct']

for field in complex\_fields:

nested\_fields = [col(field + '.' + nested\_field.name).alias(field + '\_' + nested\_field.name)

for nested\_field in df.schema[field].dataType.fields]

df = df.select("\*", \*nested\_fields).drop(field)

return df

# Flatten the DataFrame

df\_flat = flatten\_df(df)

**Step 3: Write Flattened Data as External Parquet Table**

Python code

# Write flattened DataFrame as external Parquet table

parquet\_table\_path = "dbfs:/datasets/external\_parquet\_table"

df\_flat.write.mode('overwrite').parquet(parquet\_table\_path)

# Register the Parquet table for querying (optional)

spark.sql(f"CREATE TABLE IF NOT EXISTS external\_parquet\_table USING PARQUET LOCATION '{parquet\_table\_path}'")

**Explanation:**

* **Step 1**: The dataset (assumed to be in JSON format) is loaded into DBFS using dbutils.fs.cp.
* **Step 2**: The JSON data is read into a PySpark DataFrame (df). The printSchema() method helps visualize the structure.

The flatten\_df function recursively flattens nested JSON fields by:

* + Identifying complex fields (structs).
  + Constructing new columns for each nested field.
  + Dropping the original nested field after flattening.
* **Step 3**:
  + The flattened DataFrame (df\_flat) is written to an external Parquet table (external\_parquet\_table) stored in DBFS.
  + The WRITE operation mode (overwrite) ensures any existing data at the path is replaced.
  + Optionally, you can register the Parquet table with Spark SQL for easy querying.

**Notes:**

* Adjust paths (dbfs:/datasets/...) and file names (dataset.json) based on your specific setup.
* Ensure permissions and access to DBFS paths are properly configured.
* This example assumes the use of Databricks and DBFS. If using a different environment, adjustments may be needed for file access and path management.
* Consider performance optimizations and schema evolution when handling large datasets or frequently updated data.