Code Logic - Retail Data Analysis

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

1. Introduction and Project Overview

This project involves real-time analysis of retail data using Spark Streaming and Kafka. The objective is to process incoming JSON data from Kafka, compute various key performance indicators (KPIs) based on time and country, and output summarized data to the console and JSON files.

2. Setup and Environment

- **Spark Setup**: Ensure Spark is installed and configured properly on your environment.
- **Kafka Setup**: Kafka must be installed and running, accessible via `18.211.252.152:9092`.
- **Dependencies**: Ensure you have necessary dependencies, including the Spark Kafka package (`org.apache.spark:spark-sql-kafka-0-10_2.12:3.2.0`).

3. Project Structure

- **Project Directory**: The main project file is `spark-streaming.py`, which handles data processing and streaming logic.

To execute the `spark-submit` command for your `spark-streaming.py` script with the necessary Kafka package dependency (`org.apache.spark:spark-sql-kafka-0-10_2.12:3.2.0`)

```
spark-submit --packages
org.apache.spark:spark-sql-kafka-0-10_2.12:3.2.0 spark-streaming.py
```

Code Explanation

```
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import Window
```

- **Purpose**: This section imports required libraries from PySpark, including `SparkSession` for creating a Spark application entry point, various functions (`*` imports all functions), `StructType` for defining schema, and `Window` for defining windows for aggregations.

```
# Create a SparkSession
spark =
SparkSession.builder.appName("RetailDataAnalysis").getOrCreate()

# Set log level to ERROR
spark.sparkContext.setLogLevel('ERROR')
```

- **Purpose**:
  - `SparkSession.builder.appName("RetailDataAnalysis").getOrCreate()` creates a SparkSession with the application name "RetailDataAnalysis".
  - `spark.sparkContext.setLogLevel('ERROR')` sets the log level of the SparkContext to 'ERROR' to minimize unnecessary log output.

```
# Utility Functions
def is_a_order(type):
    return 1 if type == 'ORDER' else 0

def is_a_return(type):
    return 1 if type == 'RETURN' else 0

def total_item_count(items):
    if items is not None:
        item_count = 0
```

```
        for item in items:
            item_count += item['quantity']
        return item_count

def total_cost(items, type):
    if items is not None:
        total_cost = 0
        for item in items:
            item_price = item['quantity'] * item['unit_price']
            total_cost += item_price
        return total_cost * -1 if type == 'RETURN' else total_cost
```

- **Purpose**: These are utility functions defined as Python functions to be used later as User Defined Functions (UDFs) in Spark DataFrame operations.
    - `is_a_order(type)` and `is_a_return(type)`: Determine if the record type is 'ORDER' or 'RETURN'.
    - `total_item_count(items)`: Calculate the total number of items in an order.
    - `total_cost(items, type)`: Calculate the total cost of an order (or return, if type is 'RETURN').

```
# Register UDFs
is_order = udf(is_a_order, IntegerType())
is_return = udf(is_a_return, IntegerType())
add_total_item_count = udf(total_item_count, IntegerType())
add_total_cost = udf(total_cost, FloatType())
```

- **Purpose**: These lines register the previously defined Python functions as UDFs (User Defined Functions) usable in Spark SQL DataFrame operations. Each function is associated with a data type (`IntegerType` or `FloatType`).

```
# Read input data from Kafka
raw_stream = spark.readStream.format("kafka") \
```

```
    .option("kafka.bootstrap.servers", "18.211.252.152:9092") \
    .option("subscribe", "real-time-project") \
    .option("startingOffsets", "latest") \
    .load()
```

- **Purpose**: This code sets up a Kafka source for streaming data into Spark. It reads from a Kafka topic named "real-time-project" with bootstrap servers located at `18.211.252.152:9092` and starts reading from the latest offset.

```
# Define Schema for incoming JSON data
JSON_Schema = StructType() \
    .add("invoice_no", LongType()) \
    .add("country", StringType()) \
    .add("timestamp", TimestampType()) \
    .add("type", StringType()) \
    .add("items", ArrayType(StructType([
        StructField("SKU", StringType()),
        StructField("title", StringType()),
        StructField("unit_price", FloatType()),
        StructField("quantity", IntegerType())
    ])))
```

- **Purpose**: Here, a schema (`JSON_Schema`) is defined for parsing incoming JSON data from Kafka. It specifies fields like `invoice_no`, `country`, `timestamp`, `type`, and an array of `items`, each with its own nested schema (`StructType`).

```
# Parse JSON data and create DataFrame
order_stream_data =
raw_stream.select(from_json(col("value").cast("string"),
JSON_Schema).alias("data")).select("data.*")
```

- **Purpose**: This code snippet parses the JSON data received from Kafka (`raw_stream`) according to the defined schema (`JSON_Schema`). It extracts the parsed data into a DataFrame (`order_stream_data`) for further processing.

```
# Calculate additional columns for the stream
order_stream_output = order_stream_data \
    .withColumn("total_cost", add_total_cost(order_stream_data.items,
order_stream_data.type)) \
    .withColumn("total_items",
add_total_item_count(order_stream_data.items)) \
    .withColumn("is_order", is_order(order_stream_data.type)) \
    .withColumn("is_return", is_return(order_stream_data.type))
```

- **Purpose**: Here, additional columns (`total_cost`, `total_items`, `is_order`, `is_return`) are computed and added to `order_stream_data` DataFrame using previously defined UDFs (`add_total_cost`, `add_total_item_count`, `is_order`, `is_return`).

```
# Write summarized input table to console
order_batch = order_stream_output \
    .select("invoice_no", "country", "timestamp", "total_cost",
"total_items", "is_order", "is_return") \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .option("truncate", "false") \
    .option("path", "/Console_output") \
    .option("checkpointLocation", "/Console_output_checkpoints") \
    .trigger(processingTime="1 minute") \
    .start()
```

- **Purpose**: This snippet defines a streaming query (`order_batch`) that writes the summarized input table (`order_stream_output`) to the console (`format("console")`) every minute (`trigger(processingTime="1 minute")`). It also specifies checkpoint locations for fault tolerance.

```
# Calculate time-based KPIs
agg_time = order_stream_output \
    .withWatermark("timestamp", "1 minute") \
    .groupBy(window("timestamp", "1 minute")) \
    .agg(
        sum("total_cost").alias("total_volume_of_sales"),
        avg("total_cost").alias("average_transaction_size"),
        count("invoice_no").alias("OPM"),
        avg("is_return").alias("rate_of_return")
    ) \
    .select("window.start", "window.end", "OPM",
"total_volume_of_sales", "average_transaction_size", "rate_of_return")
```

- **Purpose**: This section calculates time-based Key Performance Indicators (KPIs) (`agg_time`) from `order_stream_output`. It groups data into 1-minute windows (`window("timestamp", "1 minute")`), computes aggregate functions like `sum`, `avg`, and `count` over these windows, and selects relevant metrics.

```
# Calculate time and country-based KPIs
agg_time_country = order_stream_output \
    .withWatermark("timestamp", "1 minute") \
    .groupBy(window("timestamp", "1 minute"), "country") \
    .agg(
        sum("total_cost").alias("total_volume_of_sales"),
        count("invoice_no").alias("OPM"),
        avg("is_return").alias("rate_of_return")
    ) \
```

```
    .select("window.start", "window.end", "country", "OPM",
"total_volume_of_sales", "rate_of_return")
```

- **Purpose**: Similar to the previous snippet, this calculates time and country-based KPIs (`agg_time_country`) using `order_stream_output`. It groups data into 1-minute windows and by country, computes aggregate functions, and selects relevant metrics.

```
# Write time-based KPI values to JSON
ByTime = agg_time.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate", "false") \
    .option("path", "timeKPIvalue") \
    .option("checkpointLocation", "timeKPIvalue_checkpoints") \
    .trigger(processingTime="1 minute") \
    .start()
```

- **Purpose**: This snippet starts a streaming query (`ByTime`) that writes time-based KPI values (`agg_time`) in JSON format (`format("json")`) every minute to a specified path (`"timeKPIvalue"`). It also defines a checkpoint location for fault tolerance.

```
# Write time and country-based KPI values to JSON
ByTime_country = agg_time_country.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate", "false") \
    .option("path", "time_countryKPIvalue") \
    .option("checkpointLocation", "time_countryKPIvalue_checkpoints")
\
    .trigger(processingTime="1 minute") \
    .start()
```

- **Purpose**: Similar to the previous snippet, this starts another streaming query (`ByTime_country`) that writes time and country-based KPI values (`agg_time_country`) in JSON format (`format("json")`) every minute to a specified path (`"time_countryKPIvalue"`). It also defines a checkpoint location for fault tolerance.

```
# Wait for the streaming computations to terminate
order_batch.awaitTermination()
ByTime.awaitTermination()
ByTime_country.awaitTermination()
```

- **Purpose**: This section ensures that all streaming queries (`order_batch`, `ByTime`, and `ByTime_country`) continue to run and wait for them to terminate gracefully. These lines essentially block the main thread, keeping the Spark Streaming job running until it's manually stopped or encounters an error.

Summary

This `spark-streaming.py` script demonstrates a real-time data processing pipeline using Spark Streaming and Kafka. Here's a recap of the key functionalities:

1. **Initialization and Setup:**
   - `SparkSession` is created to set up the Spark application.
   - Log level is set to 'ERROR' to minimize unnecessary log output.
2. **Utility Functions:**
   - Custom Python functions are defined to calculate specific metrics (`total_item_count`, `total_cost`, `is_a_order`, `is_a_return`), which are then registered as UDFs for use in Spark SQL.
3. **Streaming Setup**:
   - Kafka is configured as the streaming source (`raw_stream`), reading from a specific topic (`"real-time-project"`).

    - A schema (`JSON_Schema`) is defined to parse incoming JSON data into structured format (`order_stream_data`).
4. **Data Processing:**
    - Additional columns (`total_cost`, `total_items`, `is_order`, `is_return`) are computed based on the parsed data (`order_stream_data`).
    - These transformations (`order_stream_output`) prepare the data for further aggregation and analysis.
5. **Streaming Queries:**
    - `order_batch` writes summarized input data to the console in real-time (`outputMode("append")`).
    - `agg_time` and `agg_time_country` calculate time-based and time-and-country-based KPIs respectively, grouping data into 1-minute windows (`groupBy(window("timestamp", "1 minute"))`).
6. **Output:**
    - KPIs (`agg_time` and `agg_time_country`) are written to JSON files (`"timeKPIvalue"` and `"time_countryKPIvalue"`) every minute.
7. **Execution:**
    - `awaitTermination()` method ensures the streaming queries run indefinitely until manually stopped, ensuring continuous processing of data.

Thank you