Code Logic - Retail Data Analysis

Below is the step by step explanation of the code logic and flow:

1. Imported all the required libraries for this project.

```
# Importing required libraries

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

2. Defined separate utility functions to determine total items in an order, total cost of an order and whether the order is a new buy order or a return order.
    a. Utility function to determine total items in an order i.e. sum of the quantities of each item in the order.

```
def get_total_item_count(items):
    """
    This method is used to calculate the toal number of items in an order.
    :param items: The list of items in an order.
    :return total_count: The sum of quantities of each item in the order.
    """
    total_count = 0
    for item in items:
        total_count = total_count + int(item['quantity'])
    return total_count
```

    b. Utility function to determine total cost of an order i.e. sum of quantity multiplied by unit price for each item in the order. If the order is a new buy order, cost is returned else if the order is a return order, then cost is multiplied by -1 and then returned.

```
def get_total_order_cost(items, order_type):
    """
    This method is used to calculate the total cost of an order.
    :param items: The list of items in an order.
    :param order_type: This determines whether the order is a fresh one or a return.
    :return cost: This return sum of quantity * unit_price for each item inthe order. If the order is a return order, the total cost is multiplied my -1.
    """
    cost = 0
    for item in items:
        cost = cost + (int(item['quantity'])*float(item['unit_price']))
    if order_type == 'ORDER':
        return cost
    else:
        return -1*cost
```

c. Utility function to determine the value if ‘is_order’ column. If the order is a new buy order, the value is set to 1 else 0.

```
35  def get_is_order(order):
36      """
37      This method is to determine whether a the order is a new buy order or not.
38      :param order: Detrmines the type of order on which the value of 'is_order' column is set.
39      :return 0/1: Return 1 if order is a new buy order else return 0.
40      """
41      if order == 'ORDER':
42          return 1
43      else:
44          return 0
45
```

d. Utility function to determine the value if ‘is_return’ column. If the order is a return order, the value is set to 1 else 0.

```
46  def get_is_return(order):
47      """
48      This method is to determine whether a the order is a return order or not.
49      :param order: Detrmines the type of order on which the value of 'is_return' column is set.
50      :return 0/1: Return 1 if order is a returrn order else return 0.
51      """
52      if order == 'RETURN':
53          return 1
54      else:
55          return 0
56
```

3. Establishing the spark session and setting the spark log level to ‘ERROR’.

```
57  # Setting up the spark session
58
59  spark = SparkSession \
60          .builder \
61          .appName('RetailDataAnalysis') \
62          .getOrCreate()
63
64  # Setting the spark log level to ERROR.
65
66  spark.sparkContext.setLogLevel('ERROR')
```

4. Connect to the given kafka server and topic using the spark session created above from where the data is to be read.

```
# Reading data from the given kafka server and topic

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

5. Defining the JSON schema for the reading the data from kafka topic

```
# Defining the json Schema of the data to read it properly.

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

6. Casting the raw data as a string and aliasing it to data. In addition to this, defining the UDF's (user defined functions) to calculate additional columns 'total_items', 'total_cost', 'is_order' and 'is_return' using the utility functions defined in the starting.

```
# Coverting raw data to a string and aliasing it as data.

orderStream = orderRaw.select(from_json(col('value').cast('string'), jsonSchema).alias('data')).select('data.*')

# Defining a UDF's with utility functions for calculating different column values.

# UDF to get to total items in an order.
add_total_item_count = udf(get_total_item_count, IntegerType())

# UDF to get total cost of an order.
add_total_order_cost = udf(get_total_order_cost, FloatType())

# UDF to determine if an order is an order is a new buy order.
add_is_order = udf(get_is_order, IntegerType())

# UDF to determine if an order is a return order.
add_is_return = udf(get_is_return, IntegerType())
```

7. Calculating the additional columns using UDF's defined above.

```
# Calculating additional columns using the UDF's defined above.

expandedOrderStream = orderStream \
                      .withColumn("total_items", add_total_item_count(orderStream.items)) \
                      .withColumn('total_cost', add_total_order_cost(orderStream.items, orderStream.type)) \
                      .withColumn('is_order', add_is_order(orderStream.type)) \
                      .withColumn('is_return', add_is_return(orderStream.type))

```

8. Writing the information collected for every 1 minutes time interval to the console.

```
# Writing the summarized information to console at a processing interval of 1 minute.

extendedOrderQuery = expandedOrderStream \
                     .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()

```

9. The next step is to calculate the time based KPI's (Key Performance Indicators) in a window of 1 minute. To calculate the below KPI's I'm grouping the data based on 1 minute time interval on timestamp column. Also, watermark is set to 1minute so that the data coming late up to 1 minute is also processed.
    a. Orders Per Minute (OPM): It is calculated as the count of total invoice numbers in that batch.
    b. Total Sale Volume: It is defined as the sum of 'total_cost' column for that batch.
    c. Rate of Return: It is defined as the average of 'is_return' column for that batch.
    d. Average Transaction Size: it is calculated as the average 'total_cost' column for that batch.

```
# Calculating Time Based Key Performance Indicators

aggStreamByTime = expandedOrderStream \
                  .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'))

```

10. Following this, the next step is to write the time based KPI's output to a JSON file on HDFS.

```
# Writing Time Based KPI values to a json file.

queryByTime = aggStreamByTime \
            .select("window", "OPM", 'total_sale_volume', 'rate_of_return', 'average_transaction_size') \
            .writeStream \
            .outputMode('append') \
            .format('json') \
            .option("truncate", "false") \
            .option('path', '/output/timeBasedKPI') \
            .option("checkpointLocation", "/output/timeBasedKPI") \
            .trigger(processingTime="1 minute") \
            .start()
```

11. The next step is to calculate KPI's based on time and country. To calculate these, we're grouping the data based on 1 minute interval on timestamp column and then the country column. Again, watermark is set to 1minute so that the data coming late up to 1 minute is also processed.

```
aggStreamByTimeCountry = expandedOrderStream \
                        .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'))

```

12. Finally, we need to write the time and country based API's to JSON files as well and then await for the termination of above three queries.

```
queryByTimeCountry = aggStreamByTimeCountry \
                    .select("window", 'country', "OPM", 'total_sale_volume', 'rate_of_return') \
                    .writeStream \
                    .outputMode('append') \
                    .format('json') \
                    .option('truncate', 'false') \
                    .option('path', '/output/timeCountryBasedKPI') \
                    .option("checkpointLocation", "/output/timeCountryBasedKPI") \
                    .trigger(processingTime = '1 minute') \
                    .start()

# Awaiting termination of the above written queries.

extendedOrderQuery.awaitTermination()
queryByTime.awaitTermination()
queryByTimeCountry.awaittermination()

```

13. The next step is to run the code. The below command is to be executed on spark EMR cluster.

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