Code Logic - Retail Data Analysis

In this document, you will describe the code and the overall steps taken to solve the project.

Commands Used in the project:

1. Command to create a directory in HDFS to store the time based KPIs
   ```
   hadoop fs –mkdir time_kpi
   ```

2. Command to create a directory in HDFS to be used a checkpoint while calculating time based KPIs
   ```
   hadoop fs –mkdir time_kpi/checkpoint
   ```

3. Command to create a directory in HDFS to store the country based KPIs
   ```
   hadoop fs –mkdir country_kpi
   ```

4. Command to create a directory in HDFS to be used a checkpoint while calculating country based KPIs
   ```
   hadoop fs –mkdir country_kpi/checkpoint
   ```

5. Spark-submit command used
   ```
   spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5 spark-streaming.py >  Console-output
   ```

Overall Steps taken to solve the project:

Python code which processes the streaming data from Kafka producer has the following logic

- Step 1: Import Spark libraries
- Step 2: Create Spark session
- Step 3: Declare and implement UDF(helper functions) to calculate additional columns. Following UDFs were created
    - **get_total_cost:** Calculates total cost of the items in an order
    - **get_total_items:** Calculates total items in an order
    - **is_order:** Determines if the order is an actuan order
    - **is_return:** Determines if the order is a return
- Step 4: Declare schema to read the data from Kafka Producer
- Step 5: Read orders data from Kafka Producer using the schema
- Step 6: Call UDF functions to add following columns to the Spark dataframe
    - **total_cost**: Total cost of an order arrived at by summing up the cost of all products in that invoice
    - **total_items**: Total number of items present in an order
    - **is_order**: 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.

    - **is_return**: 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.
- Step 7: Write the input to console_output file generated for each one-minute window.
- Step 8: Calculate the following KPIs for each 1 minute window with a 10 minute watermark, using aggregation functions available in Spark SQL functions
    - Total volume of sales – Total sales made in a 1 minute window
    - OPM (orders per minute) – Total orders made in a 1 minute window
    - Rate of return – The rate of returns in a 1 minute window
    - Average transaction size – Average transaction size in terms of sales volume, total orders and total returns in a 1 minute window
- Step 9: Write KPIs calculated based on time window to time_kpi directory on HDFS
- Step 10: Calculate the following KPIs for each 1 minute window with a 10 minute watermark per country basis, using aggregation functions available in Spark SQL functions
    - Total volume of sales – Total sales made in a 1 minute window grouped by country
    - OPM (orders per minute) – Total orders made in a 1 minute window grouped by country
    - Rate of return – The rate of returns in a 1 minute window grouped by country
- Step 11: Write KPIs calculated per country basis to country_kpi directory on HDFS

Python Code:

Step 1: Import Spark libraries

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

Step 2: Create Spark session

```
# Create a Spark Session to process Streaming data
spark = SparkSession  \
        .builder  \
        .appName("StructuredSocketRead")  \
        .getOrCreate()
spark.sparkContext.setLogLevel('ERROR')
```

Step 3: Define UDFs to calculate additional columns

```
# 1. UDF to get total cost of an order
def get_total_cost(order_type, items):

        total_cost = 0
        for item in items:
                total_cost = total_cost + (item['unit_price'] * item['quantity'])
```

```
    if order_type == "ORDER":
        return total_cost
    else:
        return total_cost * (-1)

total_cost_udf = udf(get_total_cost, FloatType())

# 2. UDF to get total items in an order
def get_total_items(items):

    total_items = 0

    for item in iter(items):
        total_items = total_items + item['quantity']

    return total_items

total_items_udf = udf(get_total_items, IntegerType())

# 3. UDF to determine if the order is an actual order
def is_order(order_type):

    if order_type == "ORDER":
        return 1
    else:
        return 0

is_order_udf = udf(is_order, IntegerType())

# 4. UDF to determine if the order is a return
def is_return(order_type):

    if order_type == "RETURN":
        return 1
    else:
        return 0

is_return_udf = udf(is_return, IntegerType())
```

Step 4: Declare schema to read the data from Kafka Producer

```
# Schema to read the data fromt he Kafka Producer
schema = StructType() \
    .add("type", StringType()) \
    .add("country", StringType()) \
    .add("invoice_no", LongType()) \
    .add("timestamp", TimestampType()) \
    .add("items", ArrayType(StructType() \
              .add("SKU", StringType()) \
              .add("title", StringType()) \
              .add("unit_price", FloatType()) \
              .add("quantity", IntegerType())
              ))
```

Step 5: Read orders data from Kafka Producer using the schema

```
# Read data from Kafka Producer
orders = spark  \
          .readStream  \
          .format("kafka")  \
          .option("kafka.bootstrap.servers","18.211.252.152:9092")  \
          .option("subscribe","real-time-project")  \
          .load()

# Wrangle the input data into the right columns
orders = orders.selectExpr("cast(value as string)") \
          .select(from_json('value', schema).alias("value" )) \
          .select("value.*")
```

Step 6: Call UDFs to add additional columns to the Spark dataframe

```
# Calculate following columns to help with the data analysys
#         1. total_cost: Total cost of an order arrived at by summing up the cost of all products in
that invoice
#         2. total_items: Total number of items present in an order
#         3. is_order: 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.
#         4. is_return: 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.

orders = orders.withColumn("total_cost", total_cost_udf(orders.type, orders.items))
orders = orders.withColumn("total_items", total_items_udf(orders.items))
orders = orders.withColumn("is_order", is_order_udf(orders.type))
orders = orders.withColumn("is_return", is_return_udf(orders.type))
```

Step 7: Write the input to console_output file generated for each one-minute window

```
# Calculating additional columns and writing the summarised input table to the console
orders_console= orders  \
          .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()
```

Step 8: Calculate the time based KPIs for each 1 minute window

```
# Calculating following time-based KPIs with a watermark of 10 minutes and a tumbling window
of 1 minute
#       1. Total volume of sales
#       2. OPM (orders per minute)
#       3. Rate of return
#       4. Average transaction size
orders_time_based_kpi= orders \
   .withWatermark("timestamp","10 minutes") \
   .groupby(window("timestamp", "1 minute")) \
   .agg(count("invoice_no").alias("OPM"),
                sum("total_cost").alias("total_sale_volume"),
      sum("is_order").alias("total_orders"),
      sum("is_return").alias("total_returns")) \
   .select("window","OPM","total_sale_volume","total_orders","total_returns")

orders_time_based_kpi = orders_time_based_kpi.withColumn("rate_of_return",
(orders_time_based_kpi.total_returns /(orders_time_based_kpi.total_orders +
orders_time_based_kpi.total_returns)))
orders_time_based_kpi = orders_time_based_kpi.withColumn("average_transaction_size",
(orders_time_based_kpi.total_sale_volume /(orders_time_based_kpi.total_orders +
orders_time_based_kpi.total_returns)))
```

Step 9: Write KPIs calculated based on time window to time_kpi directory on HDFS

```
# Write time based KPI values to JSON files
time_based_kpi = orders_time_based_kpi \
        .select("window", "OPM", "total_sale_volume", "rate_of_return",
"average_transaction_size") \
        .writeStream \
   .format("json") \
   .outputMode("append") \
   .option("truncate", "false") \
   .option("path", "time_kpi/") \
   .option("checkpointLocation", "time_kpi/checkpoint/") \
   .trigger(processingTime="1 minutes") \
   .start()
```

Step 10: Calculate per country based KPIs for each 1 minute window with a 10 minute

```
# Calculating following country-based KPIs with a watermark of 10 minutes and a tumbling
window of 1 minute
#        1. Total volume of sales
#        2. OPM (orders per minute)
#        3. Rate of return
orders_country_based_kpi= orders \
   .withWatermark("timestamp","10 minutes") \
   .groupby(window("timestamp", "1 minute"), "country") \
   .agg(count("invoice_no").alias("OPM"),
                sum("total_cost").alias("total_sale_volume"),
      sum("is_order").alias("total_orders"),
      sum("is_return").alias("total_returns")) \
   .select("window", "country", "OPM","total_sale_volume","total_orders","total_returns")

orders_country_based_kpi = orders_country_based_kpi.withColumn("rate_of_return",
(orders_country_based_kpi.total_returns /(orders_country_based_kpi.total_orders +
orders_country_based_kpi.total_returns)))
```

Step 11: Write KPIs calculated per country basis to country_kpi directory on HDFS

```
# Write time based KPI values
country_based_kpi = orders_country_based_kpi \
        .select("window", "country", "OPM", "total_sale_volume", "rate_of_return") \
        .writeStream \
   .format("json") \
   .outputMode("append") \
   .option("truncate", "false") \
   .option("path", "country_kpi/") \
   .option("checkpointLocation", "country_kpi/checkpoint/") \
   .trigger(processingTime="1 minutes") \
   .start()

country_based_kpi.awaitTermination()
```