Assignment 4

Shao-Yu Huang 018195554

Q1. Design a Kafka streaming data processing pipeline using the NYC Taxi Trip dataset (https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page). Review the NYC taxi trip data to understand its structures and contents. Answer the following questions about the Kafka streaming data processing pipeline.

1. **Pipeline Design: Plan a Kafka pipeline to stream and process the data. You MUST include a data processing step in this stream data processing pipeline using Kafka, meaning you cannot directly output the raw data from the consumer. Write a short summary explaining your design this Kafka streaming data processing pipeline in details and how this pipeline could be used in real-world applications.**

   Dataset: 2025(January) Yellow Taxi Trip Records

   Title: Calculates the average fare every 30 minutes

   Data Pipeline detail:

   - Producer: Reads from PARQUET file and sends each taxi trip record to the Kafka Topic "taxi_trip_data," it also will do some data cleaning, such as drop rows that the pickup time is later than the drop-off time and the fare amount that is below or equal to 0. Each record contains fare_amount, pickup_datetime, dropoff_datetime, and passenger_count.
   - Kafka Topic "taxi_trip_data": This raw data topic stores all the incoming taxi trip data.
   - Consumer: Consume data from Topic, filter it for necessary attributes, and compute the rolling average fare for each 30-minute window. The result (average fare) is then printed to the console.
   - Output: The average fare per 30 minutes is either printed in real-time or sent to another topic for monitoring or storing results in a database for future analysis.

   This pipeline enables real-time monitoring of taxi fares, helping businesses track pricing trends, monitor demand, and optimize the fleet. It could also be used to analyze fare anomalies or sudden spikes in pricing, allowing for quick responses or optimizations.

2. **Implementation: Use Python and Kafka to implement your pipeline. Submit ALL the Python scripts you developed.**

   - Producer.py:

```python
from kafka import KafkaProducer
import pandas as pd
import pyarrow.parquet as pq
import json
import time

# Read and clean the data
yellow_trips = pq.read_table('yellow_tripdata_2025-01.parquet')
yellow_trips = yellow_trips.to_pandas()
```

```

# Data cleaning process
yellow_trips = yellow_trips[yellow_trips['tpep_pickup_datetime'] <=
yellow_trips['tpep_dropoff_datetime']]
yellow_trips = yellow_trips[yellow_trips['fare_amount'] > 0]

# Kafka Producer Setup
producer = KafkaProducer(
bootstrap_servers=['localhost:9092'], # Kafka broker address
value_serializer=lambda v: json.dumps(v).encode('utf-8') # Serialize records as JSON
)

# Produce data to Kafka topic
for index, record in yellow_trips.iterrows():
message = {
'fare_amount': record['fare_amount'],
'pickup_datetime': record['tpep_pickup_datetime'].strftime('%Y-%m-%d %H:%M:%S'),
'dropoff_datetime': record['tpep_dropoff_datetime'].strftime('%Y-%m-%d %H:%M:%S'),
'passenger_count': record['passenger_count']
}

# Send message to Kafka topic "taxi_trip_data"
producer.send('taxi_trip_data', value=message)

# Print to verify
print(f"Sent: {message}")

# Add a delay between sending records (optional, to simulate real-time processing)
time.sleep(0.1)

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

- Consumer.py:

```
from kafka import KafkaConsumer
import json
from collections import deque
from datetime import datetime

# Kafka Consumer Setup
consumer = KafkaConsumer(
'taxi_trip_data', # Kafka topic name
bootstrap_servers=['localhost:9092'],
value_deserializer=lambda m: json.loads(m.decode('utf-8')),
group_id='taxi_group'
)

# Initialize a deque to store data for calculating average fare within a 30-minute window
fare_window = deque()
window_duration = 30 * 60 # 30 minutes in seconds

def calculate_average_fare(window_data):
if not window_data:
return 0
```

```python
total_fare = sum([record[1] for record in window_data]) # record[1] is fare_amount
return total_fare / len(window_data)

# Consumer behavior: Processing the streamed data from Kafka
for message in consumer:
record = message.value
fare_amount = record['fare_amount']
pickup_datetime = datetime.strptime(record['pickup_datetime'], '%Y-%m-%d %H:%M:%S')

# Add new record to the fare_window
fare_window.append((pickup_datetime, fare_amount))

# Remove records that are older than the 30-minute window
while fare_window and (pickup_datetime - fare_window[0][0]).total_seconds() >
window_duration:
fare_window.popleft()

# Calculate and display the average fare for the current window
avg_fare = calculate_average_fare(fare_window)
print(f"Current Pickup Time: {pickup_datetime} | Average fare for the last 30 minutes:
${avg_fare:.2f}")
```

3. **Results: Use Zoom or Panopto to record a video of your screen demonstrating your implemented Kafka streaming data processing pipeline, like the class demo. Share a public link to your video. No need to upload the video file. Make sure the link works and others can view it.**
   Link: https://youtu.be/KowdLCOrdOg

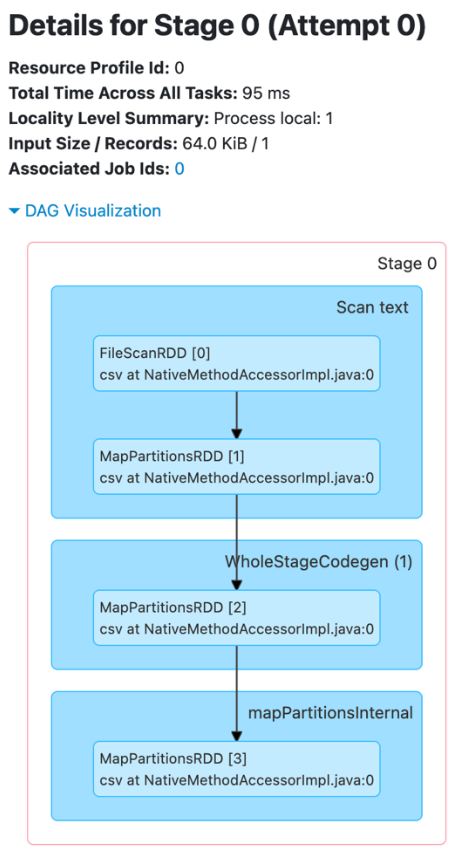
Q2. For this task, find an open-source dataset in CSV format and design a data processing task using the MapReduce framework. Implement this MapReduce job in PySpark in local environment.

1. **Submit the ALL python scripts you have developed. Share the link of your dataset.**
   Dataset Link (I only use title.crew.tsv.gz and name.basics.tsv.gz datasets):
   https://developer.imdb.com/non-commercial-datasets/

```python
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split, col
import os
import requests
import gzip
import shutil
spark = SparkSession.builder \
.config("spark.driver.host", "localhost") \
.config("spark.driver.memory", "4g") \
.config("spark.executor.memory", "4g") \
.appName("mr_prac") \
.getOrCreate()
spark

def download_and_unzip(url, output_dir="."):
```

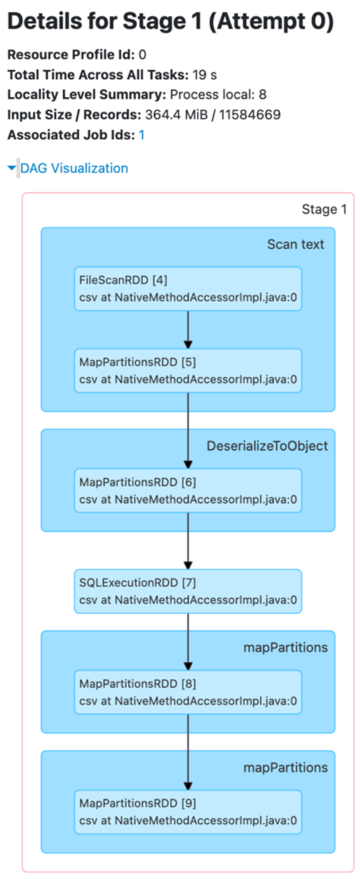
```
filename = url.split("/")[-1]
local_gz_path = os.path.join(output_dir, filename)
local_tsv_path = local_gz_path.replace(".gz", "")

# Skip download if already exists
if not os.path.exists(local_gz_path):
print(f"Downloading {filename}...")
response = requests.get(url, stream=True)
with open(local_gz_path, 'wb') as f:
shutil.copyfileobj(response.raw, f)
print(f"Downloaded: {local_gz_path}")
else:
print(f"File already exists: {local_gz_path}")

# Unzip .gz to .tsv
if not os.path.exists(local_tsv_path):
print(f"Unzipping {filename}...")
with gzip.open(local_gz_path, 'rb') as f_in:
with open(local_tsv_path, 'wb') as f_out:
shutil.copyfileobj(f_in, f_out)
print(f"Unzipped to: {local_tsv_path}")
else:
print(f"TSV already exists: {local_tsv_path}")

return local_tsv_path

# IMDb dataset URLs
urls = [
"https://datasets.imdbws.com/title.crew.tsv.gz",
"https://datasets.imdbws.com/name.basics.tsv.gz"
]

# Download and unzip all
for url in urls:
download_and_unzip(url)


# Load IMDb crew data
crew_df = spark.read.csv("title.crew.tsv", sep="\t", header=True, inferSchema=True,
nullValue="\\N")

# Load name.basics for mapping nconst -> primaryName
names_df = spark.read.csv("name.basics.tsv", sep="\t", header=True, inferSchema=True,
nullValue="\\N")

# Clean and extract directors (some titles have multiple directors)
crew_directors = crew_df.select("tconst", explode(split(col("directors"),
",")).alias("director_id")).na.drop()

# Count number of titles per director
director_counts = crew_directors.groupBy("director_id").count().alias("title_count")

# Join with names to get actual director names
director_with_names = director_counts.join(
names_df.select("nconst", "primaryName"),
director_counts["director_id"] == names_df["nconst"],
"inner"
).select("primaryName", "count").orderBy(col("count").desc())

# Show top 20 directors
director_with_names.show(20, truncate=False)

sc = spark.sparkContext
print(sc.statusTracker().getJobIdsForGroup(None))

# stopping the Spark session
spark.stop()
```

2. Provide a screenshot showing the DAG visualization for this MapReduce job.


Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 95 ms
Locality Level Summary: Process local: 1
Input Size / Records: 64.0 KiB / 1
Associated Job Ids: 0

DAG Visualization
Stage 0
Scan text
FileScanRDD [0]
csv at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [1]
csv at NativeMethodAccessorImpl.java:0
WholeStageCodegen (1)
MapPartitionsRDD [2]
csv at NativeMethodAccessorImpl.java:0
mapPartitionsInternal
MapPartitionsRDD [3]
csv at NativeMethodAccessorImpl.java:0


Details for Stage 1 (Attempt 0)

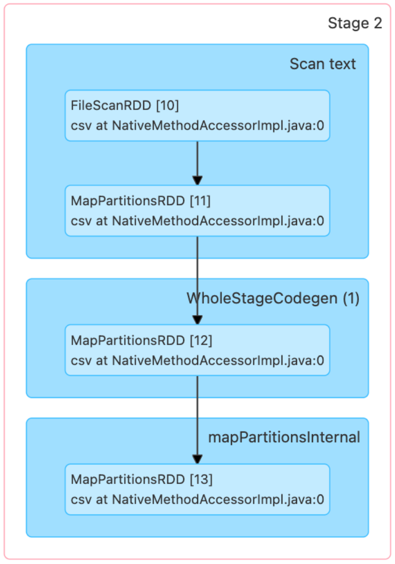
Resource Profile Id: 0
Total Time Across All Tasks: 19 s
Locality Level Summary: Process local: 8
Input Size / Records: 364.4 MiB / 11584669
Associated Job Ids: 1

DAG Visualization
Stage 1
Scan text
FileScanRDD [4]
csv at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [5]
csv at NativeMethodAccessorImpl.java:0
DeserializeToObject
MapPartitionsRDD [6]
csv at NativeMethodAccessorImpl.java:0
SQLExecutionRDD [7]
csv at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [8]
csv at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [9]
csv at NativeMethodAccessorImpl.java:0

Details for Stage 2 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 7 ms
Locality Level Summary: Process local: 1
Input Size / Records: 64.0 KiB / 1
Associated Job Ids: 2

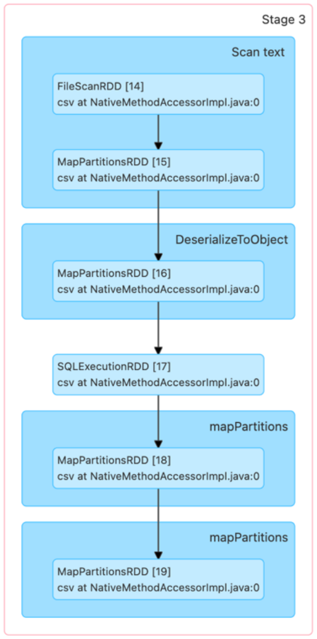
▾ DAG Visualization


Stage 2
Scan text
FileScanRDD [10]
csv at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [11]
csv at NativeMethodAccessorImpl.java:0
WholeStageCodegen (1)
MapPartitionsRDD [12]
csv at NativeMethodAccessorImpl.java:0
mapPartitionsInternal
MapPartitionsRDD [13]
csv at NativeMethodAccessorImpl.java:0

Details for Stage 3 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 24 s
Locality Level Summary: Process local: 8
Input Size / Records: 840.4 MiB / 14325454
Associated Job Ids: 3

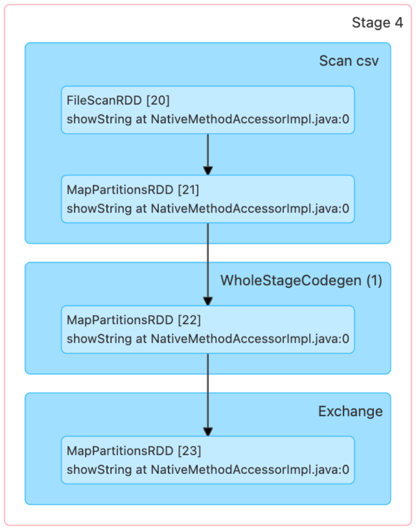
▾ DAG Visualization


Stage 3
Scan text
FileScanRDD [14]
csv at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [15]
csv at NativeMethodAccessorImpl.java:0
DeserializeToObject
MapPartitionsRDD [16]
csv at NativeMethodAccessorImpl.java:0
SQLExecutionRDD [17]
csv at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [18]
csv at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [19]
csv at NativeMethodAccessorImpl.java:0

Details for Stage 4 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 34 s
Locality Level Summary: Process local: 8
Input Size / Records: 364.4 MiB / 11584668
Shuffle Write Size / Records: 18.5 MiB / 1542375
Associated Job Ids: 4

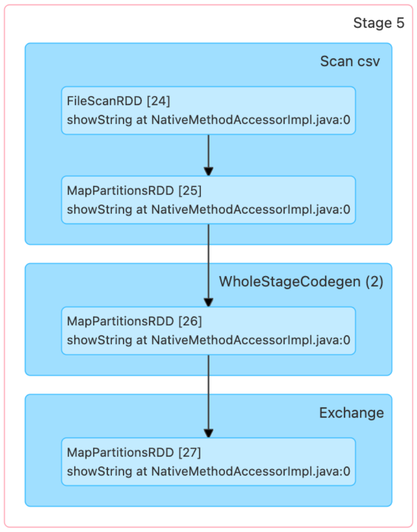
▾ DAG Visualization

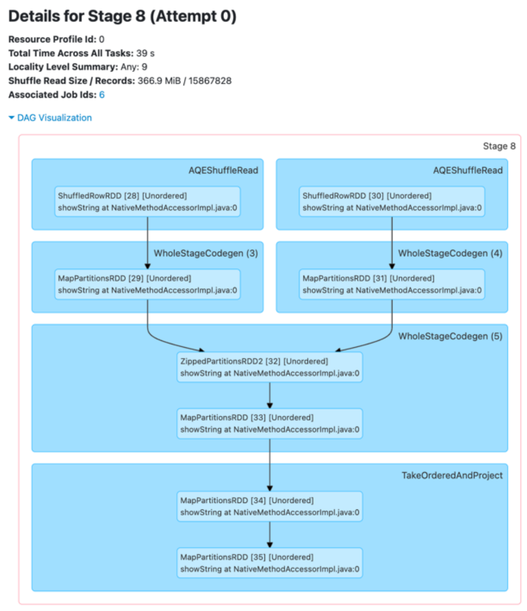

Stage 4
Scan csv
FileScanRDD [20]
showString at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [21]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (1)
MapPartitionsRDD [22]
showString at NativeMethodAccessorImpl.java:0
Exchange
MapPartitionsRDD [23]
showString at NativeMethodAccessorImpl.java:0

Details for Stage 5 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 1.2 min
Locality Level Summary: Process local: 8
Input Size / Records: 840.4 MiB / 14325453
Shuffle Write Size / Records: 348.4 MiB / 14325453
Associated Job Ids: 5

▾ DAG Visualization


Stage 5
Scan csv
FileScanRDD [24]
showString at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [25]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (2)
MapPartitionsRDD [26]
showString at NativeMethodAccessorImpl.java:0
Exchange
MapPartitionsRDD [27]
showString at NativeMethodAccessorImpl.java:0



Details for Stage 8 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 39 s
Locality Level Summary: Any: 9
Shuffle Read Size / Records: 366.9 MiB / 15867828
Associated Job Ids: 6

DAG Visualization
Stage 8
AQEShuffleRead
ShuffledRowRDD [28] [Unordered]
showString at NativeMethodAccessorImpl.java:0
AQEShuffleRead
ShuffledRowRDD [30] [Unordered]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (3)
MapPartitionsRDD [29] [Unordered]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (4)
MapPartitionsRDD [31] [Unordered]
showString at NativeMethodAccessorImpl.java:0
WholeStageCodegen (5)
ZippedPartitionsRDD2 [32] [Unordered]
showString at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [33] [Unordered]
showString at NativeMethodAccessorImpl.java:0
TakeOrderedAndProject
MapPartitionsRDD [34] [Unordered]
showString at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [35] [Unordered]
showString at NativeMethodAccessorImpl.java:0

3. **Provide a summary describing the dataset and the data processing task. Explain how the MapReduce framework was applied to solve the problem.**
   Dataset:
   - "title.crew.tsv": Contains director, writer, and tconst the unique identifier of the title.
   - "name.basics.tsv": Contains metadata for people in the database, including nconst(the unique idnetifier of the name or person), birthYear, deathYear, knownForTitles(titles the person is known for), and primary profession(the top 3 professions of the person).

   The primary goal of the data processing task is to count the number of titles each director has worked on.
   - Map Phase: Each row in the title.crew.tsv file is processed to extract the director IDs. Since titles may have multiple directors (comma-separated), the row is split, and for each director, a tuple (director_id, 1) is emitted — representing one contribution to a title.
   - Shuffle & Sort Phase: Spark automatically groups the intermediate key-value pairs by director_id, preparing them for aggregation.
   - Reduce Phase: The grouped data is reduced by summing the counts, resulting in (director_id, total_titles) pairs.

   To make the output interpretable, the result is joined with name.basics.tsv to map each director_id (nconst) to their primaryName. The final output is a list of director names with the number of titles they have worked on, sorted by count in descending order.

   This MapReduce-based PySpark job efficiently processed thousands of records to output a list of the most active directors in the IMDb database. The highly scalable approach can easily be extended to support more complex queries.