Author: Madhurima Rawat

Streaming Analytics with Kafka and Spark: A Real-Time Data Pipeline for Rainfall Analysis

Set up a data streaming pipeline to ingest real-time rainfall data using Apache Kafka and process it with Apache Spark Streaming for effective real-time analytics.

Table of Contents

1. Introduction

Apache Spark: Powerful Open-Source Processing Engine

Apache Spark is an open-source, distributed data processing engine designed for large-scale data processing. It is known for its speed, ease of use, and its ability to handle sophisticated analytics, such

as machine learning and graph processing. Spark operates on a cluster of machines, enabling parallel processing of tasks across datasets, making it ideal for handling massive amounts of data.

Key Features of Apache Spark:

- **Speed**: Spark processes data in memory, significantly improving performance compared to traditional disk-based systems like Hadoop's MapReduce. It can be up to 100x faster for large-scale data processing.
- **Ease of Use**: It provides high-level APIs in popular languages like Python, Scala, and Java. The APIs allow developers to process data efficiently without dealing with complex distributed systems.
- **Advanced Analytics**: Spark includes libraries for SQL queries (Spark SQL), machine learning (MLlib), graph processing (GraphX), and streaming data (Spark Streaming), making it a versatile tool for analytics.

Real-Life Example: Big Data Analysis in Retail:

A major retail company like Amazon can use Apache Spark to analyze millions of transactions every day to understand customer purchasing behavior. By processing data in real-time, Spark helps detect trends, such as popular products, buying patterns, and potential inventory issues. For instance, if a product starts selling out rapidly, Spark can trigger an automatic restocking process based on sales trends, reducing the risk of running out of stock.

Apache Kafka: Distributed Streaming Platform

Apache Kafka is an open-source platform used for building real-time data pipelines and streaming applications. It is a highly scalable, fault-tolerant, and distributed system capable of handling trillions of events a day. Kafka works as a messaging system where data (messages) is sent from producers (applications or services) and stored in topics, and consumers (other applications) read those messages.

Key Features of Apache Kafka:

- **High Throughput and Scalability**: Kafka can process millions of messages per second, making it ideal for high-volume, real-time applications.
- **Durability and Fault-Tolerance**: Kafka stores data across a distributed cluster, ensuring that even if some parts of the system fail, the data remains available and consistent.
- **Real-Time Stream Processing**: Kafka is designed for real-time data streams, enabling developers to build applications that respond instantly to changes in data.

Real-Life Example: Real-Time Data Processing in Financial Services:

In the banking sector, Kafka is used for processing financial transactions in real-time. For example, a bank like JP Morgan might use Kafka to handle streams of stock market data and customer

transactions. Kafka ensures that transaction records are stored reliably and processed in real-time, enabling instant fraud detection or alert systems. If a suspicious transaction is detected, Kafka allows the system to immediately send alerts to security teams to investigate, minimizing the risk of fraud.

Integrating Apache Spark and Kafka: Real-Time Data Processing

Combining Apache Spark and Kafka allows for the seamless processing of real-time data streams. Kafka handles the ingestion of streaming data, while Spark processes this data for insights and advanced analytics.

Real-Life Example: Ride-Sharing Apps (e.g., Uber, Lyft):

Ride-sharing companies like Uber use a combination of Kafka and Spark for real-time analytics on rides, driver locations, and user requests. Kafka ingests streams of data from the app, such as new ride requests, driver availability, and GPS data. Spark Streaming processes this data in real-time to match riders with nearby drivers, optimize pricing based on demand (surge pricing), and monitor ride conditions. This real-time processing is critical to ensuring low wait times, improving driver efficiency, and enhancing customer experience.

By integrating Apache Kafka and Spark, Uber can handle millions of data points every second, ensuring smooth and reliable service across its global user base.

Integrating these two technologies allows us to process streams of data in real-time.

This guide will walk us through the steps required to set up Apache Spark with Kafka, including installation, configuration, and running a sample streaming application that processes rainfall data.

2. Prerequisites

Before proceeding, ensure that the following software are installed on user machine:

- **Apache Kafka**
- **Apache Spark**
- **Python packages**: Install the following Python packages using pip:

```
pip install pyspark kafka-python pandas
```

3. Setting Up Apache Spark

Download and Install Spark

1. **Download Spark**:
    - Visit the Apache Spark downloads page and download the latest version (e.g., `spark-3.x.x-bin-hadoop3.x.tgz`).
2. **Extract Spark**:
    - Use WinRAR or any extraction tool to decompress the downloaded `.tgz` file in usr `C:/` drive, which will create a folder named `spark-3.x.x-bin-hadoop3.x`.

Configure Environment Variables

1. **Set Environment Variables**:
    - Open the **Environment Variables** settings in Windows:
        - Right-click on **This PC** or **My Computer** > **Properties** > **Advanced system settings** > **Environment Variables**.
    - Under **User variables**, create a new variable:
        - **Variable name**: `SPARK_HOME`
        - **Variable value**: `C:/spark-3.x.x-bin-hadoop3.x` (adjust the version as necessary).
    - Under **System variables**, find the `Path` variable and add:
        - `C:/spark-3.x.x-bin-hadoop3.x/bin`
2. **Configure Python**:
    - Find the path to usr Python installation (e.g., `C:/Users/usrUsername/AppData/Local/Programs/Python/Python310/python.exe`) and copy it.
3. **Create Spark Configuration**:
    - Navigate to the `conf` folder inside usr Spark installation (e.g., `C:/spark-3.x.x-bin-hadoop3.x/conf`).
    - Create a new file named `spark-env.cmd`.
    - Open `spark-env.cmd` in a text editor and add the following line:

      ```
      set PYSPARK_PYTHON=C:\Users\usrUsername\AppData\Local\
      Programs\Python\Python310\python.exe
      ```

    - Save the file.

Test Spark Installation

1. Create a simple Python script (`test_spark.py`) in Visual Studio Code or any text editor with the following content:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Test Spark") \
    .getOrCreate()

data = [("Hello, Spark!",)]
df = spark.createDataFrame(data, ["message"])
df.show()

spark.stop()
```

2. Open the command prompt in the folder where the Python file is saved and run the script using:

```
spark-submit test_spark.py
```

4. Setting Up Apache Kafka

Download and Install Kafka

1. **Download Kafka**:
   - Go to the Apache Kafka downloads page and download the latest version (e.g., `kafka_2.12-3.x.x.tgz`).
2. **Extract Kafka**:
   - Use WinRAR to extract the downloaded file and save it in `C:/Kafka`.

Configure Environment Variables for Kafka

1. Open the **Environment Variables** settings again.
2. Under **User variables**, create a new variable:
   - **Variable name**: `KAFKA_HOME`
   - **Variable value**: `C:/Kafka` (or wherever us extracted Kafka).
3. Under **System variables**, find the `Path` variable and add:
   - `C:/Kafka/bin`

5. Running Kafka

Start Zookeeper

1. Open a command prompt and navigate to the Kafka `bin` directory:

```
cd C:/Kafka/bin
```

2. Start Zookeeper with the following command:

```
.\windows\zookeeper-server-start.bat ..\config\zookeeper.properties
```

Start Kafka Broker

1. Open another command prompt and navigate to the Kafka `bin` directory:

```
cd C:/Kafka/bin
```

2. Start the Kafka broker:

```
.\windows\kafka-server-start.bat ..\config\server.properties
```

Create a Kafka Topic

1. Open another command prompt and navigate to the Kafka `bin` directory:

```
cd C:/Kafka/bin
```

2. Create a topic named `rainfall_data` :

```
.\windows\kafka-topics.bat --create --topic rainfall_data
--bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```

6. Kafka Producer Code

There are two ways to send rainfall data to Kafka:

1. Using JSON Objects Directly

we can manually send JSON data using Kafka's console producer. Here's how we can do it:

- After running the kafka producer command, we can input JSON-formatted rainfall data one by one in the console:

```
{
  "division": "North",
  "year": "2023",
  "jan": 2.5,
```

```
  "feb": 1.5,
  "mar": 3.0,
  "apr": 4.0,
  "may": 5.0,
  "jun": 6.0,
  "jul": 7.0,
  "aug": 8.0,
  "sep": 9.0,
  "oct": 10.0,
  "nov": 11.0,
  "dec": 12.0,
  "annual": 78.0
}
```

- This method requires manual input for each data point we want to send to Kafka.

2. Using a CSV File with Python

If we have a large dataset in CSV format, we can use a Python script to automate sending data to Kafka. Follow these steps:

- Save the following code as `kafka_producer.py`：

```
import pandas as pd
from kafka import KafkaProducer
import json

# Load the dataset
df = pd.read_csv("rainfall in india 1901-2015.csv")

# Initialize Kafka producer
producer = KafkaProducer(
    bootstrap_servers="localhost:9092",
    value_serializer=lambda x: json.dumps(x).encode("utf-8"),
)

# Send data to Kafka topic 'rainfall_data'
for index, row in df.iterrows():
    data = {
        "division": row["DIVISION"],
        "year": row["YEAR"],
        "jan": row["JAN"],
        "feb": row["FEB"],
        "mar": row["MAR"],
        "apr": row["APR"],
        "may": row["MAY"],
        "jun": row["JUN"],
        "jul": row["JUL"],
        "aug": row["AUG"],
        "sep": row["SEP"],
```

```
            "oct": row["OCT"],
            "nov": row["NOV"],
            "dec": row["DEC"],
            "annual": row["ANNUAL"],
            "jan_feb": row["Jan-Feb"],
            "mar_may": row["Mar-May"],
            "jun_sep": row["Jun-Sep"],
            "oct_dec": row["Oct-Dec"],
        }
        producer.send("rainfall_data", value=data)

# Ensure all messages are sent before closing
producer.flush()
producer.close()
```

Detailed Explanation of the Producer Code

1. **Loading the Dataset**:
   The `pandas` library is used to load the CSV file (`rainfall in india 1901-2015.csv`), which contains historical rainfall data for each division and year in India.

2. **Initializing Kafka Producer**:
   - `bootstrap_servers="localhost:9092"`: This connects the producer to the Kafka server running locally on port 9092.
   - `value_serializer`: The data is serialized to JSON format and encoded as UTF-8 before being sent to Kafka.

3. **Sending Data to Kafka**:
   - The `for` loop iterates over each row of the CSV dataset.
   - For each row, the script creates a dictionary (`data`) containing all relevant fields, such as the division, year, monthly rainfall data, and aggregated values like `Jan-Feb`, `Mar-May`, etc.
   - `producer.send("rainfall_data", value=data)` sends the data to the Kafka topic `rainfall_data`.

4. **Flushing and Closing**:
   - `producer.flush()` ensures that all messages are sent to Kafka before closing the producer.
   - `producer.close()` closes the connection.

This approach is ideal when handling large datasets that would be impractical to input manually.

7. Spark Streaming Code

To process the data from Kafka, save the following code as `spark_streaming.py` :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg, from_json, count, min, max, mean
from pyspark.sql.types import StructType, StructField, StringType, FloatType

# Initialize Spark Session with reduced log output
spark = SparkSession.builder.appName("Real-Time Rainfall Data Processing").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")  # Set log level to ERROR to suppress INFO logs

# Read stream from Kafka topic 'rainfall_data'
kafka_stream = (
    spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "localhost:9092")
    .option("subscribe", "rainfall_data")
    .load()
)

# Deserialize the JSON data
schema = StructType(
    [
        StructField("division", StringType(), True),
        StructField("year", StringType(), True),
        StructField("jan", FloatType(), True),
        StructField("feb", FloatType(), True),
        StructField("mar", FloatType(), True),
        StructField("apr", FloatType(), True),
        StructField("may", FloatType(), True),
        StructField("jun", FloatType(), True),
        StructField("jul", FloatType(), True),
        StructField("aug", FloatType(), True),
        StructField("sep", FloatType(), True),
        StructField("oct", FloatType(), True),
        StructField("nov", FloatType(), True),
        StructField("dec", FloatType(), True),
        StructField("annual", FloatType(), True),
        StructField("jan_feb", FloatType(), True),
        StructField("mar_may", FloatType(), True),
        StructField("jun_sep", FloatType(), True),
        StructField("oct_dec", FloatType(), True),
    ]
)

# Parse the value column
parsed_stream = (
    kafka_stream.selectExpr("CAST(value AS STRING) ")
    .select(from_json(col("value"), schema).alias("data"))
    .select("data.*")
)

# Show column names and schema
parsed_stream.printSchema()
```

```
# Calculate descriptive statistics: count, min, max, mean, etc.
descriptive_stats = parsed_stream.groupBy("division").agg(
    count("*").alias("count"),
    min("annual").alias("min_annual_rainfall"),
    max("annual").alias("max_annual_rainfall"),
    mean("annual").alias("mean_annual_rainfall"),
)

# Write the descriptive statistics to the console
query = descriptive_stats.writeStream.outputMode("complete").format("console").start()

# Await termination (uncomment if we want it to run indefinitely)
query.awaitTermination()
```

Explanation of the Streaming Code

- **Initialize Spark Session**: Sets up the Spark session for streaming.
- **Read Stream from Kafka**: Reads the stream of data from the `rainfall_data` Kafka topic.
- **Deserialize JSON Data**: Defines the schema for the incoming JSON data and parses the Kafka messages.
- **Aggregation**: Groups the data by division and calculates the average annual rainfall.
- **Output**: Writes the result to the console in real-time.

8. Running the Application

1. **Start Zookeeper and Kafka**:
    - Ensure Zookeeper and Kafka are running as described in the section **Running Kafka**.
2. **Run the Kafka Producer**:

Kafka Producer is used to send data to a Kafka topic. Here, we are sending rainfall data in two ways: directly as JSON input via a command prompt and through a CSV file using a Python script.

- **Direct Dictionary Input Method**:
    - Navigate to Kafka's `bin` folder using the command prompt:

      ```
      c:\Kafka\bin\windows
      ```

    - Run the Kafka producer and send JSON data to the topic `rainfall_data` :

      ```
      .\kafka-console-producer.bat
      --broker-list localhost:9092 --topic rainfall_data
      ```

- Example input:

```
{
  "division": "North",
  "year": "2023",
  "jan": 2.5,
  "feb": 1.5,
  "mar": 3.0,
  "apr": 4.0,
  "may": 5.0,
  "jun": 6.0,
  "jul": 7.0,
  "aug": 8.0,
  "sep": 9.0,
  "oct": 10.0,
  "nov": 11.0,
  "dec": 12.0,
  "annual": 78.0
}
```

- Press `Ctrl+C` to terminate the continuous input process. Upon receiving the prompt:

```
Terminate batch job (Y/N)? Y
```

  - Type `Y` to terminate the process.

After this, run the Kafka consumer to verify the data:

```
.\kafka-console-consumer.bat --bootstrap-server
localhost:9092 --topic rainfall_data --from-beginning
```

- **CSV File Input Method**:
  - Run the Kafka producer Python script (`kafka_producer.py`):

```
python kafka_producer.py
```

  - This script reads from a CSV file and sends data to Kafka, similar to the manual dictionary input.

If both commands execute successfully, the data will be sent to Kafka, and any issues will indicate errors in configuration or file paths.

3. **Run the Spark Streaming Application**:

When we run the Spark Streaming application using the command:

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.13:3.5.2 spark_streaming.py
```

Explanations:

1. **Submitting the Spark Job:**
   - `spark-submit` submits the Spark application (`spark_streaming.py`) to the cluster. The `--packages` flag specifies necessary dependencies, allowing Spark to integrate with Kafka.
2. **Spark-Kafka Integration:**
   - Kafka is a distributed streaming platform for real-time data feeds. Spark Streaming processes these feeds by consuming messages from Kafka topics. The specified package (`spark-sql-kafka-0-10_2.13:3.5.2`) enables Spark to interact with Kafka, with **2.12 representing the Scala version** and **3.5.2 being the Spark version**. The Kafka version being used is **3.8.0**.
3. **Execution of `spark_streaming.py` :**
   - The script establishes a Spark session, defines Kafka connection parameters, and specifies streaming transformations (e.g., filtering and aggregating data).
4. **Kafka as a Data Source:**
   - Spark connects to Kafka brokers, subscribing to specified topics to pull real-time data, processing it in parallel across partitions.
5. **Real-Time Data Processing:**
   - Incoming Kafka data is processed in micro-batches at defined intervals (configured in `spark_streaming.py`). Transformations are applied, and results can be outputted to various destinations.
6. **Continuous Execution:**
   - Spark continuously listens to Kafka for new data as long as the job is running, enabling real-time processing.
7. **Monitoring the Application:**
   - Spark logs provide progress updates and error information. A web UI (accessible at `http://localhost:4040`) offers detailed metrics about the job.
8. **Example of Output:**
   - As Spark processes data, logs indicate connections to Kafka, batch processing progress, and completion statuses.

Key Points:

- **Real-Time Processing:** Spark Streaming handles real-time data from Kafka, processing messages as they are published.
- **Batching:** Data is processed in micro-batches for efficiency.
- **Scalability:** Spark's distributed nature allows it to manage large data volumes.
- **Continuous Operation:** The application runs continuously, processing new data as it arrives.

In summary, this command sets up a continuous data pipeline between Kafka and Spark Streaming, allowing for immediate processing of incoming data.

Note: The `spark-sql-kafka-0-10_2.13:3.5.2` package uses **2.13 for the Scala version**, **3.5.2 for the Spark version**, and **3.8.0 for the Kafka version**.

Spark's execution outputs are displayed as follows:

```
C:\Users\rawat\Documents\7 SEMESTER\Big Data Analytics\Lab\Experiment 6>spark-submit
--packages org.apache.spark:spark-sql-kafka-0-10_2.


        org.slf4j#slf4j-api;2.0.7 from central in [default]
        org.xerial.snappy#snappy-java;1.1.10.5 from central in [default]
        ---------------------------------------------------------------------
        |                  |            modules            ||   artifacts   |
        |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
        ---------------------------------------------------------------------
        |      default     |   11  |   0   |   0   |   0   ||   11  |   0   |
        ---------------------------------------------------------------------
```

This output is showing the dependency resolution process for the `spark-submit` command, specifically when downloading the necessary packages and dependencies for Spark and Kafka.

1. **Resolving Dependencies:**
    - The command is resolving the required dependencies from online repositories (like Maven Central). It checks if the necessary libraries, like `slf4j-api` and `snappy-java` , are available or need to be downloaded.
2. **Dependency Versions:**
    - It lists specific versions of libraries that are needed, such as `slf4j-api;2.0.7` and `snappy-java;1.1.10.5` . These versions are fetched from the central repository.
3. **Artifact Information:**
    - The table format shows columns related to configuration (`conf`), the number of modules/artifacts, how many were searched, downloaded, and any that were evicted (if

versions were conflicting).

4. **Modules and Artifacts Count:**
    - The table reports that 11 modules (or dependencies) are required, but none have been downloaded yet (`dwnlded = 0`), indicating that the process of fetching them is either still in progress or was cached locally.
5. **No Downloaded Dependencies Yet:**
    - Since the download count is `0` , it means none of the required artifacts were downloaded at the time of this message. This could be because they are already cached or the download hasn't started yet.

This output is part of the process where Spark fetches dependencies needed for the streaming job to connect with Kafka.

```
root
|-- division: string (nullable = true)
|-- year: string (nullable = true)
|-- jan: float (nullable = true)
|-- feb: float (nullable = true)
|-- mar: float (nullable = true)
|-- apr: float (nullable = true)
|-- may: float (nullable = true)
|-- jun: float (nullable = true)
|-- jul: float (nullable = true)
|-- aug: float (nullable = true)
|-- sep: float (nullable = true)
|-- oct: float (nullable = true)
|-- nov: float (nullable = true)
|-- dec: float (nullable = true)
|-- annual: float (nullable = true)
|-- jan_feb: float (nullable = true)
|-- mar_may: float (nullable = true)
|-- jun_sep: float (nullable = true)
|-- oct_dec: float (nullable = true)
```

Explanation:

1. **`division` (string):**
    - This column represents the name of a specific geographical division where rainfall data is collected. This could refer to various regions, states, or districts (e.g., "Northeast," "Southeast," "California").
    - The data type is `string` , allowing for a variety of text values, and it allows null entries, indicating that some records may not have a specified division.
2. **`year` (string):**

- This column captures the year for which the rainfall data is reported. Though the year is typically numerical (like "2023"), it is stored as a string to accommodate any specific formatting or annotations (such as "FY2023" for fiscal year).
- Similar to the division column, it permits null values, which means that there may be instances where the year information is missing.

3. **`jan` to `dec` (float):**
   - These columns represent the monthly rainfall measurements for each month from January to December, recorded as floating-point numbers. Each value indicates the total rainfall (in millimeters, centimeters, etc.) for that specific month in the corresponding division and year.
   - All these columns are nullable, meaning that there could be months without recorded rainfall data.

4. **`annual` (float):**
   - This column summarizes the total annual rainfall for the corresponding division and year. It aggregates the monthly values from January to December, providing a comprehensive view of yearly rainfall patterns.
   - The annual rainfall value is also nullable, indicating that it might be absent in some records.

5. **`jan_feb`, `mar_may`, `jun_sep`, `oct_dec` (float):**
   - These columns represent aggregated rainfall over specific periods:
     - **`jan_feb`**: Total rainfall for January and February.
     - **`mar_may`**: Total rainfall for March, April, and May (often the spring season).
     - **`jun_sep`**: Total rainfall for June through September (usually summer).
     - **`oct_dec`**: Total rainfall for October through December (typically autumn and winter).
   - Like the other columns, these values are stored as floats and can also be nullable, meaning some periods might not have recorded data.

Summary of the Schema in Context of Rainfall Trends:

This schema is structured to effectively capture and analyze rainfall trends across different geographical divisions over time. It allows for a detailed monthly breakdown and summarizes yearly trends, enabling comparisons and analyses of rainfall patterns. This information can be vital for agricultural planning, water resource management, and understanding climate changes in various regions.

Batch: 0

```
+--------+-----+------------------+------------------+-------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
```

```
+--------+-----+------------------+------------------+-------------------+
+--------+-----+------------------+------------------+-------------------+
```

- **Structure**: The table structure is defined with five columns: division, count, min_annual_rainfall, max_annual_rainfall, and mean_annual_rainfall.
- **No Data**: The output indicates no records are available, as all values are null.
- **Data Types**: The columns are set to capture string values for divisions and float values for rainfall statistics.

Batch: 1

```
+--------+-----+------------------+------------------+-------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+-------------------+
| NULL| 1| NULL| NULL| NULL|
+--------+-----+------------------+------------------+-------------------+
```

- **Single Null Record**: The table contains one entry with all values as null, indicating a possible placeholder or error in data entry.
- **Data Count**: The count of entries is 1, but all rainfall statistics are missing.
- **Division Status**: This could imply incomplete or missing data collection for a specific division.

Batch: 2

```
+--------+-----+------------------+------------------+-------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+-------------------+
| NULL| 1| NULL| NULL| NULL|
| North| 1| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+-------------------+
```

- **North Division Data**: A new entry for the "North" division has been added, reporting consistent annual rainfall metrics.
- **Rainfall Statistics**: The minimum, maximum, and mean annual rainfall are all recorded as 78.0, indicating uniformity in the data.
- **NULL Division**: The entry with NULL for division remains, reflecting possible missing data.

Batch: 3

```
+--------+-----+------------------+------------------+------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+------------------+
| NULL| 1| NULL| NULL| NULL|
| North| 2| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+------------------+
```

- **Increment in North Division Count**: The count for the "North" division has increased to 2, suggesting additional records were processed.
- **Consistent Rainfall Data**: The rainfall statistics for the North division remain unchanged from Batch 2.
- **NULL Division Count**: The NULL entry count persists, possibly indicating ongoing data issues.

Batch: 4

```
+--------+-----+------------------+------------------+------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+------------------+
| NULL| 2| NULL| NULL| NULL|
| North| 2| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+------------------+
```

- **NULL Division Entries**: The number of NULL entries remains at 2, indicating potential gaps in the dataset.
- **North Division Consistency**: The "North" division maintains its annual rainfall data, reflecting consistent results.
- **Data Integrity**: The data is showing some consistency, but the NULL entries suggest incomplete data collection.

Batch: 5

```
+--------+-----+------------------+------------------+------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+------------------+
| NULL| 2| NULL| NULL| NULL|
| East| 1| 79.2| 79.2| 79.19999694824219|
| North| 2| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+------------------+
```

- **East Division Introduced**: A new division, "East," has been added with specific rainfall statistics.

- **Multiple Divisions**: The table now reflects data for four divisions: NULL, East, North, and West.
- **Variability in Rainfall**: The introduction of varied rainfall metrics demonstrates a more comprehensive dataset.

Batch: 6

```
+--------+-----+------------------+------------------+-------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+-------------------+
| NULL| 2| NULL| NULL| NULL|
| East| 1| 79.2| 79.2| 79.19999694824219|
| West| 1| 54.0| 54.0| 54.0|
| North| 2| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+-------------------+
```

- **West Division Data**: The addition of the "West" division provides further geographical diversity to the data set.
- **NULL Division Count Remains**: There are still entries with NULL values, reflecting a lack of complete data.
- **Consistent Reporting**: Rainfall values for all divisions are being recorded consistently.

Batch: 7

```
+--------+-----+------------------+------------------+-------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+-------------------+
| NULL| 2| NULL| NULL| NULL|
| South| 2| 89.5| 89.5| 89.5|
| Central| 1| 66.5| 66.5| 66.5|
| East| 1| 79.2| 79.2| 79.19999694824219|
| West| 1| 54.0| 54.0| 54.0|
| North| 2| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+-------------------+
```

- **South and Central Divisions Added**: New divisions "South" and "Central" provide additional context to the rainfall data.
- **Greater Data Diversity**: The inclusion of multiple divisions with varied rainfall metrics enhances data richness.
- **Steady Progression**: The data collection appears to be progressing steadily, with more divisions being accounted for.

Batch: 8

```
+--------+-----+------------------+------------------+-------------------+
|division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------+-----+------------------+------------------+-------------------+
| NULL| 2| NULL| NULL| NULL|
| South| 2| 89.5| 89.5| 89.5|
| Central| 1| 66.5| 66.5| 66.5|
| East| 1| 79.2| 79.2| 79.19999694824219|
| West| 1| 54.0| 54.0| 54.0|
| North| 3| 78.0| 78.0| 78.0|
+--------+-----+------------------+------------------+-------------------+
```

- **Increased North Division Count**: The count for the "North" division has increased to 3, showing ongoing data accumulation.
- **NULL Entries**: The table still retains NULL entries, highlighting areas that need attention.
- **Diverse Dataset**: The overall dataset becomes increasingly diverse with various divisions reporting rainfall statistics.

Batch: 9

```
+--------------------+-----+------------------+------------------+-------------------+
| division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------------------+-----+------------------+------------------+-------------------+
| NULL| 2| NULL| NULL| NULL|
| South| 2| 89.5| 89.5| 89.5|
|ANDAMAN & NICOBAR...| 51| 2352.1| NaN| NaN|
| Central| 1| 66.5| 66.5| 66.5|
| East| 1| 79.2| 79.2| 79.19999694824219|
| West| 1| 54.0| 54.0| 54.0|
| North| 3| 78.0| 78.0| 78.0|
+--------------------+-----+------------------+------------------+-------------------+
```

- **Andaman & Nicobar Islands Data**: Significant data entry for this division shows high rainfall values, indicating geographic variability.
- **NULL Division Persistence**: The NULL entry remains, indicating persistent gaps in the dataset.
- **Variability in Rainfall**: The introduction of data for regions with high rainfall highlights regional differences across divisions.

Batch: 10

```
+--------------------+-----+-------------------+-------------------+--------------------+
| division|count|min_annual_rainfall|max_annual_rainfall|mean_annual_rainfall|
+--------------------+-----+-------------------+-------------------+--------------------+
| VIDARBHA| 115| 578.5| 1606.3| 1095.4591303286345|
|NAGA MANI MIZO TR...| 115| 1353.8| 4316.2| 2433.619123641304|
| CHHATTISGARH| 115| 904.6| 1974.0| 1371.7286875849186|
| NULL| 2| NULL| NULL| NULL|
|SUB HIMALAYAN WES...| 115| 1988.2| 3655.1| 2752.2173955502717|
|GANGETIC WEST BENGAL| 115| 1015.1| 2099.8| 1490.4878226902174|
| HIMACHAL PRADESH| 115| 776.1| 1919.2| 1260.3452153744904|
| BIHAR| 115| 629.2| 1660.4| 1197.63390847911|
| ORISSA| 115| 987.0| 1945.3| 1458.1695694633152|
| JAMMU & KASHMIR| 115| 657.0| NaN| NaN|
| ASSAM & MEGHALAYA| 115| 1743.4| 3403.5| 2580.695658542799|
| South| 2| 89.5| 89.5| 89.5|
| LAKSHADWEEP| 114| 992.6| NaN| NaN|
|ANDAMAN & NICOBAR...| 110| 1849.4| NaN| NaN|
| TAMIL NADU| 115| 318.0| 1365.3| 943.7130437436311|
| Central| 1| 66.5| 66.5| 66.5|
|NORTH INTERIOR KA...| 115| 470.3| 1095.6| 717.7956508470618|
| WEST UTTAR PRADESH| 115| 371.9| 1244.2| 827.1147813879925|
|SOUTH INTERIOR KA...| 115| 733.3| 1409.5| 1040.3913027556046|
| EAST MADHYA PRADESH| 115| 653.8| 1747.1| 1205.000001061481|
+--------------------+-----+-------------------+-------------------+--------------------+
only showing top 20 rows
```

- **Detailed Division Statistics**: Multiple divisions report extensive rainfall data, showing a detailed breakdown of annual metrics.
- **NULL Entry Reduction**: The number of NULL entries has decreased, indicating improved data integrity.
- **Rich Data Presentation**: The table presents a comprehensive overview of rainfall trends across various divisions, useful for analysis and decision-making.

Final Summary Note

Overall, the data collection process demonstrates progress in compiling rainfall statistics across multiple divisions, highlighting significant variability in annual rainfall figures. However, the presence of NULL entries in several batches indicates areas where data integrity could be improved, suggesting that further efforts may be needed to ensure comprehensive coverage of all divisions. The increasing richness of the dataset provides valuable insights into regional rainfall patterns that are essential for environmental and agricultural planning.

This process is conducted through Apache Spark, which continuously ingests data from Kafka. As soon as the data is entered into Kafka, it is processed in real time by Spark. To terminate the continuous input process, follow these steps:

- Press `Ctrl+C` to stop the process. When prompted with the following message displayed twice:

```
Terminate batch job (Y/N)? Y
```

- Type `Y` both times to successfully terminate the process.

9. Monitoring and Debugging

- **Check Kafka Topic**: Monitor the output of usr Kafka topic using:

```
.\windows\kafka-console-consumer.bat --bootstrap-server
localhost:9092 --topic rainfall_data --from-beginning
```

- **Review Spark Console Output**: Observe the console output of the Spark Streaming application for any logs or errors.

10. Conclusion

By following this guide, we have successfully set up Apache Spark with Kafka for real-time data processing. we can now build and expand upon this setup to include more complex data analytics and processing pipelines.

11. Resources

- Apache Spark Documentation
- Apache Kafka Documentation
- Kafka-Python Library
- Pandas Documentation