Code Logic - Retail Data Analysis

The below Python Script is used to process the input data streams into the resultant JSON files

spark-streaming.py

Setting up the libraries and modules

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
```

The following functions are used to calculate the various attributes

1. Total Cost UDF - Total cost of an order arrived at by summing up the cost of all products in that invoice. The return cost is treated as a loss. Thus, for return orders, this value will be negative. To calculate the income from sale of each product, the unit price of the product is multiplied with the quantity of the product purchased.

```
def calculate_total_cost(items, trn_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 trn_type == "RETURN":
            return total_cost * -1
        else:
            return total_cost
```

2. Total Items UDF - Total number of items present in an order. To calculate the number of products in every invoice the quantity ordered of each product in that invoice is summed up

```
def calculate_total_item_count(items):
    if items is not None:
        total_count = 0
        for item in items:
            total_count = total_count + item['quantity']
        return total_count
```

3. Is Order UDF - This flag denotes whether an order is a new order or not. If this invoice is for a return order, the value should be 0. To determine if invoice is for an order or not an if-else statement is used

```
def flag_isOrder(trn_type):
    if trn_type == "ORDER":
        return(1)
    else:
        return(0)
```

4. Is Return UDF - This flag denotes whether an order is a return order or not. If this invoice is for a new sales order, the value should be 0. To determine if invoice is for a return or not an if-else statement is used

```
def flag_isReturn(trn_type):
    if trn_type == "RETURN":
        return(1)
    else:
        return(0)
```

Initialising the Spark session and setting the log level to error

```
spark = SparkSession  \
    .builder  \
    .appName("spark-streaming")  \
    .getOrCreate()
spark.sparkContext.setLogLevel('ERROR')
```

Reading input data from Kafka using sparkreadStream in 'retail_corp_orderData' mentioning the details of the Kafka broker, such as bootstrap server, port and topic name

```
retail_corp_orderData = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "18.211.252.152:9092") \
    .option("startingOffsets","earliest") \
    .option("failOnDataLoss", "false") \
    .option("subscribe", "real-time-project") \
    .load()
```

Defining JSON schema of each order, using appropriate datatypes and StrucField in the case of item attribute

```
jsonSchema = StructType() \
    .add("invoice_no", LongType()) \
    .add("country", StringType()) \
    .add("timestamp", TimestampType()) \
    .add("type", StringType()) \
    .add("items", ArrayType(StructType([
    StructField("SKU", StringType()),
    StructField("title", StringType()),
    StructField("unit_price", FloatType()),
    StructField("quantity", IntegerType()),
])))
```

Reading the raw JSON data from Kafka as 'order stream' by casting it to string and storing it with the alias 'data'

```
orderStream = retail_corp_orderData.select(from_json(col("value").cast("string"),
jsonSchema).alias("data")).select("data.*")
```

Defining the UDFs by Converting the Python functions defined earlier, and assigning the appropriate return datatype

```
sum_total_order_cost = udf(calculate_total_cost, FloatType())
sum_total_item_count = udf(calculate_total_item_count, IntegerType())
sum_isOrder = udf(flag_isOrder, IntegerType())
sum_isReturn = udf (flag_isReturn, IntegerType())
```

Calculating the additional columns according to the required input values

```
updatedOrderStream = orderStream \
    .withColumn("total_cost", sum_total_order_cost(orderStream.items, orderStream.type)) \
    .withColumn("total_items", sum_total_item_count(orderStream.items)) \
    .withColumn("is_order", sum_isOrder(orderStream.type)) \
    .withColumn("is_return", sum_isReturn(orderStream.type))
```

Writing the summarized input values to console, using 'append' output method and applying truncate as false and setting the processing time to 1 minute

```
extendedOrderQuery = updatedOrderStream \
    .select("invoice_no", "country", "timestamp", "total_cost", "total_items", "is_order", "is_return")
\
    .writeStream \
    .outputMode("append") \
```

```
    .format("console") \
    .option("truncate", "false") \
    .trigger(processingTime = "1 minute") \
    .start()
```

Calculating time-based KPIs (Total sale volume, OPM, Rate of return, Average transaction size) having tumbling window of one minute and watermark of one minute.

```
timebasedKPI = updatedOrderStream \
    .withWatermark("timestamp","1 minute") \
    .groupBy(window("timestamp", "1 minute", "1 minute")) \
    .agg(sum("total_cost").alias("total_sale_volume"),
         count("invoice_no").alias("OPM"),
         avg("is_return").alias("rate_of_return"),
         avg("total_cost").alias("average_transaction_size")
        ) \
    .select("window", "OPM", "total_sale_volume", "average_transaction_size", "rate_of_return" )
```

Writing the time-based KPIs data to HDFS - HDFS into JSON files for each one-minute window, using 'append' output mode, setting truncate as false, and specifying the HDFS output path for both the KPI files and for their checkpoints. Ten 1-minute window batches were taken.

```
queryByTime = timebasedKPI.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate","false") \
    .option("path","/user/ec2-user/time_kpi") \
    .option("checkpointLocation","/user/ec2-user/time_kpi_checkpoints") \
    .trigger(processingTime="1 minute") \
    .start()
```

Calculating time-and-country-based KPIs (Total sale volume, OPM, Rate of return) having tumbling window of one minute and watermark of one minute. Here the data is grouped by window and country both

```
timecountrybasedKPI = updatedOrderStream \
    .withWatermark("timestamp", "1 minute") \
    .groupBy(window("timestamp", "1 minute", "1 minute"), "country") \
    .agg(sum("total_cost").alias("total_sale_volume"),
         count("invoice_no").alias("OPM"),
         avg("is_return").alias("rate_of_return")) \
    .select("window", "country", "OPM", "total_sale_volume", "rate_of_return" )
```

Writing the time-and-country-based KPIs data to HDFS into JSON files for each one-minute window, using 'append' output mode, setting truncate as false, and specifying the HDFS output path for both the KPI files and for their checkpoints. Ten 1-minute window batches were taken.

```
queryByCountry = timecountrybasedKPI.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate","false") \
    .option("path","/user/ec2-user/country_kpi") \
    .option("checkpointLocation","/user/ec2-user/country_kpi_checkpoints") \
    .trigger(processingTime="1 minute") \
    .start()
```

Indicating Spark to await termination

```
extendedOrderQuery.awaitTermination()
queryByCountry.awaitTermination()
queryByTime.awaitTermination()
```

Console Command

To execute this project, I have used an EMR cluster along with an EC2 Kafka Instance.

Step I – Creation and initiation of the EC2 Kafka Instance

The steps to set up the EC2 instance are as follows

i. From the AWS Management Console, selecting the EC2 service and under Images selecting AMIs, from there selecting 'Public Images'. Select the AMI ID 'ami-0b47786b148f4dd16'
ii. Selecting the configure security group details and adding ports for Kafka module
iii. Reviewing and launching the instance
iv. Selecting the Network and Security and creating Elastic IP and associate it with the EC2 Kafka Instance
v. Logging in the EC2 instance as 'ec2-user' using Putty terminal and changing the directory to /home/ec2-user/downloads using 'cd' command
vi. Directing to the kafka directory using cd /home/ec2-user/downloads/kafka_2.12-2.3.0 and to go inside the config directory using cd config/
vii. To make changes to the server.properties file, edit this file enter the command vi server.properties. In place of your.host.name enter the IPv4 Public IP of your EC2 instance.

viii. To the Kafka directory using the cd kafka_2.12-2.3.0/ command and then start the Zookeeper using
bin/zookeeper-server-start.sh config/zookeeper.properties

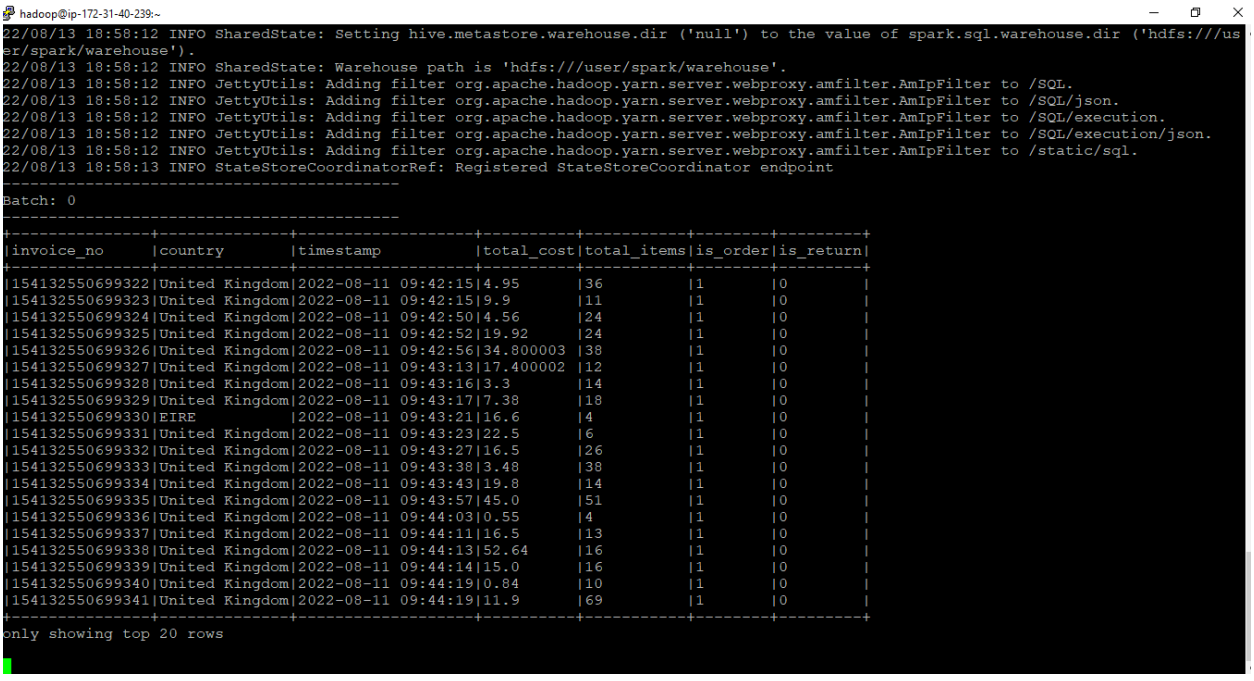
ix. In the new terminal go to the Kafka directory using the cd kafka_2.12-2.3.0/ command and Start the Kafka server using
bin/kafka-server-start.sh config/server.properties

x. The producer is started using
bin/kafka-console-producer.sh --broker-list 18.211.252.152:9092 --topic real-time-project

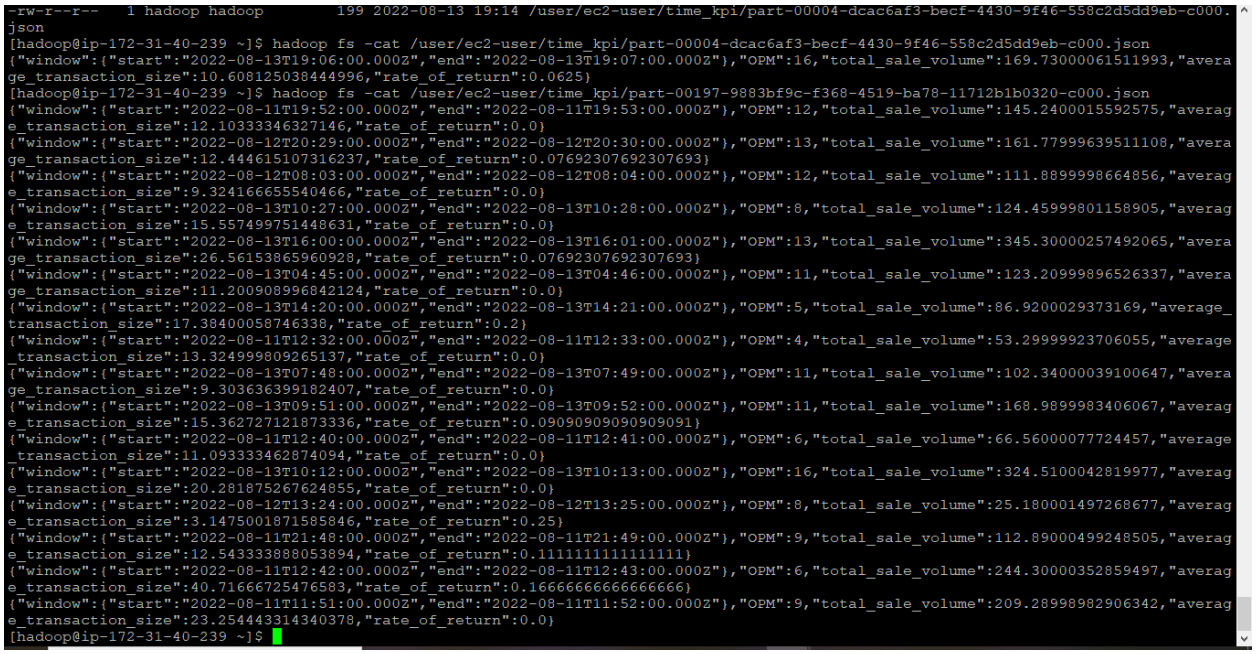
Step II – Creation and initiation of the EMR Cluster

i. Creation of python script file using vi spark-streaming.py and adding put code in that
ii. Exporting Spark Kafka version using export SPARK_KAFKA_VERSION=0.10
iii. Hitting the below spark submit command to store the batches in a file

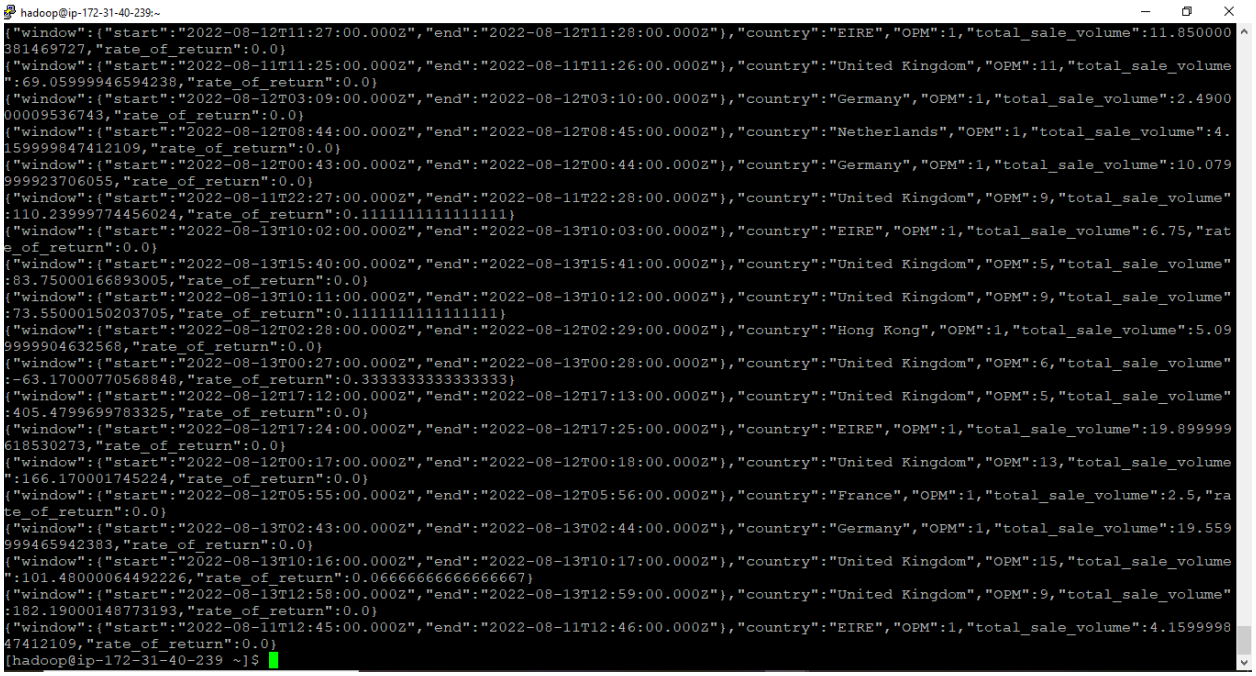
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5 spark-streaming.py>console_output



iv. Checking the hdfs locations for both time and time-country based KPIs
For time based KPI
hadoop fs –ls /user/ec2-user/time_kpi
Using the cat command opening one of the json file for time KPI
hadoop fs –cat /user/ec2-user/time_kpi/part-00004-dcac6af3-becf-4430-9f46-558c2d5dd9eb-c000.json



For time-country based KPI
hadoop fs –ls /user/ec2-user/country_kpi
Using the cat command opening for one of the json file for a country KPI
hadoop fs –cat /user/ec2-user/country_kpi/part-00004-9b289b0c-7588-4db9-a936-396897670239-c000.json



Transferring files from hadoop using get command and then using WinSCP transferring it to local machine

Command for getting time based KPI

hadoop fs –get /user/ec2-user/time_kpi

Command for getting time based KPI

hadoop fs –get /user/ec2-user/country_kpi