Programming Assignment № 2

Evangelia Panourgia, Athens University of Economics and Business University 27/03/2025

Part I

The python script for the first part.

The provided Python script is an asynchronous Kafka producer designed to simulate real-time movie rating events. It uses the `Faker` library to generate synthetic user names and reads actual movie titles from a CSV file using `pandas`. Each generated user rates a randomly selected movie with a random rating between 1 and 10, along with a current timestamp. These events are serialized to JSON and sent to a Kafka topic named `"test"` using the `aiokafka` library. The process runs continuously, with each user producing one rating per minute, making it ideal for streaming applications such as those built with Apache Spark Structured Streaming. Additionally, the script appends a specific user name, “Evangelia Panourgia”, to ensure consistent personal data inclusion for testing or filtering.

Listing 1: python-kafka-example

```
"""
Async Kafka Producer: Fake Movie Ratings Generator

Overview:
---------
This script continuously generates **fake movie rating events** and sends them to a Kafka topic ("test").
It simulates users rating random movies with timestamps, mimicking real-time data.

Components:
-----------
1. Faker: Generates fake human names.
2. Pandas: Loads real movie titles from a CSV file.
3. aiokafka: Asynchronous Kafka producer library.
4. Kafka: Acts as the message broker for event streaming.

Use Case:
---------
Run this script to simulate live rating data for a Spark Structured Streaming application to consume and process.
"""
```

```python
import json
import asyncio
import random
from datetime import datetime

import pandas as pd
from faker import Faker
from aiokafka import AIOKafkaProducer

# Load movie titles
movies_df = pd.read_csv('data/movies.csv', header=None, names=["movie_title"])
movie_titles = movies_df['movie_title'].dropna().tolist()

# Generate names
fake = Faker()
names = [fake.name() for _ in range(10)]
names.append("Evangelia Panourgia") # add my own name

# Kafka settings
KAFKA_TOPIC = "test"
KAFKA_BOOTSTRAP_SERVERS = "localhost:29092"

# JSON serializer
def serializer(data):
    return json.dumps(data).encode()

# Producer logic
async def produce():
    producer = AIOKafkaProducer(
        bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
        value_serializer=serializer,
        compression_type="gzip"
    )

    await producer.start()
    try:
        while True:
            for name in names:
                movie = random.choice(movie_titles)
                rating = random.randint(1, 10)
                timestamp = datetime.now().isoformat()

                message = {
                    "name": name,
                    "movie": movie,
                    "timestamp": timestamp,
```

```
                    "rating": rating
                }

                print(f"Sending: {message}")
                await producer.send(KAFKA_TOPIC, message)
                await asyncio.sleep(60) # One message per user per minute
    finally:
        await producer.stop()

# Run it (no deprecation warning)
if __name__ == "__main__":
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(produce())
```

The image below depicts the output of kafka prodicer runing in the first terminal.

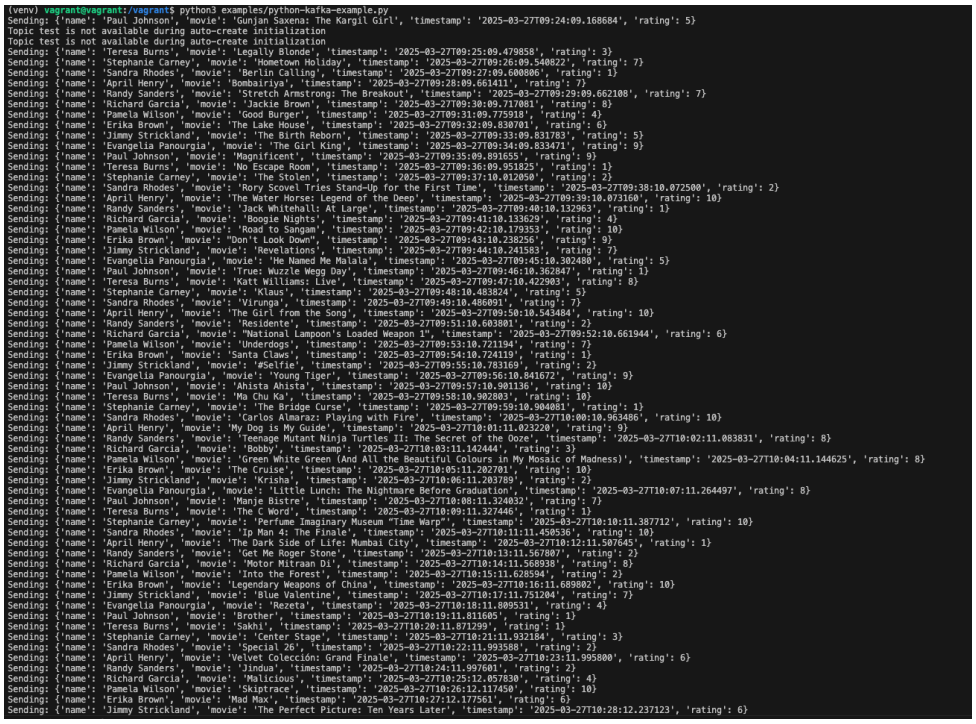

Figure 1: Terminal 1: Kafka Producer.

An additional modification was applied to the console script located in the `examples` directory, specifically to the file named `console-spark-streaming-example.py`. This adjustment was necessary to tailor the script to the project's requirements and ensure compatibility with the streaming architecture.

The image below depicts the console output helping us to debug the kafka producer real-time data. It runs in another terminal.

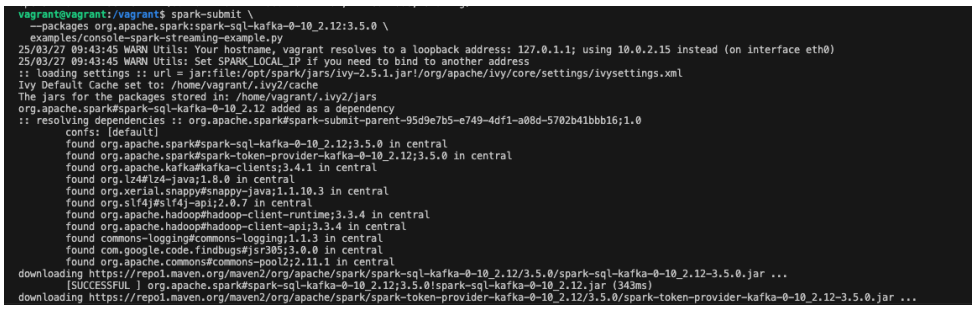

Figure 2: Terminal 2: Console for Kafka Producer.

Part 2

The pyspark script for the second part.

This PySpark application establishes a real-time data pipeline that reads movie rating events from a Kafka topic, enriches them with metadata from a static Netflix dataset, and writes the processed results into a Cassandra database. The script defines two schemas: one for parsing JSON-encoded rating messages received from Kafka, and another for reading static metadata from a CSV file. It converts timestamps to proper `TimestampType`, and groups data by hourly buckets for time-based aggregations. Ratings are joined with metadata on the movie title to create an enriched DataFrame, which is then written to Cassandra using the `foreachBatch` method in micro-batch mode. This architecture enables structured streaming analysis and time-windowed queries for individual users and movies.

To improve performance and avoid repeated disk reads, the static Netflix metadata is cached in memory, ensuring it remains efficiently accessible during each micro-batch operation within the streaming query.

Listing 2: cassandra-spark-streaming-example.py.

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.types import StructType, StructField, StringType, ↩
       IntegerType, TimestampType
3  from pyspark.sql.functions import from_json, col, to_timestamp, ↩
       date_format
4
5  # Schema for Kafka messages (movie ratings)
6  ratingSchema = StructType([
7      StructField("name", StringType(), False),
8      StructField("movie", StringType(), False),
9      StructField("timestamp", StringType(), False),
10     StructField("rating", IntegerType(), False)
11 ])
```

```
12
13 # Schema for Netflix CSV
14 netflixSchema = StructType([
15     StructField("show_id", StringType(), True),
16     StructField("title", StringType(), False),
17     StructField("director", StringType(), True),
18     StructField("country", StringType(), True),
19     StructField("release_year", StringType(), True),
20     StructField("rating", StringType(), True),
21     StructField("duration", StringType(), True)
22 ])
23
24 # Initialize Spark session
25 spark = (
26     SparkSession.builder
27     .appName("MovieRatingStreamer")
28     .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10↩
        _2.12:3.5.0,com.datastax.spark:spark-cassandra-connector_2↩
        .12:3.4.1")
29     .config("spark.cassandra.connection.host", "localhost")
30     .getOrCreate()
31 )
32
33 spark.sparkContext.setLogLevel("ERROR")
34
35 # Read Netflix CSV with renamed rating column to avoid conflict with "↩
    rating" in Kafka schema
36 netflix_df = (
37     spark.read.schema(netflixSchema)
38          .option("header", True)
39          .csv("data/netflix.csv")
40          .withColumnRenamed("rating", "rating_category")  # Rename avoids ↩
             conflict
41          .cache()
42 )
43
44 # Read streaming data from Kafka
45 df = (
46     spark.readStream.format("kafka")
47          .option("kafka.bootstrap.servers", "localhost:29092")
48          .option("subscribe", "test")
49          .option("startingOffsets", "latest")
50          .load()
51 )
52
53 # Parse JSON from Kafka messages
54 ratings_df = (
```

```
    df.selectExpr("CAST(value AS STRING)")
      .select(from_json(col("value"), ratingSchema).alias("data"))
      .select("data.*")
      .withColumn("timestamp", to_timestamp("timestamp"))  # Convert string timestamp to TimestampType
      .withColumn("hour_bucket", date_format("timestamp", "yyyy-MM-dd HH:00"))  # Grouping by hour
)

# Join ratings with static Netflix metadata
enriched_df = (
    ratings_df.join(netflix_df, ratings_df.movie == netflix_df.title, "left")
              .drop("title")
)

# Optional: Print schema for debugging (can be removed later)
enriched_df.printSchema()

# Define Cassandra write logic
def writeToCassandra(writeDF, _):
    (
        writeDF.select(
            "name", "movie", "timestamp", "rating", "hour_bucket",
            "show_id", "director", "country", "release_year", "rating_category", "duration"
        )
        .write
        .format("org.apache.spark.sql.cassandra")
        .mode('append')
        .options(table="movie_ratings", keyspace="netflix_ks")
        .save()
    )

# Write to Cassandra in a streaming loop
result = None
while result is None:
    try:
        result = (
            enriched_df.writeStream
                .foreachBatch(writeToCassandra)
                .outputMode("update")
                .option("checkpointLocation", "/tmp/checkpoints/movie")
                .trigger(processingTime="30 seconds")
                .start()
                .awaitTermination()
        )
```

```
98      except Exception as e:
99          print(f"Streaming error: {e}")
```

Details about your Cassandra data model.

The Cassandra data model was carefully designed to optimize query performance for the specific requirement of aggregating movie ratings by user and hour. The primary key is composed of a composite partition key `(name, hour_bucket)`, ensuring that all ratings provided by a specific user within a given hour are stored in the same partition. This design eliminates the need for inefficient filtering operations and allows for fast, targeted queries such as retrieving all movies rated by a user during a specific hour or calculating the average rating and duration of those movies. The clustering columns `(timestamp, movie)` further organize the data chronologically within each partition, enabling efficient range queries and ordered retrieval. Crucially, the fields `rating` and `duration` are stored as integers, allowing native support for aggregation functions like `AVG()`, which would not be possible with text fields. Overall, this schema supports the core analytical tasks required by the application while adhering to Cassandra's best practices for scalability and performance.

Listing 3: Cassandra schema for movie ratings

```
1  CREATE KEYSPACE IF NOT EXISTS netflix_ks
2  WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
3
4  CREATE TABLE IF NOT EXISTS netflix_ks.movie_ratings (
5      name text,
6      hour_bucket text,
7      timestamp timestamp,
8      movie text,
9      rating int,
10     show_id text,
11     director text,
12     country text,
13     release_year int,          -- changed from text : int
14     rating_category text,
15     duration int,              -- changed from text : int
16     PRIMARY KEY ((name, hour_bucket), timestamp, movie)
17 );
```

A sample of persisted lines (around 50) of your Cassandra table.

To inspect the data stored in the `movie_ratings` table of the `netflix_ks` keyspace, we can use the `cqlsh` shell interface provided by Apache Cassandra. The following query retrieves

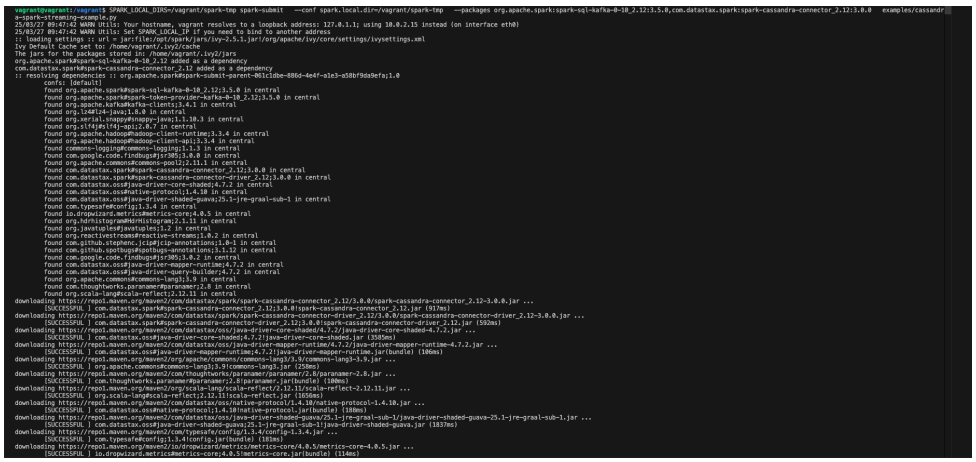

Figure 3: Terminal 3: Spark pre-process, enrich with metadata..

approximately 50 rows that were previously persisted by the streaming pipeline:

```
SELECT * FROM netflix_ks.movie_ratings LIMIT 50;
```

This query provides a representative snapshot of the data ingested into Cassandra, including key fields such as `name`, `movie`, `timestamp`, `rating`, and `duration`. Analyzing these records ensures data integrity, validates the correctness of the ingestion pipeline, and confirms that the schema design supports the desired querying capabilities.

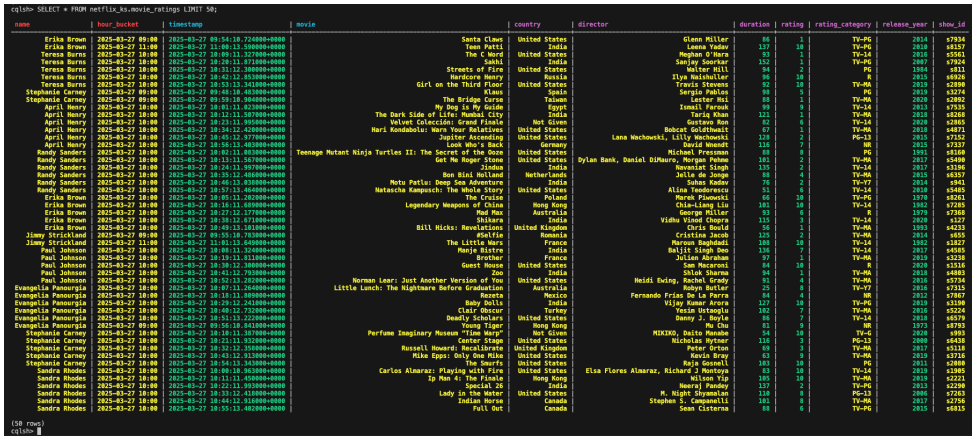

Figure 4: A sample output of the Cassandra table with 50 persisted entries.

Two CQL queries and their results in your database about your own name and a particular hour that generate the average runtime of the movies that you've rated during this hour, and the names of the movies, respectively.

To support time-based analytics in our Cassandra data model, we designed the schema to partition data by `name` and `hour_bucket`, allowing efficient access to all ratings a specific user made during a given hour. The first query retrieves the titles of movies rated by the user *Evangelia Panourgia* during the hour of `2025-03-27 09:00` using a direct lookup on the partition key. The second query, which includes the `ALLOW FILTERING` clause, calculates the average

duration of those rated movies within the same hour. While Cassandra does not natively support aggregations without filtering, our schema minimizes performance issues by narrowing queries to a specific partition, ensuring practical execution even with the filtering clause.

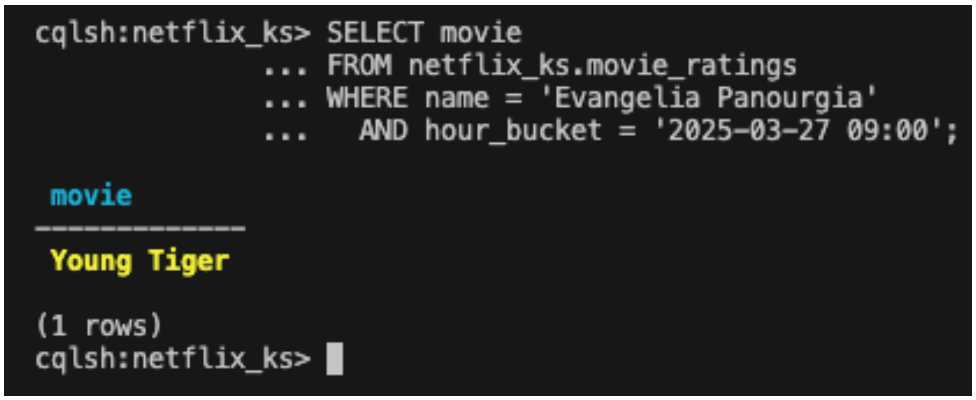

Figure 5: Names of the movies you rated during a particular hour SELECT movie.

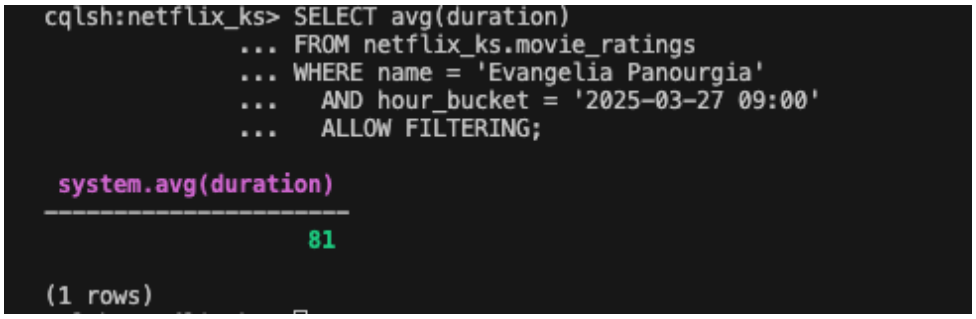

Figure 6: Average duration of the movies you rated during the same hour SELECT avg(duration).

Furthermore, after allowing the real-time streaming process to run for an additional hour, we re-executed the following queries—this time updating the hour_bucket value from 09:00 to 10:00 to reflect the new time window.

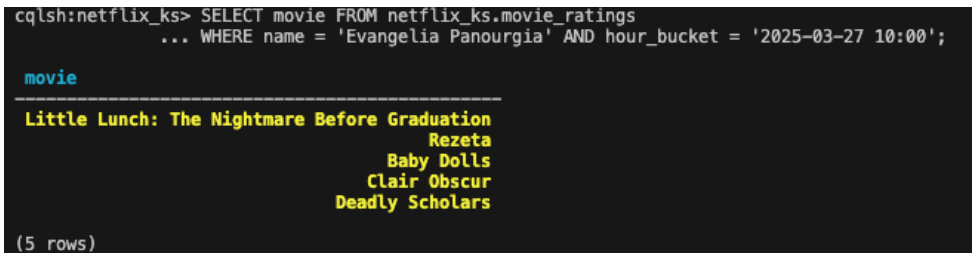

Figure 7: Names of the movies you rated during a particular hour SELECT movie.

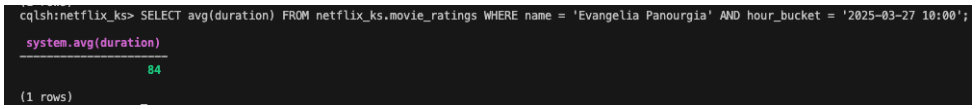

Figure 8: Average duration of the movies you rated during the same hour SELECT avg(duration).

Note: It is not necessary to explicitly include conditions such as IS NOT NULL in the CQL

queries, as Cassandra automatically excludes null values from aggregation functions like AVG(). Therefore, rows with missing values for the targeted column are implicitly ignored during computation.

Hosted on GitHub Repository - Deploy Instructions.

The complete implementation of this project is hosted on GitHub and is publicly accessible at `https://github.com/e-panourgia/large-data-kafka-cassandra`. The repository contains all necessary components, including Kafka producers, Spark Structured Streaming jobs, and Cassandra schemas. To execute the project, clone the repository using `git clone https://github.com/e-panourgia/large-data-kafka-cassandra`, then follow the instructions in the README file to set up Docker containers, start Kafka and Cassandra services, and run the streaming application using `spark-submit`. Ensure that you have installed Docker, Docker Compose, and Spark locally or are running within the provided Vagrant virtual machine for a consistent and reproducible development environment.

Important note, this implementation was designed to run on mac m4 pc.

Open source software (OSS): Contribution to Faker

As part of the Applied Software Engineering course during my BSc studies, I had the opportunity to contribute to the open-source project `Faker`, a Python library widely used for generating synthetic data. My contribution focused on enhancing the tag-handling functionality, which plays a critical role in structured data generation. Specifically, I improved the way tags are processed within data pipelines, enabling more accurate and flexible population of fields with fake data. This work was incorporated through a merged pull request on GitHub, addressing limitations in tag parsing and significantly improving Faker's usability in scenarios involving automated testing and data anonymization. This contribution was relevant, as Faker was used in our project to auto-generate event data for a producer component.

Acknowledgements

I would like to sincerely thank **Professor Liakos Panayiotis** for his invaluable guidance, support, and dedication throughout the entire teaching process. His insightful lectures and continuous encouragement greatly contributed to the successful completion of this project.