
APACHE
Spark
TM

File Source - Spark Structured Streaming

Grow Data Skills

File1.json

```
{"userId": "user1", "timestamp": "2023-07-23T00:00:00Z"}
{"userId": "user2", "timestamp": "2023-07-23T00:01:00Z"}
```

File2.json

```
{"userId": "user1", "timestamp": "2023-07-23T00:02:00Z"}
{"userId": "user3", "timestamp": "2023-07-23T00:03:00Z"}
```

We can use Structured Streaming in Spark to process files as they arrive in a directory. We'll use the **readStream** method to **monitor** a directory for new files. It does not process the files that were present in the directory before the streaming computation started.

This is different from batch processing, where you might use the **read** method to process all files present in a directory at once.

Keep in mind that **"new files"** means files that are newly added to the directory. Files that are already present when the streaming computation starts, or files that are simply modified after the streaming computation has started, will not be processed. It is recommended to move or copy atomic files into the directory to avoid Spark picking up incomplete files.

We can configure **spark.sql.streaming.schemaInference** property to get the source schema during run time itself.

File Source - Spark Structured Streaming

Grow Data Skills

```python
from pyspark.sql import SparkSession

# Create a SparkSession with schema inference enabled
spark = SparkSession \
    .builder \
    .appName("StructuredStreamingUserLogCount") \
    .config("spark.sql.streaming.schemaInference", "true") \
    .getOrCreate()

# Read JSON files from directory
logs = spark \
    .readStream \
    .format("json") \
    .option("path", "/path/to/your/directory") \
    .load()

# Generate running count of logs for each user
userLogCounts = logs.groupBy("userId").count()

# Start running the query that prints the running counts to the console
query = userLogCounts \
    .writeStream \
    .format("json") \
    .outputMode("complete") \
    .option("path","output_dir") \
    .queryName("Logging Count") \
    .start()

query.awaitTermination()
```

Triggers - Spark Structured Streaming

Grow Data Skills

In Spark Structured Streaming, a trigger determines when the system should check for new data and process it. By default, it's set to process the data as soon as the system has completed processing the last batch of data, but you can set it to a fixed time interval or to only process one batch of data and then stop.

Here are three types of triggers:

- **Default Trigger (Processing Time):** If no trigger setting is explicitly specified, the system will check for new data as soon as it finishes processing the last batch. This is called "micro-batch" processing. If the system is idle (i.e., there is no new data to process), it will just wait until new data arrives.

- **Fixed Interval Micro-Batches**: The system will check for new data at a fixed interval, regardless of whether the system has finished processing the last batch of data.

- **One-Time Micro-Batch:** The system will process one batch of data and then stop. This is useful for testing and debugging.

Triggers - Spark Structured Streaming

Grow Data Skills

```
# Write the result to the console, and set a trigger
query = counts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .trigger(processingTime='5 seconds') \
    .start()
```

```
# Write the result to the console, and set a one-time trigger
query = counts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .trigger(once=True) \
    .start()
```

In a Spark cluster, an operation like aggregation is distributed across multiple nodes, with each node processing a part of the data. But how does Spark ensure that the previous state is considered in an operation like aggregation, given that the data is scattered across different nodes?

In the case of operations like **reduce** or **groupByKey**, Spark shuffles the data so that all values associated with a single key end up on the same node. This shuffling of data is an expensive operation in terms of network I/O, but it's necessary to perform operations that require knowledge of the entire dataset.

For example, in case of aggregating counts for a word stream, Spark would distribute the task of counting words in each batch across different nodes. The nodes would each maintain a local count for the words. Spark would then shuffle the data so that the final count for each word from all the nodes is aggregated on a single node.

To make this work, you generally need to ensure that you have partitioned your data in a way that allows it to be evenly distributed across the nodes and that there is sufficient memory to hold the shuffled data. If the state grows too large to fit in memory, you may need to increase the memory allocated to Spark or adjust your application to reduce its memory usage.

Checkpointing - Fault Tolerance

Grow Data Skills

Checkpointing in Spark Structured Streaming is a mechanism that periodically persists metadata about the streaming query to a fault-tolerant storage system, like HDFS, Amazon S3, etc. This is crucial in case of failure recovery, i.e., to ensure that the streaming processing can continue where it left off, providing resilience in the face of failures.

The checkpoint data includes:

- **Metadata** - Configuration, StreamingQuery id, etc.
- **Offset** - The point till where the data has been read and processed.
- **State data** - In case of stateful operations, like window or mapGroupsWithState.

To configure checkpointing, you can specify the checkpoint directory path when you start a streaming query. Here's how you can do it:

```
query = counts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .option("checkpointLocation", "/path/to/checkpoint/directory") \
    .start()
```

Exactly once in Structured Streaming

Grow Data Skills

The **exactly-once** processing semantics in Spark Structured Streaming means that each record will affect the final results exactly once, even in the case of failures. This is a crucial property for many applications, to ensure accurate results.

To achieve this, Spark Structured Streaming uses the following mechanisms:

- **Fault-tolerance through Checkpointing**: As mentioned above, Spark periodically saves the progress information to the checkpoint directory. If a query is restarted after a failure, it will start processing from the saved offset in the checkpoint directory, thus no data will be missed.

- **Source semantics**: The source must either be replayable or acknowledgeable. Replayable sources can replay data if needed, while acknowledgeable sources acknowledge data after it is processed, so it won't deliver the same data twice.

- **Sink Idempotency**: The sink, where the output is written, should be idempotent for exactly-once semantics, i.e., rerunning the same results will not change the output data. **For example**, overwriting a file in the file sink or upserting a row in the database. To achieve exactly-once in Kafka type of sinks is complicated.

Stateless Transformations

Stateless Transformations

Stateless transformations are those where each batch is processed independently and the processing of a batch does not depend on the data in other batches. The processing engine does not need to remember any data across batches for these transformations.

Examples of stateless transformations include:

Select: Extracting a subset of columns from the data.

```
df = streaming_df.select("name", "age")
```

Filter: Filtering data based on some condition.

```
df = streaming_df.filter(streaming_df.age > 21)
```

Stateful Transformations

Grow Data Skills

Stateful Transformations

Stateful transformations are those where the processing of a batch might depend on the data in other batches, meaning the processing engine needs to remember data across batches.

Examples of stateful transformations include:

- **Aggregation: Like calculating the count, sum, average, etc. Aggregations on streaming data are a series of consecutive windowed aggregations. The result of an aggregation operation changes as new data is added over time.**

```
from pyspark.sql.functions import count
df = streaming_df.groupBy("name").count()
```

  - **Global Aggregations:** When you're performing an aggregation without specifying a window or watermark, Spark will need to update the result for every new batch of data. This effectively means considering all previous data. However, in Structured Streaming, this would require an "output mode" of "complete" or "update", and the aggregation is subject to available resources (memory and computation time).

  - **Windowed Aggregations:** When you're using windows, Spark will only consider the data within the current window. The window could be sliding or tumbling based on your requirements.

Windowing

Grow Data Skills

Windowing is a mechanism to divide the data into time windows so that you can apply transformations on those windows. This allows you to perform operations on data that fell into a specific time frame.

In Spark Structured Streaming, windows are defined by two values, the **window size** and **slide duration**. Window size defines the length of the window, and slide duration determines how frequently new windowed data should be computed.

Here's an example of using window operations in Spark Structured Streaming:

```
from pyspark.sql.functions import window

    # Calculate the average sensor reading every 1 minute, based on event time
    df = df.groupBy(
        window(df.event_time, "1 minute")
    ).avg("reading")
```

In this example, window is a function that automatically groups data into windows of 1 minute, based on the event time. Then, for each window, we calculate the average sensor reading.

One thing to remember is that in order to use **event-time** based windowing, you need to have a **column** in your DataFrame that represents the event time, which can be parsed from the data itself.

Windowing

Grow Data Skills

For example, if we have a 1-minute window that starts at 10:00:00 and ends at 10:01:00, then all records with an event time that falls within that time frame (i.e., >=10:00:00 and <10:01:00) will be included in that window.

To illustrate, let's assume you have the following streaming data:

time	value
10:00:05	1
10:00:15	2
10:00:45	3
10:01:05	4
10:01:25	5
10:01:55	6

If you define a 1-minute window starting at 10:00:00, it will contain the first three records (with times 10:00:05, 10:00:15, and 10:00:45), and a window starting at 10:01:00 will contain the last three records (with times 10:01:05, 10:01:25, and 10:01:55).

Sliding Windowing

Grow Data Skills

In Spark Structured Streaming, the window slide duration determines how often the window operation is performed, i.e., how frequently a new set of data is processed.

If you specify a 1-minute window with a 1-minute slide (which is the default), then a new window will be generated every minute. This means that every minute, the system will look back over the previous minute of data and perform the window operation on that data.

Here's an example:

```
from pyspark.sql.functions import window

# Calculate the sum of "value" for each 1-minute window, sliding every 1 minute
df = df.groupBy(
    window(df.time, "1 minute", "1 minute")
).sum("value")
```

In this case, a new window starts every minute, so you will get separate windows for the data from 10:00 to 10:01, from 10:01 to 10:02, from 10:02 to 10:03, and so on.

Sliding Windowing

However, you can also specify a slide duration that's different from the window duration. For example:

from pyspark.sql.functions import window

```
# Calculate the sum of "value" for each 2-minute window, sliding every 1 minute
df = df.groupBy(
    window(df.time, "2 minutes", "1 minute")
).sum("value")
```

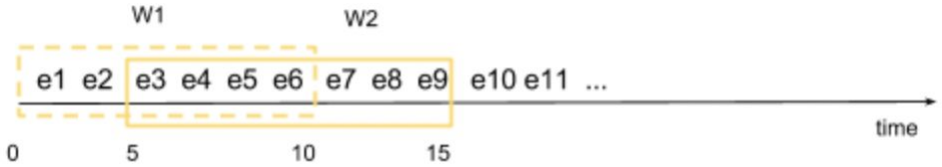
In this case, a new window starts every minute, but each window covers 2 minutes of data. So you will get overlapping windows: one for the data from 10:00 to 10:02, another for the data from 10:01 to 10:03, and so on.

Note that when you specify a slide duration, the system will perform the window operation at the frequency specified by the slide duration, not for each incoming record. If no slide duration is specified, it defaults to the window duration, which means a new window is generated for every duration of time specified by the window size.

Tumbling window vs Sliding window

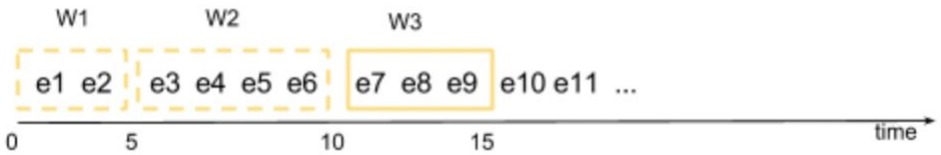
Grow Data Skills

- **Sliding Windows:** Sliding windows, on the other hand, can overlap. The "slide interval" of a sliding window determines how frequently a new window is created. If the slide interval is less than the window duration, you end up with overlapping windows. For instance, if you have a window duration of 1 minute and a slide interval of 30 seconds, you would get windows like 10:00-10:01, 10:00:30-10:01:30, 10:01-10:02, 10:01:30-10:02:30, and so on. Each event can belong to multiple windows, depending on when it occurs.



- **Tumbling Windows**: In tumbling windows, also known as hopping or fixed windows, each record in the stream belongs to exactly one window. These windows do not overlap and are mutually exclusive. This means if you have a tumbling window of 1 minute, you would get windows like 10:00-10:01, 10:01-10:02, 10:02-10:03, and so on. Each event belongs to exactly one of these windows.

  The concept of a tumbling window is represented by setting the **window duration equal to the slide duration**. That's because tumbling windows are a series of fixed, non-overlapping time intervals.



When and Why we should use Windowing?

Grow Data Skills

There are several situations where you might want to use windowing:

- **Aggregating Data over Time**: When you want to aggregate data over certain time intervals, windowing can be very useful. For example, you might want to calculate the total number of transactions per hour, or find the maximum temperature per day from a stream of weather data.

- **Detecting Trends**: If you're interested in detecting trends or patterns over time, you would use windowing. This can help you observe how your data changes over time. For instance, you might want to track how many times a particular error occurs in your application logs every 15 minutes.

- **Handling Late Data:** In real-world streaming applications, it's not uncommon for data to arrive late due to network issues or other delays. Windowing, especially when combined with watermarking (a feature that allows you to specify how late the data can be), can help handle late data and still produce accurate results.

- **Resource Efficiency**: By focusing computation on a specific window of data, you can reduce the amount of memory and computation resources needed compared to processing the entire stream.

Let’s understand Windowed Aggregations with example

a.) Input


```
{
"user_id": "user1",
"event": "click",
"eventtimestamp": "2023-07-30T10:00:00Z"
},
{
"user_id": "user1",
"event": "click",
"eventtimestamp": "2023-07-30T10:00:20Z"
},
{
"user_id": "user2",
"event": "click",
"eventtimestamp": "2023-07-30T10:01:00Z"
},
{
"user_id": "user1",
"event": "click",
"eventtimestamp": "2023-07-30T10:01:20Z"
},
{
"user_id": "user2",
"event": "click",
"eventtimestamp": "2023-07-30T10:02:00Z"
}
```

b.) Implementation

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, window
from pyspark.sql.types import StructType, StringType, TimestampType

# Define the schema of the data
schema = StructType() \
    .add("user_id", StringType()) \
    .add("event", StringType()) \
    .add("eventtimestamp", TimestampType())

# Initialize SparkSession
spark = SparkSession.builder.appName("WindowedAggregation").getOrCreate()

# Read the stream data into a DataFrame
df = spark.readStream.format("socket")\
    .option("host", "localhost")\
    .option("port", 9999)\
    .load()

# Parse the data
df = df.selectExpr("CAST(value AS STRING)")\
    .select(from_json(col("value"), schema).alias("data"))\
    .select(
        col("data.user_id"),
        col("data.event"),
        col("data.eventtimestamp")
    )

# Perform the windowed aggregation
result = df.groupBy(
    window(df.eventtimestamp, "1 minute"),
    df.user_id
).count()

# Start the query and write the result to the console
query = result.writeStream\
    .outputMode("complete")\
    .format("console")\
    .start()

query.awaitTermination()
```

c.) Output

```
-------------------------------------------
Batch: 0
-------------------------------------------
+------------------------------------+-------+-----+
|                              window |user_id|count|
+------------------------------------+-------+-----+
|[2023-07-30 10:00 to 2023-07-30 10:01| user1 |    2|
+------------------------------------+-------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------------------------------------+-------+-----+
|                              window |user_id|count|
+------------------------------------+-------+-----+
|[2023-07-30 10:00 to 2023-07-30 10:01| user1 |    2|
+------------------------------------+-------+-----+
|[2023-07-30 10:01 to 2023-07-30 10:02| user1 |    1|
+------------------------------------+-------+-----+
|[2023-07-30 10:01 to 2023-07-30 10:02| user2 |    1|
+------------------------------------+-------+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
+------------------------------------+-------+-----+
|                              window |user_id|count|
+------------------------------------+-------+-----+
|[2023-07-30 10:00 to 2023-07-30 10:01| user1 |    2|
+------------------------------------+-------+-----+
|[2023-07-30 10:01 to 2023-07-30 10:02| user1 |    1|
+------------------------------------+-------+-----+
|[2023-07-30 10:01 to 2023-07-30 10:02| user2 |    1|
+------------------------------------+-------+-----+
|[2023-07-30 10:02 to 2023-07-30 10:03| user2 |    1|
+------------------------------------+-------+-----+
```

d.) Explanation

Grow Data Skills

- **Batch 0:** The first two events from "user1" occur within the same one-minute window, between "2023-07-30T10:00:00Z" and "2023-07-30T10:01:00Z". Therefore, they're grouped together in this batch and Spark calculates the count of events (which is 2) for this window and user.

- **Batch 1:** The third event from "user2" at "2023-07-30T10:01:00Z" starts the next one-minute window. It's grouped together with the fourth event from "user1" at "2023-07-30T10:01:20Z", which falls within the same one-minute window (from "2023-07-30T10:01:00Z" to "2023-07-30T10:02:00Z").

- **Batch 2:** The fifth event from "user2" at "2023-07-30T10:02:00Z" starts the next one-minute window (from "2023-07-30T10:02:00Z" to "2023-07-30T10:03:00Z") and is the only event in that window.

Stateful Transformations

- **Arbitrary stateful processing**: **mapGroupsWithState** and **flatMapGroupsWithState** allow you to define your own stateful processing logic. This could involve maintaining a complex state, updating it based on new data, and defining a timeout condition to remove old state.

```
def update_func(key, values, state):
    # define your stateful processing logic here
    pass

df = streaming_df.groupByKey(key).mapGroupsWithState(update_func, TimeOutType)
```

Streaming with Infinite State

Grow Data Skills

```python
from pyspark.sql import SparkSession
from pyspark.sql.streaming import GroupStateTimeout
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StringType, LongType, DoubleType

# Define the schema of the JSON data
schema = StructType() \
    .add("userId", StringType()) \
    .add("transactionId", StringType()) \
    .add("transactionTime", LongType()) \
    .add("amount", DoubleType())

# Initialize SparkSession
spark = SparkSession.builder.appName("UserSession").getOrCreate()

# Read the stream data into a DataFrame
df = spark.readStream.format("kafka")\
    .option("kafka.bootstrap.servers", "localhost:9092")\
    .option("subscribe", "transactions")\
    .load()

# Parse the JSON data
df = df.selectExpr("CAST(value AS STRING)")\
    .select(from_json(col("value"), schema).alias("data"))\
    .select(
        col("data.userId"),
        col("data.transactionId"),
        from_unixtime(col("data.transactionTime")).alias("transactionTime"),
        col("data.amount"),
    )

# Define a function to update the state
def update_func(key, values, state):
    # If the state exists
    if state.exists:
        new_amount = sum([x.amount for x in values])
        # Update the state with the total amount
        state.update(state.get() + new_amount)
    else:
        # Initialize the state with the current amount
        state.update(sum([x.amount for x in values]))
    return (key, state.get())

# Group by userId and apply the update function with no timeout session
result = df.groupBy("userId").mapGroupsWithState(update_func, GroupStateTimeout.NoTimeout)

# Start the query and write the result to the console
query = result.writeStream\
    .outputMode("update")\
    .format("console")\
    .start()

query.awaitTermination()
```

From where this State variable is coming?

Grow Data Skills

The **state** variable is provided by Spark's structured streaming engine and represents the current state for a given key (userId in the provided examples).

In a **mapGroupsWithState** operation, Spark automatically maintains **state** for **each unique key** over batches of data. It does this by using a state store where it saves and retrieves state information for each key.

In the **update_func** method, Spark provides the current state object for the key being processed. This state object has several methods that allow you to:

- Check if a state exists (state.exists).
- Get the current state (state.get).
- Update the state with a new value (state.update).
- Remove the state (state.remove).

Remember, the state is managed across **micro-batches**, so the updated state from one micro-batch is made available to the next micro-batch for the same key. This is what enables **mapGroupsWithState** to maintain and **update** information across an **infinite** stream of data.

In the examples provided, the state for each **userId** is the **sum** of **amount** spent by that user. The state object is used to retrieve and update this sum as new transaction data for each user arrives in the stream.

Streaming with Fixed Timeout State

Grow Data Skills

```python
from pyspark.sql import SparkSession
from pyspark.sql.streaming import GroupStateTimeout
from pyspark.sql.functions import col, from_json, unix_timestamp
from pyspark.sql.types import StructType, StringType, LongType, DoubleType

# Define the schema of the JSON data
schema = StructType() \
    .add("userId", StringType()) \
    .add("transactionId", StringType()) \
    .add("transactionTime", StringType()) \
    .add("amount", DoubleType())

# Initialize SparkSession
spark = SparkSession.builder.appName("UserSession").getOrCreate()

# Read the stream data into a DataFrame
df = spark.readStream.format("kafka")\
    .option("kafka.bootstrap.servers", "localhost:9092")\
    .option("subscribe", "transactions")\
    .load()

# Parse the JSON data
df = df.selectExpr("CAST(value AS STRING)")\
    .select(from_json(col("value"), schema).alias("data"))\
    .select(
        col("data.userId"),
        col("data.transactionId"),
        unix_timestamp(col("data.transactionTime"), "yyyy-MM-dd HH:mm:ss").cast("timestamp").alias("transactionTime"),
        col("data.amount"),
    )
```

```python
# Define a function to update the state
def update_func(key, values, state):
    # If the state exists
    if state.exists:
        new_amount = sum([x.amount for x in values])
        # Update the state with the total amount
        state.update(state.get() + new_amount)
    else:
        # Initialize the state with the current amount
        state.update(sum([x.amount for x in values]))

    # Set the event time timeout to be 1 hour from the latest transactionTime
    timeout_timestamp = max([x.transactionTime for x in values]).add(hours=1)
    state.timeoutUntil(timeout_timestamp)

    return (key, state.get())

# Group by userId and apply the update function with a session timeout of 1 hour event-time based
result = df.groupBy("userId").mapGroupsWithState(update_func, GroupStateTimeout.EventTimeTimeout)

# Start the query and write the result to the console
query = result.writeStream\
    .outputMode("update")\
    .format("console")\
    .start()

query.awaitTermination()
```

How groupBy and mapGroupsWithState are different?

Grow Data Skills

In the context of Spark Structured Streaming, the **groupBy** operation followed by aggregations like **count()** or **sum()** does involve maintaining some state across batches. This is necessary for providing updated results that take into account data from both the current and previous batches.

These operations **don't allow** the developer to **manually define or manage** the state that's kept between batches, like **mapGroupsWithState** or **flatMapGroupsWithState** do. These latter functions give you the ability to define an **arbitrary** state, update it based on new data, and also define a timeout condition to remove old state.

In contrast, with groupBy followed by count() or sum(), Spark automatically manages the state, which in this case is the count or sum of values, and you as a developer don't have the control to define the state or update it based on your own logic.

Watermarking

Grow Data Skills

- Watermarking in Spark Structured Streaming is a technique that allows the system to **automatically track** the current event time in the data and attempt to **clean up old state** accordingly. It can be used in operations such as window operations, joins, etc. that require maintaining some state for processing.

- The key idea here is to define a "**watermark**" delay. This delay specifies how long the engine should wait for late data to arrive before it can consider a given point in time (i.e., event time) as fully processed. When this specified delay has passed, the engine assumes that no more old data will arrive and thus can perform cleanup and garbage collection of old state and data.

Watermarking with example
a.) Input

Grow Data Skills

Let's use simple clickstream data with user IDs and timestamps. Here, we're assuming we have data arriving in 3 batches, and we'll apply a watermark of 5 minutes.

- **Batch 0 (arrives at "2023-08-01T10:05:00Z")**

```
[
    {"user_id": "user1", "clicktimestamp": "2023-08-01T10:00:00Z"},
    {"user_id": "user2", "clicktimestamp": "2023-08-01T10:00:10Z"},
    {"user_id": "user1", "clicktimestamp": "2023-08-01T10:00:20Z"}
]
```

- **Batch 1 (arrives at "2023-08-01T10:10:00Z"):**

```
[
    {"user_id": "user2", "clicktimestamp": "2023-08-01T10:05:00Z"},
    {"user_id": "user2", "clicktimestamp": "2023-08-01T10:05:10Z"},
    {"user_id": "user1", "clicktimestamp": "2023-08-01T10:05:20Z"}
]
```

- **Batch 2 (arrives at "2023-08-01T10:15:00Z"):**

```
[
    {"user_id": "user1", "clicktimestamp": "2023-08-01T10:00:30Z"},
    {"user_id": "user2", "clicktimestamp": "2023-08-01T10:10:00Z"},
    {"user_id": "user1", "clicktimestamp": "2023-08-01T10:10:10Z"}
]
```

b.) Implementation

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, window
from pyspark.sql.types import StructType, StructField, StringType, TimestampType

# Start a Spark Session
spark = SparkSession.builder \
    .appName("Watermarking example") \
    .getOrCreate()

# Define the schema for our data
schema = StructType([
    StructField("user_id", StringType()),
    StructField("clicktimestamp", TimestampType())
])

# Define the source of the stream
stream_df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "clickstream") \
    .load()

# Parse the JSON data
parsed_df = stream_df \
    .select(from_json(col("value").cast("string"), schema).alias("parsed")) \
    .select("parsed.*")

# Perform the windowed aggregation
windowed_df = parsed_df \
    .groupBy(
        window(col("clicktimestamp"), "5 minutes"),
        col("user_id")
    ) \
    .count() \
    .withWatermark("window", "5 minutes")  # Apply the watermark

# Define the sink
query = windowed_df \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Wait for the streaming query to finish
query.awaitTermination()
```

c.) Output

Grow Data Skills

```
Batch 0
+------------------------------------------+-------+-----+
|                                    window|user_id|count|
+------------------------------------------+-------+-----+
|[2023-08-01 10:00:00, 2023-08-01 10:05:00]| user1 |    2|
|[2023-08-01 10:00:00, 2023-08-01 10:05:00]| user2 |    1|
+------------------------------------------+-------+-----+

Batch 1
+------------------------------------------+-------+-----+
|                                    window|user_id|count|
+------------------------------------------+-------+-----+
|[2023-08-01 10:00:00, 2023-08-01 10:05:00]| user1 |    2|
|[2023-08-01 10:00:00, 2023-08-01 10:05:00]| user2 |    1|
|[2023-08-01 10:05:00, 2023-08-01 10:10:00]| user1 |    1|
|[2023-08-01 10:05:00, 2023-08-01 10:10:00]| user2 |    2|
+------------------------------------------+-------+-----+


Batch 2
+------------------------------------------+-------+-----+
|                                    window|user_id|count|
+------------------------------------------+-------+-----+
|[2023-08-01 10:05:00, 2023-08-01 10:10:00]| user1 |    1|
|[2023-08-01 10:05:00, 2023-08-01 10:10:00]| user2 |    2|
|[2023-08-01 10:10:00, 2023-08-01 10:15:00]| user1 |    1|
|[2023-08-01 10:10:00, 2023-08-01 10:15:00]| user2 |    1|
+------------------------------------------+-------+-----+

After processing Batch - 2, data of window [2023-08-01 10:00:00, 2023-08-01 10:05:00]
will be removed because older start which is before watermark value will be deleted
```

d.) Explanation

Batch 0 Processing:

- The maximum **clicktimestamp** in this batch is "2023-08-01T10:00:20Z".
- Therefore, the watermark is "2023-08-01T10:00:20Z" - 5 minutes = "2023-08-01T09:55:20Z".
- All events are included because none of them are later than the watermark.

Batch 1 Processing:

- The maximum **clicktimestamp** in this batch is "2023-08-01T10:05:20Z".
- Therefore, the watermark is updated to "2023-08-01T10:05:20Z" - 5 minutes = "2023-08-01T10:00:20Z".
- All events are included because none of them are later than the updated watermark.

Batch 2 Processing:

- The maximum **clicktimestamp** in this batch is "2023-08-01T10:10:10Z".
- Therefore, the watermark is updated to "2023-08-01T10:10:10Z" - 5 minutes = "2023-08-01T10:05:10Z".
- The first event in this batch has a clicktimestamp of "2023-08-01T10:00:30Z", which is later than the updated watermark. Therefore, this event is considered late and is ignored in the aggregation.