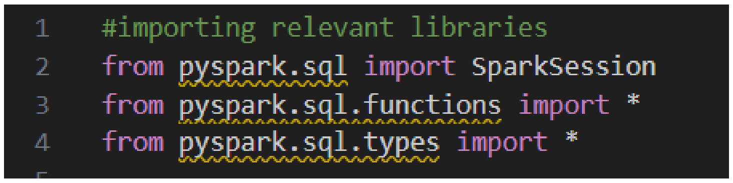
Code Logic - Retail Data Analysis

1. Library Imports



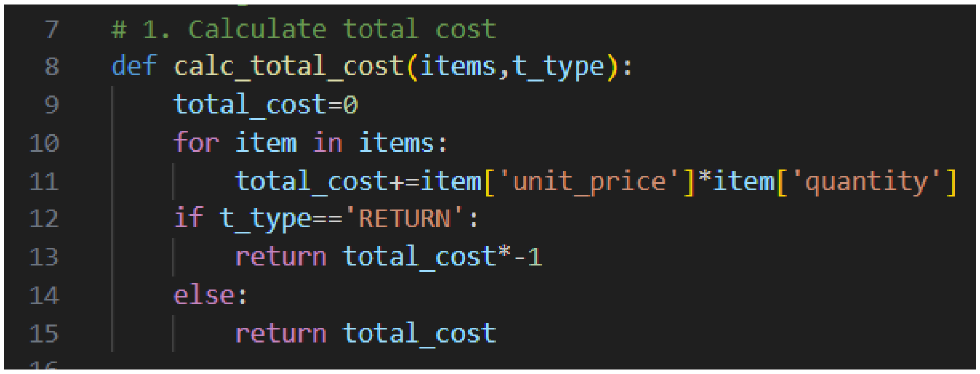
```python
#importing relevant libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
```

This section imports the necessary libraries for Spark SQL to work with Spark Streaming. It includes the SparkSession (the entry point for Spark SQL), SQL functions, and SQL data types.

2. User-Defined Functions (UDFs)

User-Defined Functions (UDFs) are custom functions created to perform specific data transformations that aren't available in Spark's built-in functions. In this code, four UDFs are defined to extract and calculate important business metrics from the transaction data.

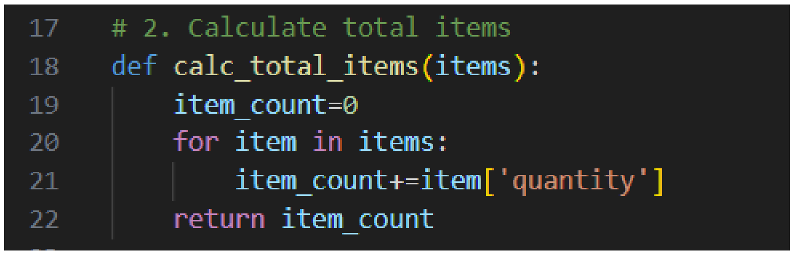
a. Total Cost



```python
# 1. Calculate total cost
def calc_total_cost(items,t_type):
    total_cost=0
    for item in items:
        total_cost+=item['unit_price']*item['quantity']
    if t_type=='RETURN':
        return total_cost*-1
    else:
        return total_cost
```

The **calc_total_cost** function calculates the total monetary value of a transaction by processing an array of items and considering the transaction type. It takes two parameters: an array of item objects (each containing unit price and quantity) and the transaction type (either 'ORDER' or 'RETURN'). The function iterates through each item in the transaction, multiplying each item's unit price by its quantity and accumulating these values into a running total. If the transaction is identified as a return, the function converts the total to a negative value, which helps in financial reporting where returns are represented as negative transactions. This function is crucial for financial analysis as it enables the accurate calculation of sales volume, revenue tracking, and financial reporting in the e-commerce application.

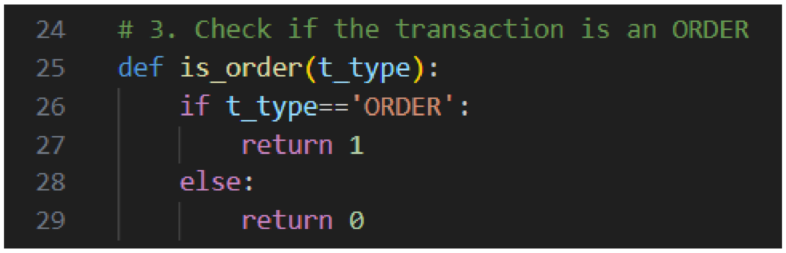
b. Total Items



```python
# 2. Calculate total items
def calc_total_items(items):
    item_count=0
    for item in items:
        item_count+=item['quantity']
    return item_count
```

The **calc_total_items** function determines the total quantity of individual items present in a transaction. It accepts a single parameter: an array of item objects, with each item containing at minimum a quantity field. The function works by iterating through every item in the provided array and summing up all the quantities into a total count. This aggregation is important because a single transaction in an e-commerce system typically contains multiple items, and each item can have different quantities. Understanding the total item count provides valuable insights into transaction size, order complexity, and customer purchasing behaviors. This metric can be used to analyze trends in basket size, identify high-volume transactions, and optimize inventory and fulfillment operations.

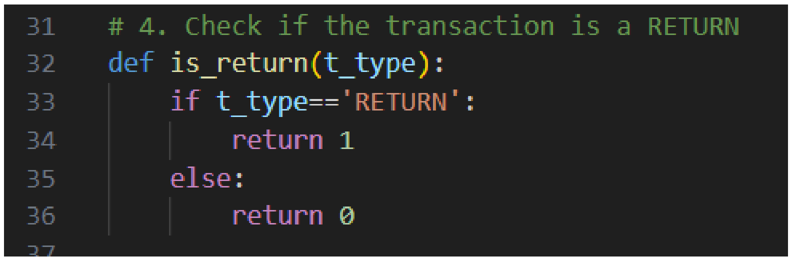
c. IS_ORDER



```python
# 3. Check if the transaction is an ORDER
def is_order(t_type):
    if t_type=='ORDER':
        return 1
    else:
        return 0
```

The **is_order** function creates a binary indicator that identifies whether a transaction is a purchase order. It takes a single parameter: the transaction type string. The function performs a simple conditional check to determine if the transaction type equals 'ORDER', returning 1 if true and 0 otherwise. This binary representation transforms a categorical variable into a numeric one that can be easily used in aggregations and calculations. By converting the transaction type into a binary indicator, the function enables straightforward filtering and counting of order transactions in the analytics pipeline. This is particularly useful when calculating metrics like orders per minute (OPM), analyzing sales patterns, or segmenting transactions by type. The numeric representation also allows for efficient storage and processing in Spark's data processing engine.

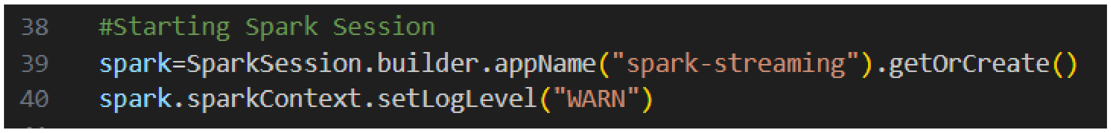
d. IS_RETURN



```python
# 4. Check if the transaction is a RETURN
def is_return(t_type):
    if t_type=='RETURN':
        return 1
    else:
        return 0
```

The **is_return** function generates a binary indicator specifically for identifying return transactions in the data stream. It accepts a transaction type string as input and performs a conditional check to determine if the transaction is classified as a 'RETURN'. When a return is detected, the function returns 1; otherwise, it returns 0. This numeric representation serves several important analytical purposes in the e-commerce context. Most significantly, it enables the calculation of return rates—a critical KPI for retail operations—by providing a value that can be averaged across transactions to determine the percentage of returns. The function also allows for easy filtering of return transactions when analyzing patterns and reasons for returns. By separating returns from orders in this manner, the application can track return metrics over time and by country, providing insights into product quality issues, customer satisfaction, and operational efficiency in the returns process.

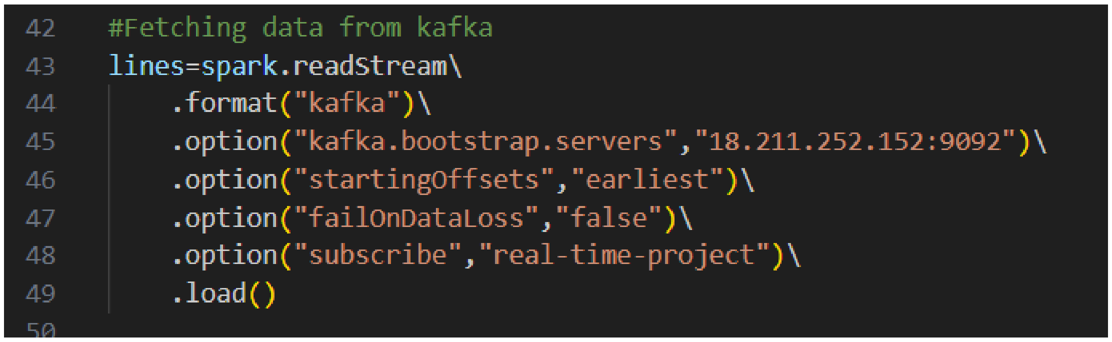
3. Spark Session Initialization



```python
#Starting Spark Session
spark=SparkSession.builder.appName("spark-streaming").getOrCreate()
spark.sparkContext.setLogLevel("WARN")
```

Creates a SparkSession named "spark-streaming" (or gets an existing one) and sets the log level to WARN to reduce verbosity.

4. Data Ingestion from Kafka

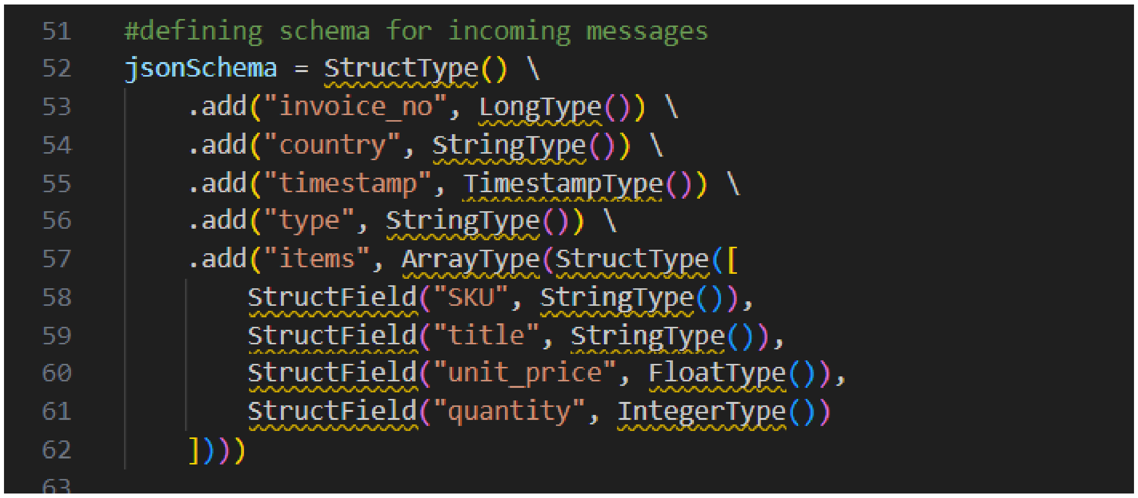


```python
#Fetching data from kafka
lines=spark.readStream\
    .format("kafka")\
    .option("kafka.bootstrap.servers","18.211.252.152:9092")\
    .option("startingOffsets","earliest")\
    .option("failOnDataLoss","false")\
    .option("subscribe","real-time-project")\
    .load()
```

Sets up a streaming connection to a Kafka topic named **"real-time-project"**. It configures:

- Kafka bootstrap servers' address: **18.211.252.152.9092**
- Reading from the earliest available offset
- Not failing if data is lost
- The topic to subscribe to

5. Schema Definition



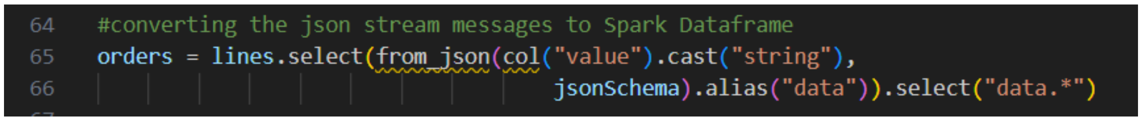
```python
#defining schema for incoming messages
jsonSchema = StructType() \
    .add("invoice_no", LongType()) \
    .add("country", StringType()) \
    .add("timestamp", TimestampType()) \
    .add("type", StringType()) \
    .add("items", ArrayType(StructType([
        StructField("SKU", StringType()),
        StructField("title", StringType()),
        StructField("unit_price", FloatType()),
        StructField("quantity", IntegerType())
    ])))
```

Defines the schema for the incoming JSON data, including:

- Basic transaction fields (invoice number, country, timestamp, type)
- An array of items with nested fields (SKU, title, unit price, quantity)

This schema ensures proper data typing and structure when parsing the JSON messages.

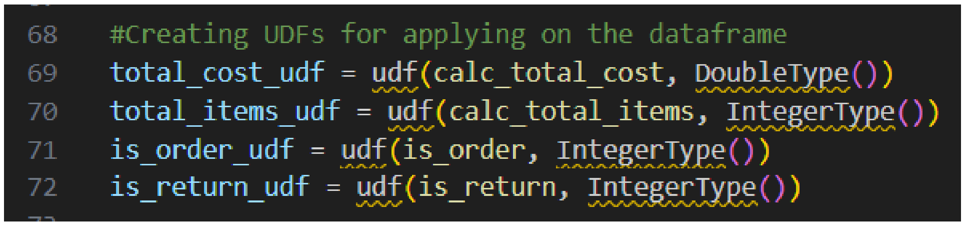
6. JSON parsing



```python
#converting the json stream messages to Spark Dataframe
orders = lines.select(from_json(col("value").cast("string"),
                                jsonSchema).alias("data")).select("data.*")
```

Parses the JSON data from Kafka messages using the defined schema and flattens the resulting structure.

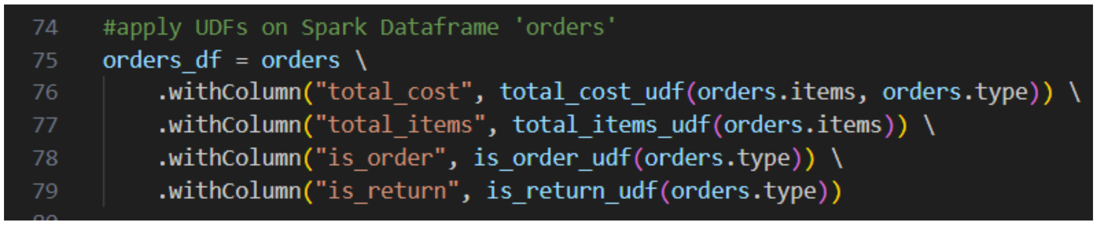
7. UDF Registration



```python
#Creating UDFs for applying on the dataframe
total_cost_udf = udf(calc_total_cost, DoubleType())
total_items_udf = udf(calc_total_items, IntegerType())
is_order_udf = udf(is_order, IntegerType())
is_return_udf = udf(is_return, IntegerType())
```

These Python functions are registered as Spark UDFs using the **udf()** function, which allows them to be applied to DataFrame columns efficiently at scale. Registration includes specifying the return data type

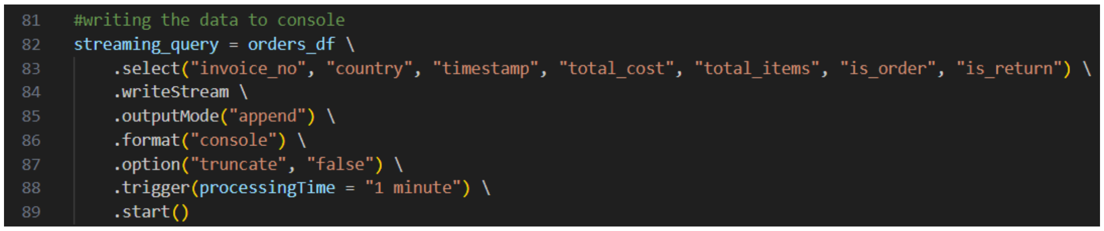
8. Application of UDF



```python
#apply UDFs on Spark Dataframe 'orders'
orders_df = orders \
    .withColumn("total_cost", total_cost_udf(orders.items, orders.type)) \
    .withColumn("total_items", total_items_udf(orders.items)) \
    .withColumn("is_order", is_order_udf(orders.type)) \
    .withColumn("is_return", is_return_udf(orders.type))
```

The UDFs are then applied to create new columns using the withColumn() method. This approach transforms raw transaction data into a format suitable for analysis by deriving business-relevant metrics from the complex nested structure of the original data.

9. Console Output Stream

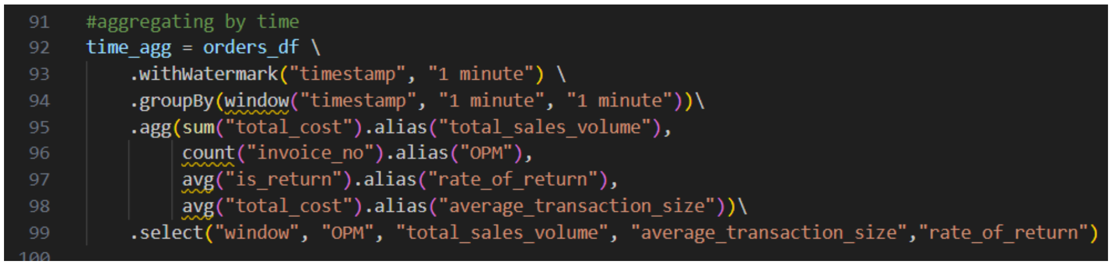


```python
#writing the data to console
streaming_query = orders_df \
    .select("invoice_no", "country", "timestamp", "total_cost", "total_items", "is_order", "is_return") \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .option("truncate", "false") \
    .trigger(processingTime = "1 minute") \
    .start()
```

Creates a streaming query to output transformed data to the console:

- Selects relevant columns
- Uses append output mode
- Formats output for console
- Prevents truncation of output
- Triggers processing every minute

10. Time-based Aggregation

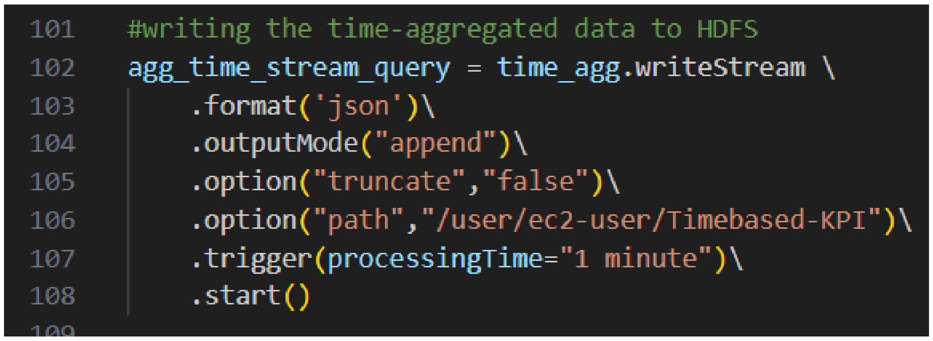


```python
#aggregating by time
time_agg = orders_df \
    .withWatermark("timestamp", "1 minute") \
    .groupBy(window("timestamp", "1 minute", "1 minute"))\
    .agg(sum("total_cost").alias("total_sales_volume"),
        count("invoice_no").alias("OPM"),
        avg("is_return").alias("rate_of_return"),
        avg("total_cost").alias("average_transaction_size"))\
    .select("window", "OPM", "total_sales_volume", "average_transaction_size","rate_of_return")
```

Performs time-window based aggregations:

- Sets a watermark for late data (1 minute)
- Groups by 1-minute tumbling windows
- Calculates KPIs:
    - Total sales volume
    - Orders per minute (OPM)
    - Return rate
    - Average transaction size

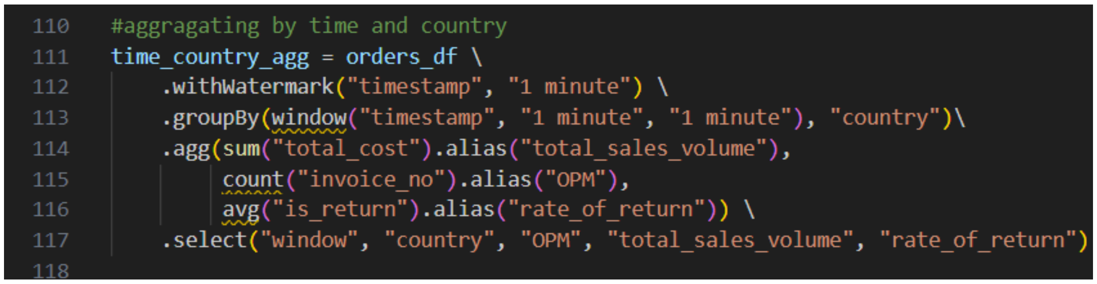
11. Writing Time Aggregates to HDFS



```
#writing the time-aggregated data to HDFS
agg_time_stream_query = time_agg.writeStream \
    .format('json')\
    .outputMode("append")\
    .option("truncate","false")\
    .option("path","/user/ec2-user/Timebased-KPI")\
    .trigger(processingTime="1 minute")\
    .start()
```

Sets up a streaming query to write time-aggregated data to HDFS in JSON format.

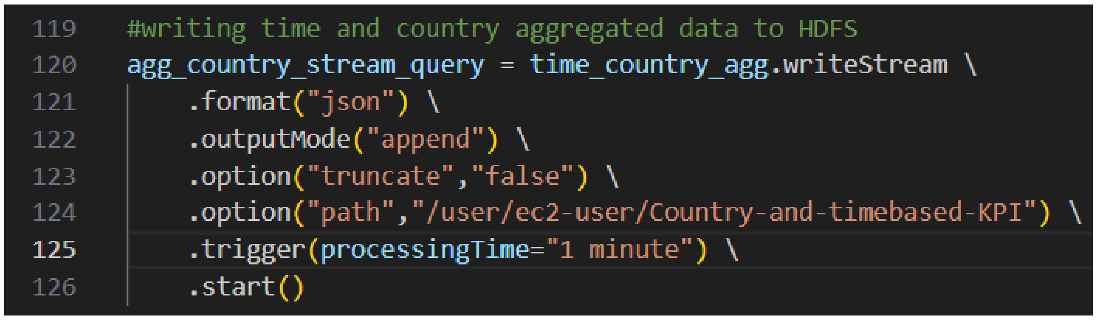
12. Time and Country Based Aggregation



```
#aggragating by time and country
time_country_agg = orders_df \
    .withWatermark("timestamp", "1 minute") \
    .groupBy(window("timestamp", "1 minute", "1 minute"), "country")\
    .agg(sum("total_cost").alias("total_sales_volume"),
        count("invoice_no").alias("OPM"),
        avg("is_return").alias("rate_of_return")) \
    .select("window", "country", "OPM", "total_sales_volume", "rate_of_return")
```

Similar to the time-based aggregation, but groups by both time window and country to provide country-specific metrics.

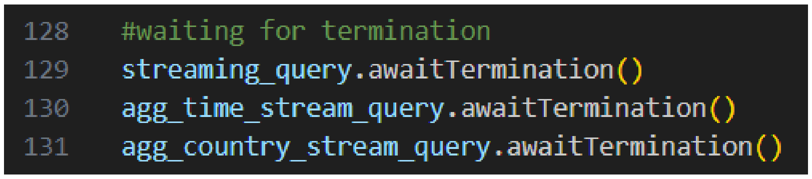
13. Writing Country and Time Aggregates to HDFS



```
#writing time and country aggregated data to HDFS
agg_country_stream_query = time_country_agg.writeStream \
    .format("json") \
    .outputMode("append") \
    .option("truncate","false") \
    .option("path","/user/ec2-user/Country-and-timebased-KPI") \
    .trigger(processingTime="1 minute") \
    .start()
```

Sets up another streaming query to write the country and time aggregated data to HDFS.

14. Awaiting Termination



```
#waiting for termination
streaming_query.awaitTermination()
agg_time_stream_query.awaitTermination()
agg_country_stream_query.awaitTermination()
```

Blocks the main thread until all the streaming queries terminate, which keeps the application running.