Large Scale Data Management
Assignment II

March 28, 2025

Kagioglou Maria

Part I: Stream of Movie Ratings Using Apache Kafka

The aim of this part of the project is to generate a real-time stream of movie ratings using Python and Apache Kafka. The system produces a stream of ratings that simulate a group of people rating various movies. This data is sent to Kafka for further consumption and processing. The system is designed to generate a random list of users, movies, and ratings, using the provided movies.csv file containing movie titles.
To achieve this goal, the `producer.py` script was developed. The primary objective of this script is to simulate a stream of movie ratings by randomly selecting users and movies, generating a random rating, and sending this data to Kafka in real-time. The script runs in an infinite loop, continuously generating and sending movie ratings to Kafka. This process ensures that the data is sent periodically, allowing for the simulation of a real-time data stream. By using the `Faker` library to generate random names and the movie titles from the provided CSV file, the script mimics a diverse set of users interacting with a large pool of movies, creating a dynamic movie rating system.
The core tasks that the `producer.py` script performs are as follows:

1. **KAFKA_BROKER**: This variable defines the address of the Kafka broker that the producer will connect to. In this case, the broker is running on `localhost` with port `29092`. The broker is responsible for handling the messages sent by the producer and making them available to consumers.

   ```
   KAFKA_BROKER = "localhost:29092"
   ```

   This configuration assumes that the Kafka service is running locally within the Vagrant environment, and the port `29092` is exposed for communication.

2. **KAFKA_TOPIC**: This variable specifies the Kafka topic where the movie ratings data will be published. In this case, the topic is called `test`, which will store the generated movie rating events.

   ```
   KAFKA_TOPIC = "test"
   ```

3. **Loading Movie Titles from a CSV File:** The script reads movie titles from a CSV file (`movies.csv`) located in the `/vagrant/data/` directory. This file contains a list of movie titles, which will later be used for generating movie ratings.

```
movies_df = pd.read_csv("/vagrant/data/movies.csv", header=0)
movies = movies_df.iloc[:, 0].tolist()
```

4. **Generating Random Users:** The script also simulates a list of users using the `Faker` library, which generates random names. It creates a list of 10 fake names (randomly generated) and adds one real name (mine), "Maria", to simulate a group of people rating movies. The list is created as follows:

```
users = [fake.name() for _ in range(10)] + ["Maria"]
```

5. **Generating Movie Ratings:** In the main loop, the script selects a random user from the `users` list and a random movie from the `movies` list. For each iteration, a random rating between 1 and 10 is generated. The current timestamp is also captured to indicate when the rating was made. This data is structured into a Python dictionary:

```
user = random.choice(users)
        movie = random.choice(movies)
        rating = random.randint(1, 10)
        timestamp = datetime.utcnow().isoformat()

data = {
    "user": user,
    "movie": movie,
    "rating": rating,
    "timestamp": timestamp
}
```

6. **Sending Data to Kafka:** The script then sends the generated data (user, movie, rating, timestamp) to a Kafka topic. It uses the `AIOKafkaProducer` from the `aiokafka` library to asynchronously send the data to the Kafka topic specified as `KAFKA_TOPIC`. The message is serialized in JSON format before being sent:

```
await producer.send(KAFKA_TOPIC, data)
print(f"Sent: {data}")
```

7. **Real-time Streaming:** After sending the data, the script waits for a random period (ranging from 5 seconds to 60 seconds) before generating the next movie

rating. This sleep interval ensures the data stream occurs at irregular intervals, simulating a real-time scenario:

```
await asyncio.sleep(random.randint(5, 60))
```

Once the `producer.py` script has been created, we execute the following steps in the Vagrant environment to start the necessary services and run the script:

1. **Start the Vagrant Environment:** To begin, bring up the Vagrant virtual machine by running the following command:

   ```
   vagrant up
   ```

2. **SSH into the Vagrant Environment:** Once the VM is up, SSH into it to access the environment:

   ```
   vagrant ssh
   ```

3. **Start Kafka and Zookeeper Using Docker:** Kafka and Zookeeper are running inside Docker containers. To start them, use the following command:

   ```
   docker-compose up -d
   ```

   This command starts Kafka and Zookeeper in detached mode.

4. **Check Kafka Container Status:** To ensure that the Kafka container is running, use the following command to list the running Docker containers:

   ```
   docker ps
   ```

5. **Check if the Kafka Topic Exists:** To check if the desired Kafka topic already exists, run the following command (the name of docker as it is specified in `docker ps`:

   ```
   docker exec -it kafka kafka-topics --list --bootstrap-server localhost:29092
   ```

   This command lists all the Kafka topics on the specified Kafka broker.

6. **Check Kafka Logs for Errors:** To troubleshoot any issues, view the last 20 lines of the Kafka container's logs using the command:

   ```
   docker logs kafka | tail -20
   ```

   This will show recent Kafka logs and help diagnose any potential problems.

7. **Run the Producer Script:** Finally, run the `producer.py` script to start producing movie ratings and streaming them to Kafka:

   ```
   python3 producer.py
   ```

   When the `producer.py` script is executed, it generates a continuous stream of movie ratings data and sends it to the Kafka topic. Below is a sample of the output displayed in the terminal during the execution of the script:

```
Sent: {'user': 'Henry Payne', 'movie': 'Kalushi: The Story of
    Solomon Mahlangu', 'rating': 9, 'timestamp': '2025-03-25
   T09:51:42.365162'}
Sent: {'user': 'Maria', 'movie': 'MR. RIGHT', 'rating': 2, '
   timestamp': '2025-03-25T09:52:13.394683'}
Sent: {'user': 'Andrea Jones', 'movie': 'LEGENDS OF THE
   HIDDEN TEMPLE', 'rating': 1, 'timestamp': '2025-03-25T09
   :52:26.406194'}
Sent: {'user': 'Stacie Martinez', 'movie': 'Message from the
   King', 'rating': 6, 'timestamp': '2025-03-25T09
   :53:26.467442'}
Sent: {'user': 'Henry Payne', 'movie': 'Halkaa', 'rating':
   10, 'timestamp': '2025-03-25T09:53:42.472256'}
Sent: {'user': 'Eric Rodriguez', 'movie': 'Frankensteins
   Monsters Monster, Frankenstein', 'rating': 9, '
   timestamp': '2025-03-25T09:54:35.521083'}
Sent: {'user': 'Maria', 'movie': 'How to Be a Player', '
   rating': 6, 'timestamp': '2025-03-25T09:55:18.565377'}
```

Part II: Developing a PySpark Script for Processing and Persisting Movie Ratings

In the second part of the project, we implement a PySpark script that receives, processes, and persists the movie ratings data produced by the `producer.py` script from Part I. This process involves consuming Kafka messages, enriching them with additional metadata from a static dataset (`netflix.csv`), and persisting the resulting enriched data in a Cassandra database.

The `netflix.csv` file contains movie details such as movie title, director, release year, and duration, while the `movies.csv` file, generated by `producer.py`, provides movie ratings from users. The PySpark script processes this data and writes the results to Cassandra in a real-time streaming fashion.

To achiece this goal, the `cassandra-spark-streaming.py` was developed. This script is designed to consume Kafka messages, enrich them with additional metadata from a static movie dataset (`netflix.csv`), and persist the enriched information into a Cassandra database for further analysis. The process ensures real-time processing of movie ratings data, allowing for scalable and efficient querying of the ratings data.

1. **Spark Session Initialization:** A Spark session is created with Cassandra connection configurations, specifying the host and port for the Cassandra cluster. The log level is set to "ERROR" to limit verbosity:

```
spark = SparkSession.builder \
    .appName("KafkaToCassandra") \
    .config("spark.cassandra.connection.host", "172.18.0.3") \
    .config("spark.cassandra.connection.port", "9042") \
    .getOrCreate()

spark.sparkContext.setLogLevel("ERROR")
```

2. **Kafka Schema Definition:** The schema (`rating_schema`) is defined for the incoming Kafka data. It includes fields for `user`, `movie`, `rating`, and `timestamp`:

```
rating_schema = StructType([
    StructField("user", StringType(), True),
    StructField("movie", StringType(), True),
    StructField("rating", IntegerType(), True),
    StructField("timestamp", StringType(), True)
])
```

3. **Kafka Streaming Setup:** The Kafka stream is read from a specific topic (`test`) using Spark's `readStream` API. The Kafka consumer starts reading from the latest offset and is configured to handle data loss gracefully:

```
kafka_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:29092") \
    .option("subscribe", "test") \
    .option("startingOffsets", "latest") \
    .option("failOnDataLoss", "false") \
    .load()
```

4. **Data Parsing:** The Kafka stream data is parsed into a structured DataFrame. The `timestamp` field is converted to a proper timestamp format:

```
parsed_df = kafka_df.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), rating_schema).alias("data")) \
    .select("data.*")

parsed_df = parsed_df.withColumn("rating_timestamp",
    to_timestamp(col("timestamp"), "yyyy-MM-dd'T'HH:mm:ss.SSSSSS"))
```

5. **Column Renaming and Normalization:** The columns `user` and `movie` are renamed to match the Cassandra schema. The `movie_name` field is also normalized:

```
parsed_df = parsed_df.withColumnRenamed("user", "user_name") \
                     .withColumnRenamed("movie", "movie_name")

parsed_df = parsed_df.withColumn("movie_name", trim(lower(col("movie_name"))))
```

6. **Netflix Metadata Schema Definition:** The schema for the Netflix movie metadata (`netflix_schema`) is defined to ensure proper parsing of the CSV file:

```
netflix_schema = StructType([
    StructField("show_id", StringType(), True),
    StructField("title", StringType(), False),
    StructField("director", StringType(), True),
    StructField("country", StringType(), True),
```

```
    StructField("release_year", IntegerType(), True),
    StructField("rating", StringType(), True),
    StructField("duration", StringType(), True)
])
```

7. **Loading Netflix Metadata:** The script reads a static CSV file (`netflix.csv`) containing movie metadata:

```
movies_df = spark.read.schema(netflix_schema) \
    .option("header", True) \
    .csv("/vagrant/data/netflix.csv")
```

   The movie titles are normalized to ensure accurate joins:

```
movies_df = movies_df.withColumn("movie_name", trim(lower(col("title")))) \
                     .withColumnRenamed("rating", "rating_category")
```

8. **Duration Conversion:** The `duration` field, originally a string (e.g., "90 min"), is converted to an integer:

```
movies_df = movies_df.withColumn("duration", col("
    duration").substr(1, 3).cast(IntegerType()))
```

9. **Joining Rating Data with Netflix Metadata:** The parsed rating data is joined with the movie metadata on the `movie_name` field:

```
enriched_df = parsed_df.join(movies_df, "movie_name", "
    left") \
     .select(
          parsed_df.user_name, parsed_df.movie_name,
              parsed_df.rating, parsed_df.rating_timestamp
              ,
          movies_df.show_id, movies_df.director,
              movies_df.country,
          movies_df.release_year, movies_df.
              rating_category, movies_df.duration
     )
```

10. **Filtering Null Values:** Rows where the `user_name` is null are filtered out to ensure only valid data is written to Cassandra:

```
enriched_df = enriched_df.filter(col("user_name").isNotNull())
```

11. **Cassandra Write Function:** A function `writeToCassandra` is defined to handle writing the enriched data to a Cassandra table. The data is written in "append" mode:

```
def writeToCassandra(writeDF, _):
    writeDF.write \
        .format("org.apache.spark.sql.cassandra") \
        .mode("append") \
        .options(table="ratings", keyspace="netflix") \
        .save()
```

12. **Streaming with Retry Logic:** The streaming job is started with a checkpoint location for fault tolerance. If any errors occur, the script catches the exception, logs it, and waits for 5 seconds before retrying:

```
result = None
while result is None:
    try:
        result = enriched_df.writeStream \
            .option("checkpointLocation", "/tmp/checkpoint") \
            .foreachBatch(writeToCassandra) \
            .outputMode("append") \
            .trigger(processingTime='30 seconds') \
            .start()
        result.awaitTermination()
    except Exception as e:
        print("Error:", e)
        traceback.print_exc()
        time.sleep(5)
```

13. **Cassandra Data Model:** The Cassandra table is optimized for storing user ratings, allowing fast queries based on user and timestamp. The `user_name` is selected as `partition key` ensuring that that ratings are grouped by user and `rating_timestamp, movie_name` are the `clustering key` allowing for sorting the ratings by time in descending order and for keeping ratings for each movie distinct under each user's partition. The table schema is defined as follows:

```
CREATE TABLE netflix.ratings (
    user_name text,
    rating_timestamp timestamp,
    movie_name text,
    director text,
    country text,
    release_year int,
    rating_category text,
    duration int,
    show_id text,
    rating int,
    PRIMARY KEY ((user_name), rating_timestamp, movie_name)
) WITH CLUSTERING ORDER BY (rating_timestamp DESC);
```

The next step was the creation of the `console-spark-streaming.py` script, which consumes movie ratings data from Kafka and outputs the results to the console. The significant parts of the script are as follows:

1. **Importing Necessary Libraries:** The script begins by importing essential libraries and initializing the Spark session:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
from pyspark.sql.functions import from_json, col
```

2. **Spark Session Initialization:** The Spark session is created and configured to read from Kafka.

```
spark = SparkSession \
.builder \
.appName("SSKafka") \
.config("spark.jars.packages", "org.apache.spark:spark-
    sql-kafka-0-10_2.12:3.5.0") \
.getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
```

3. **Defining the Schema:** The schema for the movie ratings data is defined to match the structure of the incoming JSON messages.

```
rating_schema = StructType([
    StructField("user", StringType(), True),
```

```
    StructField("movie", StringType(), True),
    StructField("rating", IntegerType(), True),
    StructField("timestamp", StringType(), True)
])
```

4. **Reading Data from Kafka:** The script reads streaming data from Kafka, where it is subscribed to the topic 'test' containing the movie ratings data.

```
 df = spark \
.readStream \
.format("kafka") \
.option("kafka.bootstrap.servers", "localhost:29092") \
.option("subscribe", "test") \
.option("startingOffsets","earliest") \
.load()
```

5. **Processing the Data:** The script then deserializes the Kafka messages from JSON format and extracts the relevant fields into a DataFrame.

```
sdf = df.selectExpr("CAST(value AS STRING)") \
  .select(from_json(col("value"), rating_schema).alias("data"))\
  .select("data.*")
```

6. **Writing to the Console:** The processed data is output to the console for debugging or inspection.

```
sdf.writeStream \
  .outputMode("append") \
  .format("console") \
  .option("truncate", False) \
  .start() \
  .awaitTermination()
```

7. **Verify results of console:** Once the `console-spark-streaming.py` script is created, you can verify that Kafka is receiving messages by running the following command in the Vagrant environment:

```
docker exec -it kafka kafka-console-consumer --topic
   test --bootstrap-server localhost:29092 --from-
   beginning
```

This command reads messages from the Kafka topic `test` starting from the beginning. If the producer is correctly sending messages, you should see output similar to the following sample:

{"user": "Maria", "movie": "Inception", "rating": 9, "timestamp": "2025-03-25T00:00:00"}
{"user": "Maria", "movie": "6 Underground", "rating": 10, "timestamp": "2025-03-24T23:37:44.261746"}
{"user": "David Woods", "movie": "Isoken", "rating": 10, "timestamp": "2025-03-24T23:37:58.269033"}
{"user": "Ashley Brown", "movie": "Todd Glass: Act Happy", "rating": 4, "timestamp": "2025-03-24T23:38:43.305098"}

In order to check the output of the project, the following files were executed simultaneously in different terminals:

- `producer.py`
- `console-spark-streaming.py`
- `cassandra-spark-streaming.py`

In order to run cassandra `cassandra-spark-streaming.py` file, the following command was executed in vagrant environment:

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2
    .12:3.5.0,com.datastax.spark:spark-cassandra-connector_2
    .12:3.0.0 cassandra-spark-streaming.py
```

After that, in order to examine the output of the project, the following commands were executed:

1. `docker exec -it cassandra cqlsh`
2. Creation of the Cassandra table.
3. `DESCRIBE KEYSPACE netflix;`
4. `cqlsh:netflix> SELECT * FROM ratings LIMIT 60;`

The output of the command is displayed in the following table. Each time new data is sent to Cassandra, the table is refreshed to reflect the latest updates.

User Name	Timestamp	Movie Name	Country	Director	Duration	Rating	Category	Year	Show ID
Mackenzie Moreno	2025-03-26 17:55:36	Christmas Survival	UK	James Dearden	101	6	TV-MA	2018	s3333
Christopher Soto	2025-03-26 18:08:29	Get Santa	UK	Christopher Smith	103	5	PG	2014	s6841
Christopher Soto	2025-03-26 17:59:11	Hannah Gadsby: Nanette	Australia	Madeleine Parry, Jon Olb	70	2	TV-MA	2018	s4824
Robert Ochoa	2025-03-26 17:57:32	The BFG	USA	Steven Spielberg	118	5	PG	2016	s1204
Katherine Bird	2025-03-26 18:06:37	Chhota Bheem Aur Kaala Yodha	Not Given	Sidheswar Shukla, Asit Mohapatra	63	8	TV-Y7	2018	s6462
Katherine Bird	2025-03-26 18:04:20	Vincent N Roxxy	USA	Gary Michael Schultz	101	2	R	2016	s8679
Katherine Bird	2025-03-26 18:00:41	Ricky Gervais: Humanity	UK	John L. Spencer	79	1	TV-MA	2018	s4990

Table 1: Netflix Ratings Data