# Code Logic - Retail Data Analysis

```
#importing necessary libraries and python modules

import sys
import os
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from ast import literal_eval
```

Required python modules and libraries are imported here. Literal_eval will be used to convert string from items column into a proper python list of dictionary.

```

# get total cost. It will be arrived at by summing up the unit price and quantity of products.
def get_total_cost(items):
    items = literal_eval(items)
    total_cost = 0
    for item in items:
        total_cost += item["unit_price"] * item["quantity"]
    return total_cost

# get total items. It will be arrived by summing up the total quantity of the products.
def get_total_items(items):
    items = literal_eval(items)
    total_items = 0
    for item in items:
        total_items += item["quantity"]
    return total_items

# if that order is ORDER or RETURN. Incase the category is ORDER return 1 else i.e., category is RETURN, return 0 for order type.
def type_order(category):
    if category == "ORDER":
        return 1
    return 0

# if that order is ORDER or RETURN. Incase the category is ORDER return 0 else i.e., category is RETURN, return 1 for return type.
def type_return(category):
    if category == "RETURN":
        return 1
    return 0
```

Below are the details of the custom functions:

1. get_total_cost(items): This function takes item as an argument and calculate the total cost by summing up the unit price and quantity of products. The formulae to calculate get total cost is :

$$\sum(\text{quantity} * \text{unitprice})$$

2. get_total_items(items): This function takes item as an argument and used to retrieve the total items by summing up total quantity of the products. The formulae to calculate the total item is:

$$\sum(\text{quantity})$$

3. type_order(category): This function takes category as an argument, and it is used to map type of order. If that order is ORDER or RETURN. In case the category is ORDER return 1 else i.e., category is RETURN, return 0 for order type.
4. type_return(category): This function takes category as an argument, and it is used to map type of order. If that order is ORDER or RETURN. In case the category is ORDER return 0 else i.e., category is RETURN, return 1 for return type.

```
39
40      if len(sys.argv) != 4:
41          print("Usage: spark-submit spark-streaming.py <hostname> <port> <topic>")
42          exit(-1)
43
44      host = sys.argv[1]
45      port = sys.argv[2]
46      topic = sys.argv[3]
47
48      spark = SparkSession  \
49  ⟶      .builder  \
50  ⟶      .appName("RetailDataAnalysis")  \
51  ⟶      .getOrCreate()
52      spark.sparkContext.setLogLevel('ERROR')
53
54      bootstrap_server = host + ":" + port
55
56      lines = spark  \
57  ⟶      .readStream  \
58  ⟶      .format("kafka")  \
59  ⟶      .option("kafka.bootstrap.servers", bootstrap_server)  \
60  ⟶      .option("subscribe", topic)  \
61  ⟶      .load()
62
```

In the beginning, the host, port number and topic is received from the command line argument.
host: 18.211.252.152
port: 9092
topic: real-time-project

This is used to read the spark stream from kafka bootstrap server.

```
62
63      schema = StructType() \
64              .add("invoice_no", StringType()) \
65              .add("country", StringType()) \
66              .add("timestamp", TimestampType()) \
67              .add("type", StringType()) \
68              .add("items", StringType())
69
70      raw_data = lines.selectExpr("cast(value as string)").select(from_json("value", schema).alias("temp")).select("temp.*")
71
72      # create user-defined functions for each
73      total_cost = udf(lambda items: get_total_cost(items))
74      total_quantity = udf(lambda items: get_total_items(items))
75      is_order = udf(lambda types: type_order(types))
76      is_return = udf(lambda types: type_return(types))
77
78      new_df = raw_data
79      new_df = new_df.withColumn("total_cost", total_cost("items")) \
80              .withColumn("total_items", total_quantity("items")) \
81              .withColumn("is_order", is_order("type")) \
82              .withColumn("is_return", is_return("type"))
83
84
85      # create kafka dataframe with invoice number, country, timestamp, total cost, total items, is order and is return
86      kafkaDF = new_df.select(["invoice_no", "country", "timestamp", "total_cost", "total_items", "is_order", "is_return"])
87      kafkaDF = kafkaDF.withColumn("total_cost", when(kafkaDF.is_order == 1, kafkaDF.total_cost).otherwise(-kafkaDF.total_cost))
88
```

At first the schema is created. Schema consist of the following:

i. invoice_no – String type
ii. country – String type
iii. timestamp – Timestamp type
iv. type – String type
v. items – String type

raw_data = lines.selectExpr("cast(value as string)").select(from_json("value", schema).alias("temp")).select("temp.*")

The above line reads the data in SQL data frame format.

```
# create user-defined functions for each
total_cost = udf(lambda items: get_total_cost(items))
total_quantity = udf(lambda items: get_total_items(items))
is_order = udf(lambda types: type_order(types))
is_return = udf(lambda types: type_return(types))
```

user defined functions are created and it will return custom function outputs.

The next lines will create a new data frame with columns total_cost, total_quantity, is_order, is_return. Total cost will become negative if the order is 0 else the total cost remains as is.

```
88
89      # streaming raw data
90      query0 = kafkaDF.select(["invoice_no", "country", "timestamp", "total_cost", "total_items", "is_order", "is_return"])
91
92
93      # create time-based KPI with tumbling window of one minute
94      query1 = kafkaDF.select(["timestamp", "invoice_no", "total_cost", "is_order", "is_return"])
95      query1 = query1.withWatermark("timestamp", "1 minute").groupBy(window("timestamp", "1 minute")) \
96              .agg(round(sum("total_cost"), 2).alias("total_sales_volume"), count("invoice_no").alias("OPM"), \
97                  round(sum("is_return") / (sum("is_order") + sum("is_return")), 2).alias("rate_of_return"), \
98                  round(sum("total_cost") / count("invoice_no"), 2).alias("average_transaction_size"))
99
100
101     # create time-and-country based KPI with tumbling window of one minute
102     query2 = kafkaDF.select(["timestamp", "invoice_no", "country", "total_cost", "is_order", "is_return"])
103     query2 = query2.withWatermark("timestamp", "1 minute").groupBy(window("timestamp", "1 minute"), "country") \
104             .agg(round(sum("total_cost"), 2).alias("total_sales_volume"), count("invoice_no").alias("OPM"), \
105                 round(sum("is_return") / (sum("is_order") + sum("is_return")), 2).alias("rate_of_return"))
106
```

The next steps explains the batch SQL data frames with proper schema are created.

Query 0 gives us the output in console.
Query 1 to create time-based KPI with tumbling window of 1 minute.
Query 2 to create time-and-country based KPI with tumbling window of 1 minute.

To calculate the KPIs, it can be done by summing total costs because the values of total_cost are already in decent positive and negative values that were transformed earlier.

```
107
108     # write stream data to write the time-based KPIs into one minute window each
109     query0 = query0.writeStream \
110         .format("console") \
111         .outputMode("append") \
112         .option("truncate", "false") \
113         .trigger(processingTime="1 minute") \
114         .start()
115     #write stream data into json format
116     query1 = query1.writeStream \
117         .format("json") \
118         .outputMode("append") \
119         .option("truncate", "false") \
120         .option("path", "/user/ec2-user/real-time-project/warehouse/op1") \
121         .option("checkpointLocation", "hdfs:///user/ec2-user/real-time-project/warehouse/checkpoints1") \
122         .trigger(processingTime="1 minute") \
123         .start()
124
125     query2 = query2.writeStream \
126         .format("json") \
127         .outputMode("append") \
128         .option("truncate", "false") \
129         .option("path", "/user/ec2-user/real-time-project/warehouse/op2") \
130         .option("checkpointLocation", "hdfs:///user/ec2-user/real-time-project/warehouse/checkpoints2") \
131         .trigger(processingTime="1 minute") \
132         .start()
133
134     query0.awaitTermination()
135     query1.awaitTermination()
136     query2.awaitTermination()
137
```

Here the dataframes will be written. Query 0 will write in  console and query 1 and query 2 will print the JSON format.

Await termination wait for the stream to finish.