Handling real-time Kafka data streams using PySpark

What is Kafka and PySpark ?

Kafka is a real-time messaging system that works on publisher-subscriber methodology. Kafka is a super-fast, fault-tolerant, low-latency, and high-throughput system built for real-world scenarios with out-of-the-box traffic and data transfer.

Pyspark is a python wrapper built over the real deal Apache Spark developed in Scala. Apache Spark is a distributed processing and analytics system providing multiple systems to handle huge amounts of data. The provided systems include Spark Core Engine, Spark SQL, Spark Streaming, MLlib, GraphX, and Spark R.

What to do with Kafka and PySpark?

We already know Kafka is a real-time messaging system that works as a queue as well as storage for transferring this huge amount of messages to different systems using either consumers or sinks like Cassandra. These real-time streams can also be used to extract different pieces of important information from the data in real-time to build a robust system, where any case of flaw or ambiguity can be handled and resolved quickly. These pieces of analytics are extracted using integration with PySpark, spark streaming to be more precise, as spark streaming enables us to work with real-time data streams.

Let’s get started - Walk

Create Spark Session

Connect spark streaming with Kafka topic to read data streams

Extract Topic information and apply suitable schema

Analyze the data using structured streaming SQL queries

****** COMMON FUNCTIONS******

Expose the calculated results back to Kafka

Note: Before moving forward make sure you have Kafka and PySpark installed in your system. Also do verify the working by making a simple producer publish data to a sample Kafka topic

1) Create Spark Session

```
from pyspark.sql import SparkSessionif __name__ == "__main__":
  spark = (
    SparkSession.builder.appName("Kafka Pyspark Streaming Learning")
    .master("local[*]")
    .getOrCreate()
  )
  spark.sparkContext.setLogLevel("ERROR")
```

2) ***Connect Spark Streaming with Kafka topic to read Data Streams*** since we have to read a real-time data stream from a Kafka topic its important to connect Spark Streaming to a Kafka Topic

```
KAFKA_TOPIC_NAME = "TOPIC_NAME"
KAFKA_BOOTSTRAP_SERVER = "localhost:9092"sampleDataframe = (
    spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVER)
    .option("subscribe", KAFKA_TOPIC_NAME)
    .option("startingOffsets", "latest")
    .load()
  )
```

Step 3: Extract Topic information and apply suitable schema

All the data in a Kafka topic is stored in the format of ***KEY: VALUE.***

Along with key and value, other metadata like timestamp is also stored in the Kafka topic. To extract the actual information from the Kafka topic we need to get the value from the stored JSON.

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

base_df = sampleDataframe.selectExpr("CAST(value as STRING)", "timestamp")
base_df.printSchema()
```

Here, the **printSchema** method is used to print the schema of the data imported. Also,
selectExpr("CAST(value as STRING)")
, means :
Select the value column from the data imported and cast the data to a string

Since the data published in a Kafka topic is in JSON format, a proper schema needs to be applied to it to convert it to a proper data frame. Apply a schema as per the JSON structure of the data published by the producer.

Request JSON Format :

```
{
   "col_a": "",
   "col_b": "",
   "col_c": "",
   "col_d": ""
}
```

Applying suitable schema,

```
sample_schema = (
      StructType()
      .add("col_a", StringType())
```

```
    .add("col_b", StringType())
    .add("col_c", StringType())
    .add("col_d", StringType())
)info_dataframe = base_df.select(
    from_json(col("value"), sample_schema).alias("sample"), "timestamp"
)
```

The last step to simplify the next steps of processing is to explode the multi-dimensional data to a 1D data frame by selecting suitable data.

```
info_df_fin = info_dataframe.select("sample.*", "timestamp")
```

Step 4: Analyze the data using structured streaming SQL queries

From spark 2.0, real-time data from Kafka topics can be analyzed efficiently using an ORM-like approach called the structured streaming component of spark. Structured streaming provides us functions to develop queries over the data frames. Before we move forward with building the queries using the functions provided, there are some pointers :

- *JOIN operation cannot be applied over real-time data streams*
- *Multiple clubbed aggregations (over common column) cannot be applied*
- *countDistinct function (for counting distinct values in a column cannot be applied), instead use approx_count_distinct()*

******COMMON FUNCTIONS******

1. **df*.agg(aggregation_1.alias, aggregation_2.alias())***
   << Used to apply aggregations like count & count distinct>>
2. ***df.groupBy()***
   << Used to group the data by a particular column >>
3. ***df.withColumn("COLUMN_NAME", "COLUMN_VALUE")***
   << Used to create a new column with a particular value >>

4. ***df.withColumnRenamed("OLD_COLUMN_NAME", "NEW_COLUMN_NAME)***
   << Used to rename a column >>

5. ***df.select("COLUMN_1", "COLUMN_2")***
   << Used to select a series of columns from the dataframe>>

6. ***df.where("CONDITION")***
   << Used to select data based on a condition >>

There are a lot more commands to carry out the most complex and sophisticated queries over the data frames. Example, ***lit(), struct(), cast(), alias(), from_json(), to_json()***

Step 5: Expose the calculated results back to Kafka

Writing data back from PySpark to other sources for future consumption over reports or analysis or dashboards is done using a simple source-sink methodology.

The source is the point from where the data is generated, here, PySpark
The sink is the point where the data is dumped.

PySpark provides multiple sinks for the purpose of writing the calculated analytics. These are :

- Files Sink, like JSONs, CSVs, etc
- Kafka Sink
- Console Sink (debugging sink, not persistent)
- ForEach Sink (debugging sink, not persistent), apply an additional function over each element or row of the result.

For the sake of this short tutorial, we will work with the smartest selection of sink for writing results for their analysis over a dashboard. and that is, writing data back to Kafka.

Before writing data to Kafka, points to remember :

- Since we already know that the data read from Kafka has ***"VALUE"*** attribute in it, we have to have a same-named column to write data back to Kafka Topic
- In case the result consists of multiple columns, condense them to a JSON, cast as a string, write to a value column
- Each column's data should be cast to String
- To keep a track of all the data written and other metadata related to the writing of data streams from PySpark to Kafka, a physical location (directory structure) has to be set as a Checkpoint Location.
- Set a trigger time to calculate the updated results every X seconds.
- PySpark provides multiple output modes while writing data to the sinks, which are complete, append, truncated, etc. While working with real-time streams make sure to keep the ***outputMode as COMPLETE***

```
DESTINATION_TOPIC = "DESTINATION_TOPIC"
CHECKPOINT_NAME = "A PHYSICAL DIRECTORY LOCATION"result_1 = (
      query_1.selectExpr(
         "CAST(col_a AS STRING)",
         "CAST(col_b_alias AS STRING)",
         "CAST(col_c_alias AS STRING)",
      )
      .withColumn("value", to_json(struct("*")).cast("string"),)
   )result2_1 = (
      result_1
      .select("value")
      .writeStream.trigger(processingTime="10 seconds")
```

```
    .outputMode("complete")
    .format("kafka")
    .option("topic", "DESTINATION_TOPIC")
    .option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVER)
    .option("checkpointLocation", CHECKPOINT_LOCATION)
    .start()
    .awaitTermination()
)
```

Now just execute the spark job using the below-mentioned command, keeping in mind the Kafka system along with the producers is up and running.

***Command**: spark-submit — packages org.apache.spark:spark-sql-kafka-0–10_2.12:<< PySpark Version >> << Streamer File Name >>*

spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.2 code/streamer.py

Note: Change the path of streamer.py as per yours.

And this is it, you have finally worked out a simple working query over your real-time data stream from Kafka and have written the results back to a Kafka topic for consumption over different channels or platforms.

Apps installed

Kafka , Zookeeper, Spark, install flask, kafka-python via pip.

1) Start zookeeper
2) Start Kafka
3) Spark-submit
4) Flaskapi
5) Producer
6) Consumer