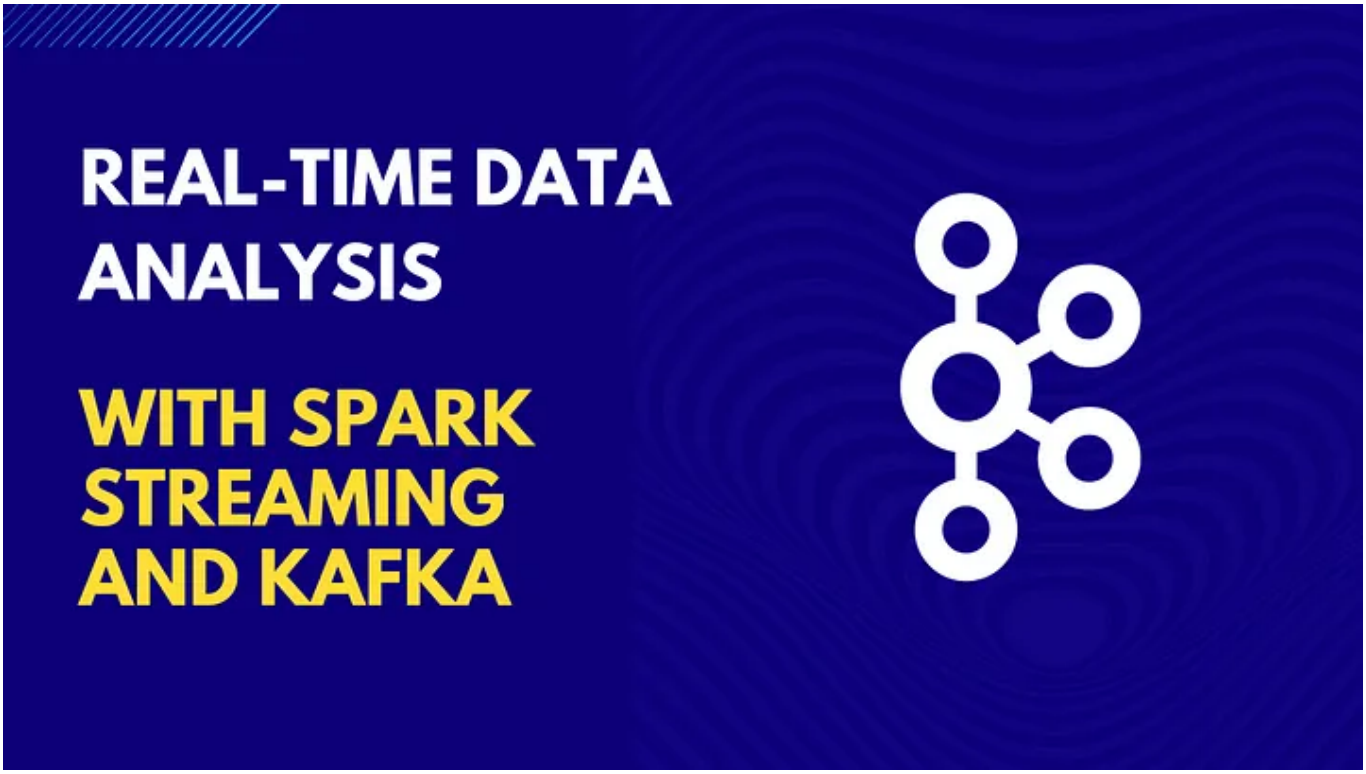



# Real-time AVRO Data Analysis with Spark Streaming and Confluent Kafka in Python

Mrugank Ray · Follow

12 min read · Mar 25



## Overview:

The goal of this article is to learn how to use Spark Streaming to process real-time AVRO data that we will consume from Confluent Kafka with Python. We'll store the schema of Kafka messages in the schema registry.

In my previous article, We covered how to create AVRO producers for Confluent Kafka with python.

## Source Code:

You can find the source code for this project on GitHub at: https://github.com/mrugankray/Spark_Streaming_with_kafka

## Setup:

Clone Big Data Cluster repo. This will set up an environment that runs spark with confluent kafka. Please ensure that Docker is installed on your machine before proceeding.

Run the following command to initialize a docker cluster

```
sudo docker-compose -f kafka-docker-compose.yaml up
```

## Dependency:

To execute the Spark consumer, it is necessary to have certain dependencies installed on your system. The primary dependencies include Pyspark, Confluent-Kafka, and Fastavro package. In addition to these requirements, Some Jar files are also essential for using Spark with Kafka.

But don't worry, as the Hadoop Cluster we previously setup has everything in place. There is no requirement for additional installation on your system.

## Source:

We're going to use Wikimedia's event streams as the source. To learn more visit https://stream.wikimedia.org/?doc#/streams/get_v2_stream_recentchange.

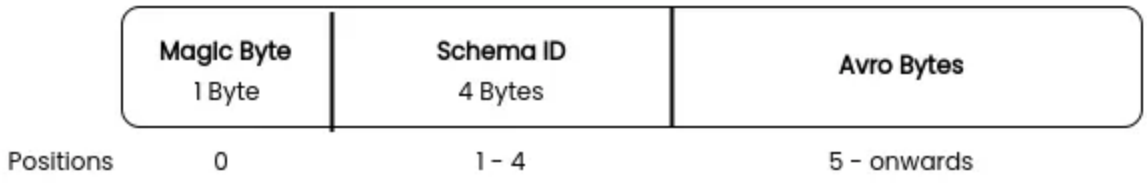
Our next step involves developing a producer to send the Wikimedia events to Confluent Kafka in AVRO format, I have extensively discussed this topic in the article.

## AVRO Data Storage:

Confluent Kafka stores data in a binary format. Along with the actual data, the Confluent Kafka team decided to store a **magic byte** and **schema id**. A magic byte is used to identify the **protocol format**.

Currently, the protocol format is always set to 0. The next four bytes represent the schema ID.

The Confluent Avro format looks like this:




Confluent Kafka Avro Storage Format

You can find more details on magic bytes in this article.

Here is a good article on Confluent Avro format.

## Analysing Data:

Let's talk about how to consume messages with PySpark and analyse the consumed messages using PySpark's data transformation and analysis capabilities.

### Run PySpark Code

To run PySpark code for consuming messages, you need to log in to **namenode** container by using the following command:

```
sudo docker exec -it namenode /bin/bash
```

Once you are inside the namenode container, navigate to the directory where you have saved your PySpark code and execute it using the following command

```
spark-submit <file_name.py>
```

### Wikimedia Dataframe Schema

Let's consume wikimedia data from Kafka with PySpark. We will see what metadata Kafka sends along with the primary data.

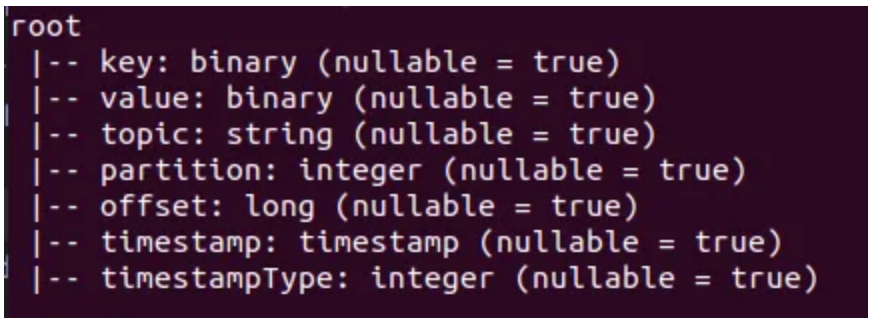
```python
# pyspark imports
from pyspark.sql import SparkSession

kafka_url = "kafka-broker:29092"
schema_registry_url = "http://schema-registry:8083"
kafka_producer_topic = "wikimedia"
schema_registry_subject = f"{kafka_producer_topic}-value"

# Create a SparkSession (the config bit is only for Windows!)
spark = SparkSession.builder.appName("wikimedia_consumer").getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

def spark_consumer():
    wikimedia_df = spark \
    .readStream\
    .format("kafka")\
    .option("kafka.bootstrap.servers", kafka_url)\
    .option("subscribe", kafka_producer_topic)\
    .option("startingOffsets", "earliest")\
    .load()

    wikimedia_df.printSchema()

spark_consumer()
```

print_wikimedia_df_schema.py hosted with ❤ by GitHub view raw

Let's understand the code:

- Line 1: Let's import Spark Session
- Line 4 to 7: — Kafka broker and schema registry URLs are defined. Then, the topic name is specified from where we will consume data. In addition to this, a schema registry subject is defined which typically follows the format of **topic_name-value** for value schemas. If storing schemas for keys is required then it can be named **topic_name-key**.
- Line 10: We create a spark session object and set the app name to **wikimedia_consumer.**

- Line 12: We set the log level to **ERROR** to reduce unnecessary logs and improve performance.
- Line 14 to 23: We create a function called **spark_consumer** to consume messages from Kafka. We set the **startingOffsets** to the earliest available offset in Kafka for this topic. You can also set the value to **latest** to consume messages from the latest offset in Kafka. To learn more about Spark's offset options, you can refer to the official documentation




wikimedia dataframe schema

You can see in the above screenshot that we get key, topic, partition, offset and timestamp metadata along with the value. We are interested in consuming **value** and performing further transformations or analysis on it using PySpark. The value column contains all the information that our producer script is pulling from the Wikimedia EventStreams API.

### Get Magic Byte and Schema ID

Let's see how to get the magic byte, schema ID and Wikimedia's data from value.

```python
# pyspark imports
import pyspark.sql.functions as func
from pyspark.sql import SparkSession

kafka_url = "kafka-broker:29092"
schema_registry_url = "http://schema-registry:8083"
kafka_producer_topic = "wikimedia"
schema_registry_subject = f"{kafka_producer_topic}-value"

# Create a SparkSession (the config bit is only for Windows!)
spark = SparkSession.builder.appName("wikimedia_consumer").getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

def spark_consumer():
    wikimedia_df = spark \
    .readStream\
    .format("kafka")\
    .option("kafka.bootstrap.servers", kafka_url)\
    .option("subscribe", kafka_producer_topic)\
    .option("startingOffsets", "earliest")\
    .load()

    # wikimedia_df.printSchema()

    #  get magic byte value
    wikimedia_df = wikimedia_df.withColumn("magicByte", func.expr("substring(value, 1, 1)"))

    #  get schema id from value
    wikimedia_df = wikimedia_df.withColumn("valueSchemaId", func.expr("substring(value, 2, 4)"))

    # remove first 5 bytes from value
    wikimedia_df = wikimedia_df.withColumn("fixedValue", func.expr("substring(value, 6, length(v

    # creating a new df with magicBytes, valueSchemaId & fixedValue
    wikimedia_value_df = wikimedia_df.select("magicByte", "valueSchemaId", "fixedValue")

    # write to sink(console)
    ## trigger is used for batch interval
    wikimedia_value_df \
    .writeStream \
    .format("console") \
    .outputMode("append") \
    .option("truncate", "true") \
     start() \
```

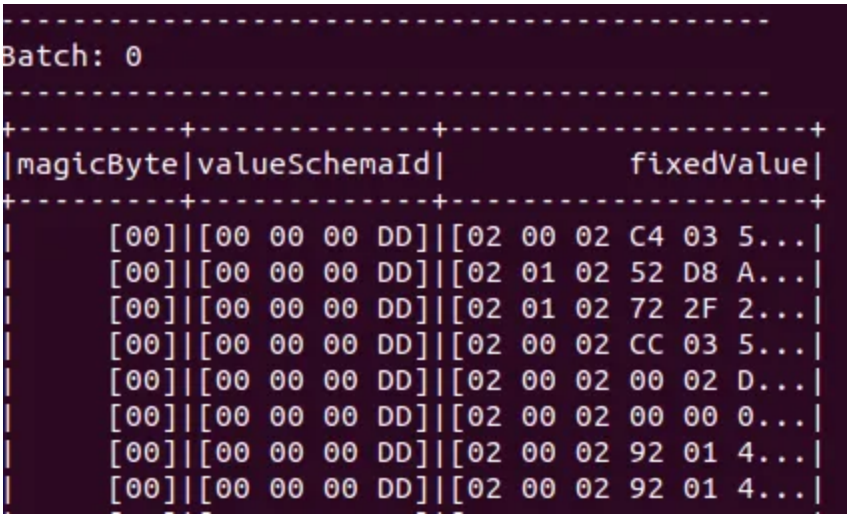
```
46        .awaitTermination()
47
48  spark_consumer()
```

magic_byte_schema_id_value.py hosted with ❤ by GitHub view raw

Let's understand the code:

- Line 1: We import spark sql functions as func.
- Line 27: We add a new column named **magicByte** to wikimedia_df. Since the value column obtained in PySpark is in string format, we utilize SQL's **substring** function to extract the Magic Byte from it. In SQL indexing starts at 1; therefore, for substring arguments, we pass 1 as the second parameter and specify that Magic Byte has a fixed length of one byte by passing 1 as the third argument. **withColumn** method expects a PySpark Column type in the second argument. Therefore, we enclose our SQL statement within the **func.expr()** function.
- Line 30: We add a new column named **valueSchemaId** to wikimedia_df. This column will store the ID of the schema used for serializing values in Kafka. This information can be useful for maintaining schema compatibility between producers and consumers. To extract the schema ID, we again use SQL's substring function. In this case, we'll start from the second position and specify a length of four bytes for the third parameter as the Schema ID has a fixed length of four bytes.
- Line 33: We add a new column named **fixedValue** to wikimedia_df. This column will store the actual serialized data. To extract the actual serialized data, we use SQL's substring function again. In this case, we skip the first 5 bytes (Magic Byte + Schema ID) by specifying a starting position of 6 in the second parameter and setting length to the length of the value column minus 5.

- Line 36: We create a new data frame containing the magicByte, valueSchemaId and fixedValue column from wikimedia_df.
- Line 40 to 46: Write to console sink to view the resulting data frame. We set the output mode to **append**.




magic byte, schema id and avro data

### Get Schema from Schema Registry

To get the schema from Schema Registry, we use a library like **confluent-kafka-python** which provides APIs for interacting with the Schema Registry server.

Let's understand the code:

- Line 1: We import the **confluent-kafka-python** package for interacting with the Schema Registry.
- Line 4 to 8: We define a function named **get_schema_from_schema_registry** that takes two arguments: schema_registry_url and schema_id. We get the latest version of the

schema from Schema Registry. We then return the latest_schema object which contains the schema details such as schema_id, version, subject and schema object.

### Deserializing Wikimedia data

We will use an AVRO deserializer to deserialize the data ingested by the producer in Kafka.

Let's understand the code:

- Line 1: We import the from_avro function for deserializing AVRO data.
- Line 4: We get the latest schema from the Registry.
- Line 7: We set the mode of the from_avro function to **PERMISSIVE**. This mode allows the deserializer to return **null** if there is a mismatch between the schema and the data being deserialized.
- Line 8 to 13: **from_avro** expects the serialized data in bytes format as the first argument and the schema string obtained as the second argument. We can optionally pass additional parameters such as the mode. We give it an alias to make it easier to reference in our code. We select the new **wikimedia** column and create a new data frame containing the deserialized data.
- Line 14 to 15: Finally, we extract all the fields from the deserialized data frame to be used for further processing and then we print its schema for reference.

```
root
 |-- bot: boolean (nullable = true)
 |-- comment: string (nullable = true)
 |-- id: integer (nullable = true)
 |-- length: struct (nullable = true)
 |    |-- new: integer (nullable = true)
 |    |-- old: integer (nullable = true)
 |-- meta: struct (nullable = true)
 |    |-- domain: string (nullable = true)
 |    |-- dt: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- offset: long (nullable = true)
 |    |-- partition: integer (nullable = true)
 |    |-- request_id: string (nullable = true)
 |    |-- stream: string (nullable = true)
 |    |-- topic: string (nullable = true)
 |    |-- uri: string (nullable = true)
 |-- minor: boolean (nullable = true)
 |-- namespace: integer (nullable = true)
 |-- parsedcomment: string (nullable = true)
 |-- patrolled: boolean (nullable = true)
 |-- revision: struct (nullable = true)
 |    |-- new: integer (nullable = true)
 |    |-- old: integer (nullable = true)
 |-- schema: string (nullable = true)
 |-- server_name: string (nullable = true)
 |-- server_script_path: string (nullable = true)
 |-- server_url: string (nullable = true)
 |-- timestamp: integer (nullable = true)
 |-- title: string (nullable = true)
 |-- type: string (nullable = true)
 |-- user: string (nullable = true)
 |-- wiki: string (nullable = true)
```

deserialised wikimedia dataframe

From the above screenshot, we can observe that the **bot** column is of **boolean type** and the **timestamp column** is of int type.

### Running Aggregations for Analysis

In this section, we will examine the number of bots and humans that are making edit requests.

Let's understand the code:

- Line 1: We import **TimestampType** from PySpark
- Line 3 to 5: We only consider data that falls under the category of **edit** as we filter by type. The timestamp column contains Unix time whose type is int. However, the watermark expects Timestamp type data. Therefore,

We need to convert it to the Timestamp format that PySpark can understand. Lastly, we select the **bot** and **timestamp** column.

- Line 8 to 19: We group by the **bot** column and apply a **count** aggregation. The groupBy function expects a window object, which defines a window for aggregation and the column(s) to group by. The window function expects a timestamp column, which will be used to define a window. In this case, we use a **sliding window** with a window interval of 60 seconds and a sliding interval of 30 seconds. We also specify a **watermark** of 60 seconds to ignore late data. This means that if a record arrives later than 60 seconds after its event time, it will be dropped from the aggregation. The behaviour of watermarks varies depending on the output mode. To learn more about watermarks in PySpark, refer to the official documentation.

### Writing Dataframe to Console Sink

Finally, we write the aggregated dataframe to a console sink using PySpark's **writeStream** function.

Let's understand the code:

- Line 1 to 6: We create a new column called **requested_by.** This column will contain the type of user making edit requests — Bot or Human. We achieve this by using PySpark's when and otherwise functions to set the value of requested_by based on a condition — if the bot column contains the value True, then the user is a **Bot**, if it contains the value False, then the user is a **Human** otherwise the value of the column is **null.**

- Line 7 to 11: We select window, requested_by and counts columns
- Line 12: We print the dataframe's schema
- Line 16 to 23: We write the dataframe to the console using PySpark's writeStream function. We have also set the output mode to **append** which means that only the results of the finalised window will be printed to the console. **processingTime** in trigger is set to 1 second which specifies how often the system should check for new data in the input source, It is also known as the **batch interval.**

```
root
 |-- window: struct (nullable = true)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- requested_by: string (nullable = true)
 |-- counts: long (nullable = false)
```

Processed Dataframe Schema

As you can see from the above screenshot that the schema of the processed data contains three columns — window, requested_by, and counts.

**window** column is a **struct** containing the start and end fields. **start** and **end** are of type timestamp.

```
-------------------------------------------
Batch: 5
-------------------------------------------
+------------------------------------------+------------+------+
|window                                    |requested_by|counts|
+------------------------------------------+------------+------+
|{2023-03-20 13:33:00, 2023-03-20 13:34:00}|Human       |24    |
|{2023-03-20 13:33:00, 2023-03-20 13:34:00}|Bot         |6     |
|{2023-03-20 13:33:30, 2023-03-20 13:34:30}|Bot         |59    |
|{2023-03-20 13:33:30, 2023-03-20 13:34:30}|Human       |265   |
+------------------------------------------+------------+------+
```

Processed Data

The processed data displays the aggregated counts of edit requests made by bots and humans within specified time windows.

### Insert Processed Data into Kafka

Now that we have processed and analyzed the data, the next step is to write this information back into Kafka in **AVRO** format for further downstream processing or reporting.

### Create Kafka Topic for the Processed Data

To do this, we need to log into our **kafka-broker** container. Once logged in run the following command:

```
kafka-topics - bootstrap-server kafka-broker:29092 - create - topic wikimedia.pr
```

This command will create a Kafka topic called **wikimedia.processed** with one partition and a replication factor of 1.

NOTE: The kafka cluster only has one broker, so set the replication factor to 1.

### Define AVRO Schema for the Processed Data

To insert the processed data into Kafka, we must first define an appropriate AVRO schema that corresponds to the structure of our processed DataFrame.

Below is the AVRO schema that I've created for our processed DataFrame:

As demonstrated in the code, this AVRO schema consists of three fields — window, requested_by, and counts. The **window** is considered a **record type**, containing two nested fields: **start** and **end**, both of which are **string types**.

Register this schema with the Schema Registry to ensure compatibility with downstream systems. To know how to register a schema follow my previous article.

### Convert the Timestamp columns (start and end) into String Types

The Big Data Cluster uses Schema Registry version 5.4.0-ce which does not support timestamp type; hence, we need to convert the **start** and **end** columns to **string format** before writing the data back to Kafka.

Let's understand the code:

- Line 3 to 4: Type casting start and end fields of window struct to string type and assign them aliases. This ensures that newly created columns do not have column names as ‘window.start’ and ‘window.end’. Instead, the aliases ensure that the new column names are ‘start’ and ‘end’, which makes it easier to work with these columns in future.
- Line 2 to 5: We create a struct and name it **window** containing the newly created start and end columns as nested columns.

### Serializing Processed Data

Next, we need to serialize the processed data using the AVRO schema defined earlier.

Let's understand the code:

- Line 1: We import **to_avro** to serialize the processed data using the AVRO schema defined earlier.
- Line 5 to 6: We define kafka topic and schema registry subject name for processed data.

- Line 12: We get the latest version of the schema from the Schema Registry.
- Line 15 to 20: We create a new Struct Column using **func.struct** to hold the processed data in a format compatible with the AVRO schema. The function **to_avro** can receive a Column data type and schema as inputs, and then produce a column of **binary-type data**. Therefore, This struct is then passed to the **to_avro** function, along with the stringified schema. Finally, the serialized AVRO data is stored in a new column named **value** for writing back to Kafka.

### Concatenate Magic Byte, Schema Id and Processed Data

To concatenate the Magic Byte, Schema Id, and processed data in binary format, we can use PySpark's built-in concat function.

We can use our previously defined UDF **int_to_binary_udf** to convert the protocol format (magic byte) and Schema Id to binary format before concatenating them with the processed data.

Let's understand the code:

- Line 2: As discussed protocol format is always set to 0 and it should be 1 byte in length. A UDF accepts Column or String-type values. So, We will use **lit** function to convert integers to Column type. Then we pass 0 as the value and 1 as the length to the UDF function.

- Line 3: As discussed earlier Schema Id is represented in 4 bytes. So, we pass schema id as the value and 4 as the length to the UDF function.
- Line 4: We concatenate the binary representation of the magic byte, schema id, and processed data using PySpark's concat function. Finally, we set the concatenated binary data as the **value** column. Kafka expects this column while sending data to it.

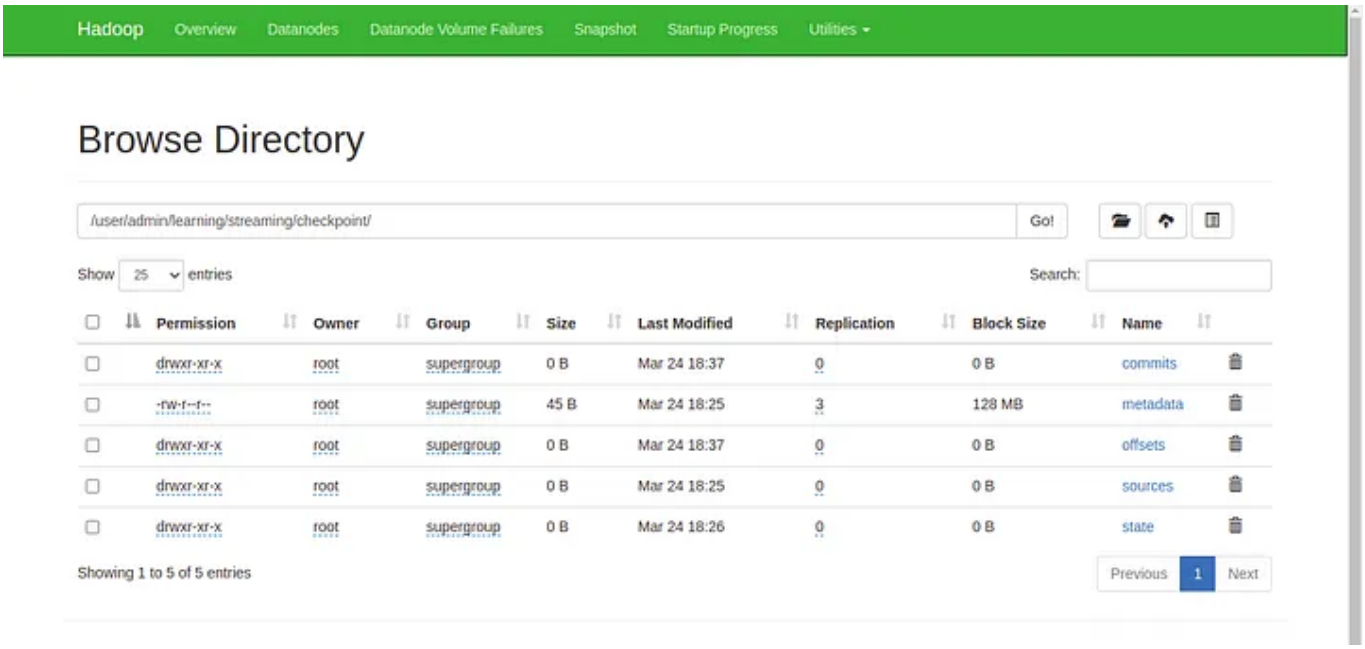
### Writing Processed Data to Kafka Sink

Finally, we write the processed dataframe to the Kafka sink.

Let’s understand the code:

- Line 7: We specify the Kafka broker Url to which we will write.
- Line 8: We write the processed data in the Kafka topic that we created earlier.
- Line 9: PySpark maintains an intermediate state in HDFS to recover from failures. This stores the batches of data processed successfully, PySpark

configurations, offsets and other metadata necessary to ensure fault tolerance and data consistency in case of failures or interruptions during the writing process to Kafka sink.




Checkpoint

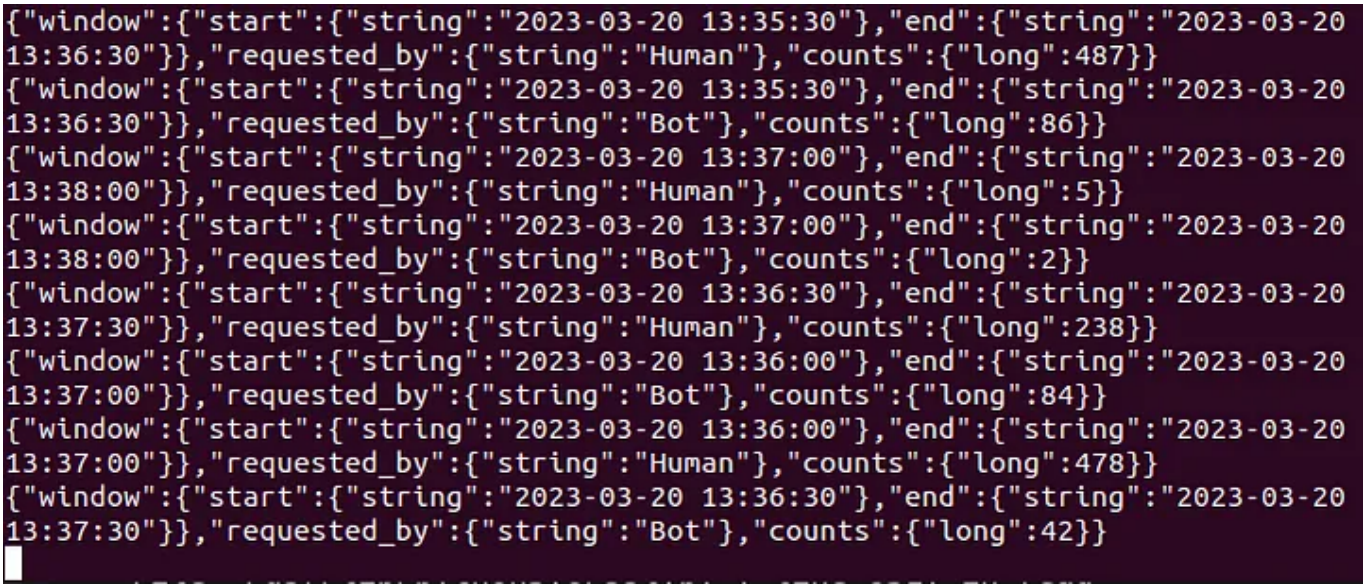
## Consume Processed Data

We will use CLI to consume the messages from Kafka topic. To do this, we need to log into our **schema-registry** container. Once logged in run the following command:

```
kafka-avro-console-consumer \
- bootstrap-server kafka-broker:29092 \
- topic wikimedia.processed \
- from-beginning \
- property schema.registry.url=http://schema-registry:8083
```

This command will consume the messages from Kafka topic and display them on CLI. This gets latest schema from the schema registry and uses it to

deserialize the binary data into a readable format.




Consuming Processed Data

## References:


- Deserialzing Confluent Avro Records in Kafka with Spark (June 19, 2020)| https://datachef.co/blog/deserialzing-confluent-avro-record-kafka-spark/
- How to Consume Data from Apache Kafka Topics and Schema Registry with Confluent and Azure Databricks (Feb 4, 2021)| https://www.confluent.io/blog/consume-avro-data-from-kafka-topics-and-secured-schema-registry-with-databricks-confluent-cloud-on-azure/
- unknown magic byte kafka connect | https://zditect.com/blog/59317864.html


Spark | Spark Streaming | Kafka | Data Analysis | Data Streaming


## Written by Mrugank Ray

7 Followers

Follow

### More from Mrugank Ray


Mrugank Ray

#### Create Avro Producer for Kafka using Python

We will create a python script to send avro data to kafka

5 min read · Feb 21

3

Mrugank Ray

#### Unlocking the Vault of Tech Interview Questions

As a Senior Engineer who has gone through several job interviews in the tech industry an...

10 min read · Jun 18

16


Mrugank Ray

## PostgreSQL vs MySQL | Choose What's Best For Your Project

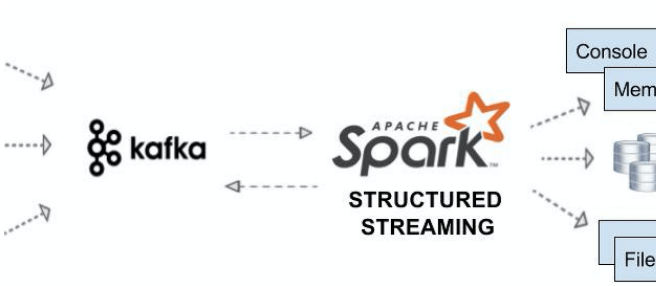
Introduction :There are many options available when it comes to database system...

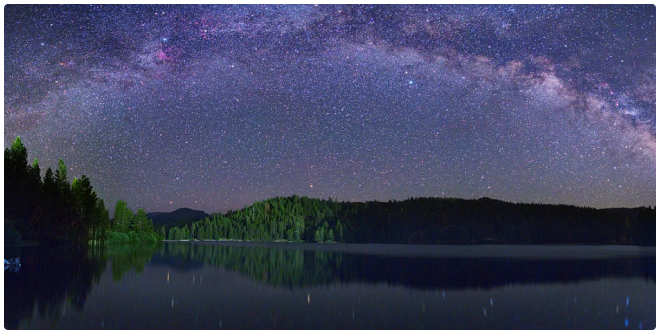
8 min read · Sep 25, 2022

1 1

See all from Mrugank Ray

# Recommended from Medium



Ahmed Uz Zaman in Dev Genius

## PySpark and Kafka Streaming: Reading and Writing Data in...

A Comprehensive Guide to Reading and Writing Data in Different Formats from Kafka...

· 4 min read · May 7

11

Pawan Kumar Ganjhu

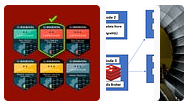
## PySpark Examples: Real-time, Batch, and Stream Processing for...

Real-time processing, batch processing, and stream processing are three different...

6 min read · May 25

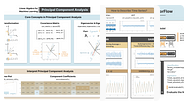
5

### Lists



**New_Reading_List**

174 stories · 174 saves



**Practical Guides to Machine Learning**

10 stories · 636 saves

**ChatGPT prompts**

27 stories · 602 saves

**It's never too late or early to start something**

15 stories · 192 saves

Clément Delteil in Towards AI

## Real-Time Sentiment Analysis with Docker, Kafka, and Spark...

A Step-By-Step Guide to Deploying a Pre-trained Model in an ETL Process

12 min read · May 6

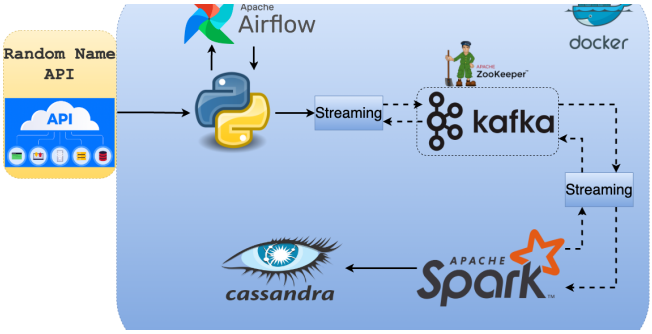

Bragadeesh Sundararajan

## Full-Stack Data Streaming: From Real-Time Generation with Kafka...

Real-time data processing is a methodology applied to process large volumes of data at...

15 min read · Oct 27

87 2 6




Dogukan Ulu

## Data Engineering End-to-End Project — Spark, Kafka, Airflow,...

First of all, please visit my repo to be able to understand the whole process better. This...

9 min read · Jul 14

1.3K 5

Ismael Sánchez Chaves in C# Programming

## Schema Registry in Kafka: Avro, JSON and Protobuf

The importance of having a structured data schema for messaging-based systems

7 min read · May 15

24 1

See more recommendations