Lab6-Task1: Guided solution for Producing Data From Kafka To Kafka

breaking down the code and explaining each section:

# Section 1: Imports and Environment Setup

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql import types as T
```

Here, we're importing necessary modules and setting the correct environment paths for PySpark, Hadoop, and Python.

# Section 2: Define JSON Schema for Incoming Data

```
json_schema = T.StructType([
    T.StructField('application_name', T.StringType()),
    T.StructField('translated_review', T.StringType()),
    T.StructField('sentiment_rank', T.IntegerType()),
    T.StructField('sentiment_polarity', T.FloatType()),
    T.StructField('sentiment_subjectivity', T.FloatType())
])
```

Defines the structure (**json_schema**) for parsing the incoming JSON data. This structure dictates the expected fields in the JSON and their corresponding data types.

# Section 3: Initialize Spark Session

```
spark = SparkSession \
    .builder \
    .master("local") \
    .appName('ex6_calculate_reviews') \
    .config('spark.jars.packages', 'org.apache.spark:spark-sql-kafka-0-10_2.12:3.1.2') \
    .getOrCreate()
```

- Initializes a **SparkSession** which is the entry point to any Spark functionality.
- Sets the application name and the master for the SparkSession.
- Adds a specific jar for Kafka integration to the Spark job.

# Section 4: Consume Data from Kafka

```
stream_df = spark \
    .readStream \
    .format('kafka') \
    .option("kafka.bootstrap.servers", "course-kafka:9092") \
    .option("subscribe", "gps-user-review-source") \
    .option('startingOffsets', 'earliest') \
    .load() \
    .select(F.col('value').cast(T.StringType()))
```

- Reads streaming data from the Kafka topic named "gps-user-review-source" .
- The incoming byte data from Kafka is then converted to a string.

# Section 5: Parse the JSON Data

```
parsed_df = stream_df \
   .withColumn('parsed_json', F.from_json(F.col('value'), json_schema)) \
   .select(F.col('parsed_json.*'))
```

- Uses the **from_json** function to parse the incoming JSON-formatted string data.
- Extracts the parsed JSON data into a new DataFrame named **parsed_df**.

# Section 6: Load Static Data from Parquet and Cache

```
static_data_df = spark.read.parquet('s3a://spark/data/source/google_apps/')

static_data_df.cache()
```

- Reads static data stored in Parquet format.
- Caches (in-memory storage) the data for better performance in subsequent operations.

# Section 7: Aggregation, Transformation, and Join Operations

```
joined_df = parsed_df\
   .groupBy(F.col('application_name'))\
   .agg(F.sum(F.when(F.col('sentiment_rank') == 1, 1).otherwise(0)).alias('num_of_positive_sentiments'),
       F.sum (F.when (F.col('sentiment_rank') == 0, 1).otherwise(0)).alias('num_of_neutral_sentiments'),
       F.sum(F.when (F.col('sentiment_rank') == -1, 1).otherwise(0)).alias('num_of_negative_sentiments'),
       F.avg(F.col('sentiment_polarity')).alias('avg_sentiment_polarity'),
       F.avg(F.col('sentiment_subjectivity')).alias('avg_sentiment_subjectivity')\ (
  .join(static_data_df, ['application_name'])
```

Let's break down this part of the code:

A. Grouping by Application Name:

`.groupBy(F.col('application_name'))`

The data is grouped based on the application_name. This means that all reviews related to a specific application will be grouped together, and then operations within the .agg() function will be applied to each group.

B. Aggregation

- **Counting Positive Sentiments:**

`F.sum(F.when(F.col('sentiment_rank') == 1, 1).otherwise(0)).alias('num_of_positive_sentiments')`

Explanation: For each row within the group, if the sentiment_rank column is equal to 1 (indicating a positive sentiment), it contributes a count of 1. If not, it contributes a count of 0. The sum of these counts will give the total number of positive sentiments for the application.

- **Counting Neutral Sentiments**:

`F.sum(F.when(F.col('sentiment_rank') == 0, 1).otherwise(0)).alias('num_of_neutral_sentiments')`

Explanation: Similar to the previous step, but here, a sentiment rank of 0 indicates a neutral sentiment. The sum of these counts will give the total number of neutral sentiments for the application.

- **Counting Negative Sentiments**:

`F.sum(F.when(F.col('sentiment_rank') == -1, 1).otherwise(0)).alias('num_of_negative_sentiments')`

Explanation: A sentiment rank of -1 indicates a negative sentiment. The sum of these counts will give the total number of negative sentiments for the application.

- **Calculating Average Sentiment Polarity:**

`F.avg(F.col('sentiment_polarity')).alias('avg_sentiment_polarity')`

Explanation This computes the average of the sentiment_polarity values for all reviews related to a specific application.

- **Calculating Average Sentiment Subjectivity:**

`F.avg(F.col('sentiment_subjectivity')).alias('avg_sentiment_subjectivity')`

Explanation: Similarly, this computes the average of the sentiment_subjectivity values for all reviews related to a specific application.

C. **Joining with Static Data**:

.join(static_data_df, ['application_name'])

Explanation: Once the aggregation is done, the result is joined with another DataFrame static_data_df on the column application_name. This means for every application in the aggregated data (joined_df), the corresponding details from the static_data_df are added.

In essence, the code segment takes streaming review data (parsed_df), groups it by application, computes various sentiment metrics for each application, and then enriches this data by joining with static application data (static_data_df).

# Section 8: Fetching Field Names:

```
fields_list = joined_df.schema.fieldNames()
```

Explanation: The **fieldNames** method retrieves the list of column names (or fields) from the schema of the **joined_df** DataFrame. So **fields_list** is a Python list containing these column names.

# Section 9: Mapping Field Names to Columns:

```
fields_as_cols = list(map(lambda col_name: F.col(col_name), fields_list))
```

Explanation: This line maps each column name in **fields_list** to its respective column in the DataFrame using the **F.col** function. The **map** function is used for this purpose, and the result is a list of columns, which we store in **fields_as_cols**.

# Section 10: Converting Data to JSON Format:

```
json_df = joined_df \
    .withColumn('to_json_struct', F.struct(fields_as_cols)) \
    .select(F.to_json(F.col('to_json_struct')).alias('value'))
```

xplanation:

- First, we use the **withColumn** method to create a new column named 'to_json_struct'. This column is a struct type, combining all the columns in **fields_as_cols**.
- Then, we use the **F.to_json** function to convert the struct to a JSON string. We select only this JSON string and alias it as 'value'.

# Section 11: Writing the Stream to Kafka:

```
query = json_df \
    .writeStream \
    .format('kafka') \
    .option("kafka.bootstrap.servers", "course-kafka:9092") \
    .option("topic", "gps-with-reviews") \
    .option('checkpointLocation', 's3a://spark/checkpoints/ex6/review_calculation') \
    .outputMode('update') \
    .start()
```

Explanation:

- The **writeStream** method indicates we're setting up a streaming write.
- We specify the format as 'kafka' since we're writing to a Kafka topic.
- We provide the Kafka bootstrap server details and the target topic name.

- The checkpoint location (**checkpointLocation**) is specified to allow Spark to keep track of which records have been processed, ensuring fault tolerance and stream resumption capabilities.
- The **outputMode** is set to 'update', meaning only the rows that have changed are written to the output sink.
- Finally, **start()** initiates the streaming process.

# Section 12: Waiting for Stream Termination:

```
query.awaitTermination()
```

Explanation: This line ensures the Spark application keeps running and processing the data until an external action stops it or it encounters an error.

# Section 13: Removing Cached Data:

```
static_data_df.unpersist()
```

Explanation: The unpersist method removes the static_data_df DataFrame from cache. Since we called cache() on this DataFrame earlier, it's a good practice to free up memory once it's no longer needed by unpersisting.

# Section 13: Stopping the Spark Session:

```
spark.stop()
```

Explanation: This method stops the active SparkSession, ensuring all resources are freed and no further operations occur.

Summary:
after joining and aggregating the data, the code prepares the results in a JSON format and streams it to a Kafka topic. Once streaming is done, it releases the cached static data and stops the Spark session.

Full code solution

```
from pyspark.sql import SparkSession
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql import types as T

json_schema = T.StructType([
   T.StructField('application_name', T.StringType()),
   T.StructField('translated_review', T.StringType()),
   T.StructField('sentiment_rank', T.IntegerType()),
   T.StructField('sentiment_polarity', T.FloatType()),
   T.StructField('sentiment_subjectivity', T.FloatType())])

spark = SparkSession \
   .builder \
   .master("local") \
   .appName('ex6_calculate_reviews') \
   .config('spark.jars.packages', 'org.apache.spark:spark-sql-kafka-0-10_2.12:3.1.2') \
   .getOrCreate()

stream_df = spark \
   .readStream \
   .format('kafka') \
   .option("kafka.bootstrap.servers", "course-kafka:9092") \
   .option("subscribe", "gps-user-review-source") \
   .option('startingOffsets', 'earliest') \
   .load() \
   .select(F.col('value').cast(T.StringType()))

parsed_df = stream_df \
   .withColumn('parsed_json', F.from_json(F.col('value'), json_schema)) \
   .select(F.col('parsed_json.*'))

static_data_df = spark.read.parquet('s3a://spark/data/source/google_apps/')

static_data_df.cache()

joined_df = parsed_df \
   .groupBy(F.col('application_name')) \
   .agg(F.sum(F.when(F.col('sentiment_rank') == 1, 1).otherwise(0)).alias('num_of_positive_sentiments'),
      F.sum(F.when(F.col('sentiment_rank') == 0, 1).otherwise(0)).alias('num_of_neutral_sentiments'),
      F.sum(F.when(F.col('sentiment_rank') == -1, 1).otherwise(0)).alias('num_of_negative_sentiments'),
      F.avg(F.col('sentiment_polarity')).alias('avg_sentiment_polarity'),
      F.avg(F.col('sentiment_subjectivity')).alias('avg_sentiment_subjectivity')) \
   .join(static_data_df, ['application_name'])

fields_list = joined_df.schema.fieldNames()
fields_as_cols = list(map(lambda col_name: F.col(col_name), fields_list))

json_df = joined_df \
   .withColumn('to_json_struct', F.struct(fields_as_cols)) \
   .select(F.to_json(F.col('to_json_struct')).alias('value'))

query = json_df \
   .writeStream \
   .format('kafka') \
   .option("kafka.bootstrap.servers", "course-kafka:9092") \
   .option("topic", "gps-with-reviews") \
   .option('checkpointLocation', 's3a://spark/checkpoints/ex6/review_calculation') \
   .outputMode('update') \
   .start()

query.awaitTermination()

static_data_df.unpersist()

spark.stop()
```