Code Logic - Retail Data Analysis

<u>Commands used in addition to spark-streaming.py:</u>

1. Downloaded kafka.

   ```
   wget https://downloads.apache.org/kafka/3.5.0/kafka-3.5.0-src.tgz
   ```
2. Unzipped kafka file.

   ```
   tar -xvf kafka-3.5.0-src.tgz
   ```
3. Command used to store pyspark code.

   ```
   vi spark-streaming.py
   ```
4. Running spark-submit job.
   ```
   export SPARK_KAFKA_VERSION=0.10
   ```

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

<u>Code Explanation:</u>

1. Imported the required libraries.

```python
# Import Dependencies
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import Window
```

2. Defined the utility functions:
   a. <u>util_total_cost:</u> It takes item array and type column as input parameter and calculates total cost of each invoice and returns negative of the total_cost incase it's a return order type else return total_cost as it is if it's a normal order.
   b. <u>util_total_items:</u> It takes item array as input and returns the length of array to determine the number of items in each invoice.
   c. <u>util_is_order:</u> It takes type column as input and returns 1 if the value in this column is ORDER and 0 otherwise.
   d. <u>util_is_return:</u> It takes type column as input and returns 1 if the value in this column is RETURN and 0 otherwise.

```python
# Utility method for checking total cost for each invoice
def util_total_cost(items,type):
   total_price = 0
   for item in items:
       total_price = total_price + item['unit_price'] * item['quantity']
   if type=="RETURN":
       return total_price * (-1)
   else:
       return total_price
```

```
# Utility method for checking total items count for each invoice
def util_total_item_count(items):
   return len(items)

# Utility method for checking order type for each invoice
def util_is_order(column):
    if column == "ORDER":
        return 1
    else:
        return 0

# Utility method for checking return type for each invoice
def util_is_return(column):
    if column == "RETURN":
        return 1
    else:
        return 0
```

3. Created a spark session with appName "RetailDataAnalysis" and used sparkContext.setLoglevel('ERROR') to display only log messages with severity 'ERROR' or higher and suppress other log messages with lower severity.

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

4. Read Kafka stream using readStream method with format "kafka" and provided bootstrap server details and topic name to subscribe to.

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

5. Defined the schema to store the input stream as well as the custom columns that will be created using UDFs. Here, ArrayType is used for the items array.

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

6. Converted the value to string using the schema since the input stream is received as a key, value pair with key null and input in json format in value.

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

7. Converted the utility functions defined to UDF.

```
total_cost = udf(util_total_cost, FloatType())
total_item_count = udf(util_total_item_count, IntegerType())
is_order = udf(util_is_order, IntegerType())
is_return = udf(util_is_return, IntegerType())
```

8. Prepared the final input stream by adding custom columns to our source input stream. Custom columns total_items, total_cost, is_order and is_return are calculated using the UDFs.

```
orderInputStream = orderStream \
       .withColumn("total_items", total_item_count(orderStream.items)) \
       .withColumn("total_cost",
total_cost(orderStream.items,orderStream.type)) \
       .withColumn("is_order", is_order(orderStream.type)) \
       .withColumn("is_return", is_return(orderStream.type))
```

9. Selected the required columns and wrote them to console in append mode using writeStream method with processing time 1 minute and specifying checkpoint location to keep track of last data read.

```
order_query_console = orderInputStream \
       .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") \
       .option("checkpointLocation", "checkpoint/") \
       .start()
```

10. Calculated time based KPIs using .groupby to group on the basis of timestamp and .agg to calculate the KPIs using aggregate functions. Window function is used to determine a tumbling window of 1 minute and withWatermark is used to handle late arriving data.

```
aggTimeKPI = orderInputStream \
    .withWatermark("timestamp","1 minute") \
    .groupby(window("timestamp", "1 minute","1 minute")) \
    .agg(format_number(sum("total_cost"),2).alias("total_sales_volume"),
        format_number(avg("total_cost"),2).alias("average_transaction_size"),
        format_number(avg("is_Return"),2).alias("rate_of_return")) \
    .select("window.start","window.end","total_sales_volume","average_transact
ion_size","rate_of_return")
```

11. Calculated time and country based KPIs using .groupby to group on the basis of timestamp and country and .agg to calculate the KPIs using aggregate functions. Window function is used to determine a tumbling window of 1 minute and withWatermark is used to handle late arriving data.

```
aggTimeCountryKPI = orderInputStream \
    .withWatermark("timestamp", "1 minute") \
    .groupBy(window("timestamp", "1 minute", "1 minute"), "country") \
    .agg(format_number(sum("total_cost"),2).alias("total_sales_volume"),
        count("invoice_no").alias("OPM"),
        format_number(avg("is_Return"),2).alias("rate_of_return")) \
    .select("window.start","window.end","country",
"OPM","total_sales_volume","rate_of_return")
```

12. Wrote the time based KPIs as json files for 1 minute window.

```
queryTimeKPI = aggTimeKPI.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate", "false") \
    .option("path", "time_kpi/") \
    .option("checkpointLocation", "checkpoint/time_kpi/") \
    .trigger(processingTime="1 minutes") \
    .start()
```

13. Wrote the time and country based KPIs as json files for 1 minute window.

```
queryTimeCountryKPI = aggTimeCountryKPI.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate", "false") \
    .option("path", "country_kpi/") \
    .option("checkpointLocation", "checkpoint/country_kpi/") \
    .trigger(processingTime="1 minutes") \
    .start()
```

14. awaitTermination is used to continuously read data in real time.

```
queryTimeCountryKPI.awaitTermination()
```