Retail Data Analysis Project

Code Logic Explanation

Submitted by:

Prerna Gupta (prerna.gupta1792@gmail.com)

Code Logic- Retail Data Analysis

1. **Importing necessary spark libraries and functions**

```
4 from pyspark.sql import SparkSession
5 from pyspark.sql.functions import *
6 from pyspark.sql.types import *
7 from pyspark.sql.window import Window
8
```

The code starts by importing the necessary libraries.

- SparkSession is the entry point to programming Spark with the DataFrame API. It allows you to create a DataFrame and execute SQL queries on the data.
- Second line imports all functions from the pyspark.sql.functions module. These functions are used to manipulate data within DataFrames. Examples include functions for data aggregation, filtering, and transformation such as sum, avg, etc.
- Third line imports all types from the pyspark.sql.types module. These types are used to define the schema for DataFrames. Examples include StringType, IntegerType, StructType, StructField, etc.
- Fourth line imports the Window class from the pyspark.sql.window module. Window class is used to specify windowing functions (such as ranking, aggregations etc.) that operate over a specified range of rows relative to the current row.

2. **Creating Spark Session**

```
9  #creating SparkSession
10 spark = SparkSession  \
11     .builder  \
12     .appName("RetailDataKPI")  \
13     .getOrCreate()
14 spark.sparkContext.setLogLevel('ERROR')
```

Starting the spark session with RetailDataKPI name to establish connection to the spark cluster. getorCreate() method is used to either create a new SparkSession if one does not already exist or return the existing SparkSession. spark.sparkContext.setLogLevel('ERROR') sets the log level for the Spark context to ERROR, so that only error messages will be logged, which can help reduce the amount of log output and make it easier to spot issues.

3. Creating UDFs (User Defined Functions)

- **is_Order:** Defining a function using if-else statement to check if the type of order is a new order, returning 1 if it's a new order.

```
# Creating UDFs
#Checking for a new order
def is_order(type):
   if type=="ORDER":
       return 1
   else:
       return 0
```

- **is_return:** Defining a function using if-else statement to check if it's a return order, returning 1 if it's a return.

```
#Checking for return of an order
def is_return(type):
   if type=="RETURN":
       return 1
   else:
       return 0
```

- **total_items:** Creating a function to calculate the total number of items in an order. Using for loop summed up the quantities of each product in a particular invoice to get the number of total items associated with a particular transaction.

```
# Calculating the total number of items
def total_items(items):
    if items is not None:
        item_count =0
        for item in items:
            item_count = item_count + item['quantity']
        return item_count
```

- **total_cost :** Defining a function to calculate the total cost of a particular transaction. Calculated the cost by multiplying the unit price with quantity and adding it for all products in the transaction. If it's a return transaction then multiplying the cost by -1 as it will not contribute to the revenue.

```
# Calculating the total cost of the order
def total_cost(items,type):
    if items is not None:
        total_cost =0
        item_price =0
    for item in items:
        item_price = (item['quantity']*item['unit_price'])
        total_cost = total_cost+ item_price
        item_price=0
    if type  == 'RETURN':
        return total_cost *(-1)
    else:
        return total_cost
```

4. Registering UDFs

```
53 # Register UDFs
54 is_order = udf(is_order, IntegerType())
55 is_return = udf(is_return, IntegerType())
56 add_total_items = udf(total_items, IntegerType())
57 add_total_cost = udf(total_cost, FloatType())
```

Each line registers a Python function (is_order, is_return, total_items, total_cost) as a UDF in PySpark, specifying the expected return type for each function. These UDFs will be used to transform streaming data and adding columns to the batch output in console.

5. Reading Data from Kafka topic

```
61 # Reading input data from Kafka topic
62 raw_data_stream = spark \
63     .readStream \
64     .format("kafka") \
65     .option("kafka.bootstrap.servers","18.211.252.152:9092") \
66     .option("subscribe","real-time-project") \
67     .option("startingOffsets", "latest")  \
68     .load()
```

Above code sets up a PySpark streaming DataFrame (raw_data_stream) to read real-time data from a Kafka topic named **'real-time-project'**. It connects to a Kafka broker at **18.211.252.152:9092** and starts reading from the latest offset.

6. Defining the Schema

```
70 # Defininig json Schema
71 json_Schema = StructType() \
72     .add("invoice_no", LongType()) \
73     .add("country",StringType()) \
74     .add("timestamp", TimestampType()) \
75     .add("type", StringType()) \
76     .add("items", ArrayType(StructType([
77         StructField("SKU", StringType()),
78         StructField("title", StringType()),
79         StructField("unit_price", FloatType()),
80         StructField("quantity", IntegerType())
81         ])))
```

Defining the schema for the incoming json data from the kafka topic.

7. Deserialize Kafka Messages

```
82
83 orders_stream_data = raw_data_stream.select(from_json(col("value").cast("string"), json_Schema).alias("data")).select("data.*")
84
```

Above code Reads the value column from Kafka messages, converts it to a string, parses the JSON data using the provided schema, and aliases it as data after that it selects all fields from the parsed JSON data, making them individual columns in the resulting DataFrame.

8. Adding new columns

```
# Calculating additional columns from the stream
orders_stream_output = orders_stream_data \
    .withColumn("total_cost", add_total_cost(orders_stream_data.items,orders_stream_data.type)) \
    .withColumn("total_items", add_total_items(orders_stream_data.items)) \
    .withColumn("is_order", is_order(orders_stream_data.type)) \
    .withColumn("is_return", is_return(orders_stream_data.type))
```

Above code adds the 4 more columns to the dataframe using the UDFs created in the beginning of the code by taking the streaming data as input and transforming it.

9. Writing to Console

```
# Writing the summarised input table to the console
orders_batch = orders_stream_output \
    .select("invoice_no", "country", "timestamp","total_cost","total_items","is_order","is_return") \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .option("truncate", "false") \
    .option("path", "/Console_output") \
    .option("checkpointLocation", "/Console_output_checkpoints") \
    .trigger(processingTime="1 minute") \
    .start()
```

Above code lines selects the specific columns from the orders_stream_output DataFrame and writes it to console in append mode every 1 minute and also maintains checkpoints to keep track of the streaming state and progress.

10. Calculating Time Based KPIs

```
# Calculating Time based KPIs
agg_on_time = orders_stream_output \
    .withWatermark("timestamp","1 minutes") \
    .groupby(window("timestamp", "1 minute")) \
    .agg(sum("total_cost").alias("total_volume_of_sales"),
        avg("total_cost").alias("average_transaction_size"),
        count("invoice_no").alias("OPM"),
        avg("is_return").alias("rate_of_return")) \
    .select("window.start","window.end","OPM","total_volume_of_sales","average_transaction_size","rate_of_return")

```

Above processes streaming order data, aggregates it in 1-minute intervals, and calculates several metrics like total sales volume, average transaction size, number of transactions, and return rate for each 1-minute window, then selects the required columns and stores in agg_on_time DF.

11. Calculating Time and Country based KPIs

```
# Calculating Time and country based KPIs
agg_on_time_country = orders_stream_output \
    .withWatermark("timestamp", "1 minutes") \
    .groupBy(window("timestamp", "1 minutes"), "country") \
    .agg(sum("total_cost").alias("total_volume_of_sales"),
        count("invoice_no").alias("OPM"),
        avg("is_return").alias("rate_of_return")) \
    .select("window.start","window.end","country", "OPM","total_volume_of_sales","rate_of_return")
```

Above code lines processes streaming data and calculate the KPIs like Order per minute (OPM), total volume of sales and rate of return in a 1-minute interval and then aggregating the results on country in the 1-minute window.

12. Writing to HDFS

```
# Writing to the json files in hdfs : Time based KPI values
ByTime = agg_on_time.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate", "false") \
    .option("path", "timeKPIvalue") \
    .option("checkpointLocation", "timeKPIvalue_cp") \
    .trigger(processingTime="1 minutes") \
    .start()

# Writing to the hdfs : Time and country based KPI values
ByTime_country = agg_on_time_country.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate", "false") \
    .option("path", "time_countryKPIvalue") \
    .option("checkpointLocation", "time_countryKPIvalue_cp") \
    .trigger(processingTime="1 minutes") \
    .start()
```

Above code lines write the Kpis calculated based on time and based time-country into json files in append mode in HDFS and also maintains the checkpoint information for a processing time of 1 minute.

13. Await Termination

```
orders_batch.awaitTermination()
ByTime.awaitTermination()
ByTime_country.awaitTermination()
```

Above 3 lines are used to keep the streaming jobs running and continuously processing incoming data. They ensure that the Spark application does not exit immediately after starting but remains active to handle streaming data as it arrives.

14. Running the pyspark script

[hadoop@ip-172-31-7-149 ~]$ export SPARK_KAFKA_VERSION=0.10

[hadoop@ip-172-31-7-149 ~]$ spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.1 spark-streaming.py

Above commands are used to submit the spark job and read data from kafka. In essence, this command runs a Spark Streaming application defined in **spark-streaming.py**, with the necessary Kafka connector (spark-sql-kafka-0-10_2.12:3.5.1) to enable integration between Spark and Kafka. The export SPARK_KAFKA_VERSION=0.10 line ensures compatibility with **Kafka version 0.10**.