Code Logic

1. **Import the modules**

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

Start by importing the essential libraries from Spark's SQL, functions, and types modules. These imports are vital for setting up and managing the Spark session, along with efficiently handling structured data.

2. **Initialize Spark Session**

```
spark = SparkSession \
    .builder \
    .appName("RetailDataStreaming") \
    .getOrCreate()

spark.sparkContext.setLogLevel('ERROR')
```

Here, we initialize a Spark session with the application name "RetailDataStreaming." The Spark session acts as the primary gateway for loading data, creating DataFrames, and executing SQL queries. To reduce log output and focus on only critical messages, we set the log level to 'ERROR'.

3. **Read the input data from Kafka server**

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

Connect to a Kafka server to stream data from the "real-time-project" topic. The `startingOffsets` is set to "latest" to start from the most recent data. Setting `failOnDataLoss` to "false" allows the stream to continue even if some data loss happens.

4. Schema and Parsing

```
order_schema = StructType([
    StructField("invoice_no", LongType(), False),
    StructField("country", StringType(), False),
    StructField("timestamp", TimestampType(), False),
    StructField("type", StringType(), False),
    StructField("items", ArrayType(StructType([
        StructField("SKU", StringType(), False),
        StructField("title", StringType(), False),
        StructField("unit_price", DoubleType(), False),
        StructField("quantity", IntegerType(), False)
    ])), True),
])

# Parse the JSON data
orders_df = kafka_stream.select(from_json(col("value").cast("string"),
order_schema).alias("data")).select("data.*")
```

Define a schema for the incoming JSON data to ensure accurate parsing. This schema includes fields like invoice number, country, timestamp, type, and an array of product details in the "items" field. With this schema, the Kafka stream can be parsed into a structured DataFrame, making data transformations and analysis simpler.

5. UDF (user Defined Functions)

```python
def calculate_total_price(items, type):
    total_price = 0
    for item in items:
        unit_price = item["unit_price"] if item["unit_price"] is not None else
0
        quantity = item["quantity"] if item["quantity"] is not None else 0
        total_price += unit_price * quantity

    return -total_price if type == "RETURN" else total_price

def calculate_total_items(items):
    total_items = 0
    for item in items:
        quantity = item["quantity"] if item["quantity"] is not None else 0
        total_items += quantity
    return total_items

def is_order(type):
    return 1 if type == "ORDER" else 0

def is_return(type):
    return 1 if type == "RETURN" else 0

# Register the UDFs
calculate_total_price_udf = udf(calculate_total_price, DoubleType())
calculate_total_items_udf = udf(calculate_total_items, IntegerType())
is_order_udf = udf(is_order, IntegerType())
is_return_udf = udf(is_return, IntegerType())

```

Calculating Total Price:

- The `calculate_total_price` function accepts a list of items along with their type, which can be either "ORDER" or "RETURN."
- It starts with a total price of zero.
- For each item, it checks the unit price and quantity. If they aren't provided (i.e., are `None`), it assumes them to be zero.
- It then multiplies the unit price by the quantity to get the price for that item and adds it to the total.
- If the type is "RETURN," it returns the negative total price (indicating money refunded). Otherwise, it returns the total price as is.

Calculating Total Items:

- The function `calculate_total_items` counts how many items are in the list.
- It initializes a total item count to zero.

- For each item, it checks the quantity, using zero if it's not provided.
- It adds up all the quantities to get the total number of items.

Identifying Orders and Returns:

- The function `is_order` checks if the type is "ORDER." If so, it returns 1 (indicating it's an order); otherwise, it returns 0.
- Similarly, the function `is_return` checks if the type is "RETURN" and returns 1 if true; otherwise, it returns 0.

Registering User-Defined Functions (UDFs):

- The last few lines register these functions as User-Defined Functions (UDFs) in Spark, which means they can be used in Spark DataFrames.
- `calculate_total_price_udf` will return a decimal number (DoubleType) for the total price, while `calculate_total_items_udf` will return an integer (IntegerType) for the total count of items.
- The functions for identifying orders and returns are also registered in a similar manner, returning integers.

6. Creating New Columns and writing it to console:

```
# Add the new fields to ordersDF
enriched_orders_df = orders_df \
    .withColumn("total_cost", calculate_total_price_udf(col("items"), col("type"))) \
    .withColumn("total_items", calculate_total_items_udf(col("items"))) \
    .withColumn("is_order", is_order_udf(col("type"))) \
    .withColumn("is_return", is_return_udf(col("type")))

# Write summarized input values to the console
orders_to_console = enriched_orders_df\
    .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()
```

The code enhances the original order DataFrame, `orders_df`, by adding additional columns to form a new DataFrame, `enriched_orders_df`. It computes `total_cost` using a user-defined function (UDF) that accounts for the items and their type (either order or return), and `total_items` using another UDF that counts the items. The code also introduces `is_order` and `is_return` columns to indicate whether the transaction is an order or a return. Finally, it configures a streaming output to the console, displaying selected fields—`invoice_no`, `country`, `timestamp`, `total_cost`, `total_items`, `is_order`, and `is_return`. This output is set to append new rows every minute without truncating, enabling real-time monitoring of the enriched order data.

7. Creating Time based KPIs

```
agg_by_time_df = enriched_orders_df.withWatermark("timestamp", "1 minute") \
    .groupBy(window(col("timestamp"), "1 minute")) \
    .agg(
        count("*").alias("OPM"),
        count(when(col("is_order") == 1, True)).alias("orders"),
        sum(col("total_cost")).alias("total_sale_volume"),
        count(when(col("is_return") == 1, True)).alias("returns")
    ) \
    .select(
        "window",
        "OPM",
        "total_sale_volume",
        (col("returns") / (col("orders") +
col("returns"))).alias("rate_of_return"),
        (col("total_sale_volume") / (col("orders") +
col("returns"))).alias("average_transaction_size")
    )
```

Watermarking:

- The line `.withWatermark("timestamp", "1 minute")` introduces a watermark to the DataFrame, which helps manage late data. It indicates that any data arriving more than one minute after the timestamp can be ignored, ensuring the aggregation remains timely and efficient.

Grouping by Time Window:

- The `.groupBy(window(col("timestamp"), "1 minute"))` groups the data into 1-minute time intervals based on the `timestamp` column. This allows the calculations to be done for each minute separately.

Aggregation:

- The `.agg(...)` function calculates several metrics for each time window:
    - `count("*").alias("OPM")`: Counts the total number of transactions (orders and returns) in each 1-minute window, labeled as "OPM" (Orders Per Minute).
    - `count(when(col("is_order") == 1, True)).alias("orders")`: Counts the number of orders in the window, where `is_order` equals 1.

- `sum(col("total_cost")).alias("total_sale_volume")`: Sums up the total cost of all transactions in the window to get the total sales volume.
- `count(when(col("is_return") == 1, True)).alias("returns")`: Counts the number of returns in the window, where `is_return` equals 1.

Selecting and Calculating Additional Metrics:

- The `.select(...)` method specifies which columns to include in the resulting DataFrame:
  - `"window"`: Includes the time window for which the calculations were made.
  - `"OPM"`: The total number of transactions in that window.
  - `"total_sale_volume"`: The total sales volume for that window.
  - `(col("returns") / (col("orders") + col("returns"))).alias("rate_of_return")`: Calculates the rate of returns by dividing the number of returns by the total number of transactions (orders + returns).
  - `(col("total_sale_volume") / (col("orders") + col("returns"))).alias("average_transaction_size")`: Calculates the average transaction size by dividing the total sales volume by the total number of transactions.

8. Calculating Country and Time bases KPIs

```
agg_by_country_time_df = enriched_orders_df.withWatermark("timestamp", "1 minute") \
    .groupBy(window(col("timestamp"), "1 minute"), col("country")) \
    .agg(
        count("*").alias("OPM"),
        count(when(col("is_order") == 1, True)).alias("orders"),
        sum(col("total_cost")).alias("total_sale_volume"),
        count(when(col("is_return") == 1, True)).alias("returns")
    ) \
    .select(
        "window",
        "country",
        "OPM",
        "total_sale_volume",
        (col("returns") / (col("orders") + col("returns"))).alias("rate_of_return")
    )

```

Watermarking:

- The `.withWatermark("timestamp", "1 minute")` method is applied to manage late data. It specifies that any data arriving more than one minute after the timestamp will be ignored. This helps keep the streaming aggregation efficient and relevant.

Grouping by Time Window and Country:

- The `.groupBy(window(col("timestamp"), "1 minute"), col("country"))` groups the data into 1-minute intervals based on the `timestamp` and further breaks it down by `country`. This allows calculations to be done for each country within each time window, enabling geographic analysis of the data.

Aggregation:

- The `.agg(...)` function calculates several metrics for each combination of time window and country:
    - `count("*").alias("OPM")`: Counts the total number of transactions (both orders and returns) in each 1-minute window for each country, labeled as "OPM" (Orders Per Minute).
    - `count(when(col("is_order") == 1,`

`True)).alias("orders")`: Counts the number of orders in the window for each country, where `is_order` equals 1.

- `sum(col("total_cost")).alias("total_sale_volume")`: Sums up the total cost of all transactions in that window for each country to calculate the total sales volume.
- `count(when(col("is_return") == 1, True)).alias("returns")`: Counts the number of returns in the window for each country, where `is_return` equals 1.

Selecting Key Metrics:

- The `.select(...)` method specifies which columns to include in the resulting DataFrame:
  - `"window"`: Includes the time window for the calculations.
  - `"country"`: Identifies the country for each aggregated result.
  - `"OPM"`: The total number of transactions in that window for the country.
  - `"total_sale_volume"`: The total sales volume for that window in the country.
  - `(col("returns") / (col("orders") + col("returns"))).alias("rate_of_return")`: Calculates the rate of returns for each country by dividing the number of returns by the total number of transactions (orders + returns).

9. Write KPIs to HDFS

```
time_aggregates_to_json = agg_by_time_df.writeStream \
    .outputMode("append") \
    .format("json") \
    .option("path", "/retail-data-analysis/kpis/time-based") \
    .option("checkpointLocation", "/retail-data-analysis/kpis/time-
based/checkpoint") \
    .trigger(processingTime="1 minute") \
    .start()

# Write time and country-based KPIs to JSON files
country_time_aggregates_to_json = agg_by_country_time_df.writeStream \
    .outputMode("append") \
    .format("json") \
    .option("path", "/retail-data-analysis/kpis/time-and-country-based") \
    .option("checkpointLocation", "/retail-data-analysis/kpis/time-and-
country-based/checkpoint") \
    .trigger(processingTime="1 minute") \
    .start()
```

1. **Creating the Write Stream for Time-Based KPIs**:
    - `time_aggregates_to_json = agg_by_time_df.writeStream`: This initiates a streaming write operation on the DataFrame `agg_by_time_df`, which contains time-based KPIs.
2. **Setting the Output Mode**:
    - `.outputMode("append")`: This specifies that only new rows added to the DataFrame will be written out, rather than rewriting the entire dataset. This is suitable for continuously updated data.
3. **Choosing the Output Format**:
    - `.format("json")`: This indicates that the output should be in JSON format, making it easy to read and compatible with various data processing tools. 4. **Specifying the Output Path**:
    - `.option("path", "/retail-data-analysis/kpis/time-based")`: This sets the directory where the JSON files will be stored. Each new batch of data will be saved in this location. 5. **Setting Up Checkpointing**:
    - `.option("checkpointLocation", "/retail-data-analysis/kpis/time-based/checkpoint")`: This defines a checkpoint directory used by Spark to store the progress of

the streaming query. Checkpointing is crucial for fault tolerance, allowing the stream to recover from failures without losing data.

6. **Defining the Trigger Interval**:
    - `.trigger(processingTime="1 minute")`: This sets the frequency at which the stream will be processed, in this case, every minute. It ensures that new data is output regularly.
7. **Starting the Write Stream**:
    - `.start()`: This command initiates the write stream, allowing the specified KPIs to be continuously written to the defined path in JSON format.

Writing Time and Country-Based KPIs to JSON 8.

Creating the Write Stream for Country-Based KPIs:

- `country_time_aggregates_to_json = agg_by_country_time_df.writeStream`: This initiates a streaming write operation on the DataFrame `agg_by_country_time_df`, which contains time and country-based KPIs.

9. **Repeating the Setup**:
    - The subsequent lines mirror the setup for the time-based KPIs:
        - `.outputMode("append")`: Only new rows will be written.
        - `.format("json")`: Output in JSON format.
        - `.option("path", "/retail-data-analysis/kpis/time-and-country-based")`: Specifies a different directory for storing country-based KPI data.
        - `.option("checkpointLocation", "/retail-data-analysis/kpis/time-and-country-based/checkpoint")`: Sets a checkpoint directory for fault tolerance.
        - `.trigger(processingTime="1 minute")`: Processes the stream every minute.
        - `.start()`: Initiates the write stream for the country-based KPIs.

10. **Awaiting Termination of the Streaming Queries**

```
# Terminate the Spark streaming job
orders_to_console.awaitTermination()
time_aggregates_to_json.awaitTermination()
country_time_aggregates_to_json.awaitTermination()
```

1. **`orders_to_console.awaitTermination()`**:
    - This line instructs the Spark application to wait for the streaming query that outputs enriched orders to the console to finish. The `awaitTermination()` method blocks the current thread until the associated streaming query is terminated, either by an external action (such as a stop command) or due to an error. This ensures that the application continues running and processing data until there's a reason to stop.
2. **`time_aggregates_to_json.awaitTermination()`**:
    - Similar to the previous line, this line makes the application wait for the streaming query that writes time-based KPIs to JSON files to complete. The application will remain active and continue to process incoming data for this query until it is terminated.
3. **`country_time_aggregates_to_json.awaitTermination()`**:
    - This line functions the same way as the previous two, but for the streaming query that writes time and country-based KPIs to JSON files. It ensures that this query is also kept running, processing data continuously until it is stopped

11. **We can start the spark job by this command**

```
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
```

`export SPARK_KAFKA_VERSION=0.10`:

- This command sets an environment variable named `SPARK_KAFKA_VERSION` to

`0.10`. This variable indicates the version of the Kafka integration that Spark should use. It helps ensure compatibility with the specific Kafka version being utilized in your application, particularly when you are working with features that may be version-specific.

`spark-submit`:

- This is a command-line tool provided by Apache Spark that allows users to submit their applications to a Spark cluster. It manages the application's execution, including resource allocation, configuration settings, and more, making it an essential tool for running Spark jobs.

`--packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5`:

- This option specifies additional libraries that Spark should include when running the application. Here, the `spark-sql-kafka-0-10` package is specified, which provides the necessary support for Spark SQL to interact with Kafka. The details include:
  - `org.apache.spark`: The group ID for the Spark project.
  - `spark-sql-kafka-0-10_2.11`: The artifact ID for the Kafka integration, indicating that this version is compatible with Scala 2.11.
  - `2.4.5`: The version of the Kafka integration library being used. This ensures that the application uses the correct version that matches the Spark version.

`spark-streaming.py`:

- This is the Python script that contains the Spark application code. The script is expected to implement logic for processing data from Kafka, such as reading from a Kafka topic, performing transformations, and potentially writing results to another destination.