# Simple pyspark program

from pyspark.sql import SparkSession

from pyspark.sql.types import \*

# Create a SparkSession

spark = SparkSession.builder.appName("LoadExcel").getOrCreate()

# Define the schema of your Excel data (adjust as needed)

schema = StructType([

StructField("column1", StringType(), True), # Example column, replace with your actual column names and types

StructField("column2", IntegerType(), True),

# ... more columns

])

# Read the Excel file (replace "your\_file.xlsx" and "Sheet1" with your actual file and sheet name)

df = spark.read.format("com.crealytics.spark.excel") \

.option("header", "true") \

.option("inferSchema", "false") \

.schema(schema) \

.load("your\_file.xlsx", sheetName="Sheet1")

# Display the DataFrame

df.show()

# Stop the SparkSession

spark.stop()

# Scenario 1: Log Analysis

Question: You have a large dataset of server logs stored as text files. Each log entry contains a timestamp, server ID, and error message. Write a PySpark script to find the top 5 most common error messages across all servers.

Answer:

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, split

# Create a SparkSession

spark = SparkSession.builder.appName("LogAnalysis").getOrCreate()

# Read the log files

logs = spark.read.text("path/to/log/files")

# Split the log entries into columns

parsed\_logs = logs.select(

split(col("value"), "\t").getItem(0).alias("timestamp"),

split(col("value"), "\t").getItem(1).alias("server\_id"),

split(col("value"), "\t").getItem(2).alias("error\_message")

)

# Count occurrences of each error message

error\_counts = parsed\_logs.groupBy("error\_message").count()

# Find top 5 most common errors

top\_errors = error\_counts.orderBy(col("count").desc()).limit(5)

# Show results

top\_errors.show(truncate=False)

# Stop the SparkSession

spark.stop()

# Scenario 2: E-commerce Data Analysis

Question: You have two DataFrames: orders (order\_id, customer\_id, order\_date, total\_amount) and customers (customer\_id, name, email, registration\_date). Write a PySpark script to find the top 10 customers by total order amount in the last month.

Answer:

from pyspark.sql import SparkSession

from pyspark.sql.functions import sum, col, datediff, current\_date, lit, rand, round

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, DateType, DoubleType

from datetime import datetime, timedelta

import random

# Create a SparkSession

spark = SparkSession.builder.appName("CustomerAnalysis").getOrCreate()

# Define schemas for orders and customers DataFrames

**orders\_schema** = StructType([

StructField("order\_id", IntegerType(), False),

StructField("customer\_id", IntegerType(), False),

StructField("order\_date", DateType(), False),

StructField("total\_amount", DoubleType(), False)

])

**customers**\_**schema** = StructType([

StructField("customer\_id", IntegerType(), False),

StructField("name", StringType(), False),

StructField("email", StringType(), False),

StructField("registration\_date", DateType(), False)

])

# Generate sample data

def generate\_sample\_data(num\_customers, num\_orders):

customers\_data = []

orders\_data = []

**for i in range(1, num\_customers** + 1):

customers\_data.append((

i,

f"Customer {i}",

f"customer{i}@example.com",

datetime(2023, 1, 1) - timedelta(days=random.randint(0, 365))

))

**for i in range(1, num**\_orders + 1):

orders\_data.append((

i,

random.randint(1, num\_customers),

datetime(2023, 6, 1) - timedelta(days=random.randint(0, 60)),

round(random.uniform(10, 1000), 2)

))

return customers\_data, orders\_data

# Create sample DataFrames

customers\_data, orders\_data = generate\_sample\_data(100, 1000)

customers = spark.createDataFrame(customers\_data, customers\_schema)

orders = spark.createDataFrame(orders\_data, orders\_schema)

# Show sample of original DataFrames

print("Sample of Customers DataFrame:")

customers.show(5)

print("Sample of Orders DataFrame:")

orders.show(5)

# Filter orders for the last month

recent\_orders = orders.filter(datediff(current\_date(), col("order\_date")) <= 30)

# Join orders with customers and calculate total amount per customer

customer\_totals = recent\_orders.join(customers, "customer\_id") \

.groupBy("customer\_id", "name") \

.agg(round(sum("total\_amount"), 2).alias("total\_spent"))

# Find top 10 customers

top\_customers = customer\_totals.orderBy(col("total\_spent").desc()).limit(10)

# Show results

print("Top 10 Customers by Total Spent in the Last Month:")

top\_customers.show()

# Additional analysis: Average order amount per customer

avg\_order\_amount = orders.groupBy("customer\_id") \

.agg(round(avg("total\_amount"), 2).alias("avg\_order\_amount"))

print("Average Order Amount per Customer:")

avg\_order\_amount.join(customers, "customer\_id") \

.select("customer\_id", "name", "avg\_order\_amount") \

.orderBy(col("avg\_order\_amount").desc()) \

.show(10)

# Stop the SparkSession

# spark.stop()Scenario 3: Time Series Analysis

Question: You have a large DataFrame of sensor readings with columns (sensor\_id, timestamp, temperature, humidity). Write a PySpark script to calculate the moving average of temperature for each sensor over the last 5 readings.

Answer:

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, avg, to\_timestamp, round

from pyspark.sql.window import Window

from pyspark.sql.types import StructType, StructField, StringType, DoubleType, TimestampType

import random

from datetime import datetime, timedelta

# Create a SparkSession

spark = SparkSession.builder.appName("SensorAnalysis").getOrCreate()

# Define the schema for the sensor\_data DataFrame

schema = StructType([

StructField("sensor\_id", StringType(), False),

StructField("timestamp", TimestampType(), False),

StructField("temperature", DoubleType(), False),

StructField("humidity", DoubleType(), False)

])

# Generate sample sensor data

def generate\_sensor\_data(num\_sensors, readings\_per\_sensor):

data = []

for sensor in range(1, num\_sensors + 1):

start\_time = datetime(2023, 1, 1, 0, 0, 0)

for \_ in range(readings\_per\_sensor):

timestamp = start\_time + timedelta(minutes=random.randint(1, 60))

temperature = round(random.uniform(20.0, 30.0), 2)

humidity = round(random.uniform(30.0, 70.0), 2)

data.append((f"sensor\_{sensor}", timestamp, temperature, humidity))

start\_time = timestamp

return data

# Create the sensor\_data DataFrame

sensor\_data = spark.createDataFrame(generate\_sensor\_data(3, 20), schema)

# Show the original DataFrame

print("Original Sensor Data:")

sensor\_data.orderBy("sensor\_id", "timestamp").show(10)

# Define the window specification

window\_spec = Window.partitionBy("sensor\_id") \

.orderBy("timestamp") \

.rowsBetween(-4, 0)

# Calculate moving average

moving\_avg = sensor\_data.withColumn("moving\_avg\_temp",

round(avg("temperature").over(window\_spec), 2)

)

# Show results

print("Sensor Data with Moving Average Temperature:")

moving\_avg.select("sensor\_id", "timestamp", "temperature", "moving\_avg\_temp") \

.orderBy("sensor\_id", "timestamp") \

.show(15)

# Calculate overall average temperature for each sensor

overall\_avg = sensor\_data.groupBy("sensor\_id") \

.agg(round(avg("temperature"), 2).alias("overall\_avg\_temp"))

print("Overall Average Temperature by Sensor:")

overall\_avg.show()

# Stop the SparkSession

spark.stop()

# Scenario 4: Data Cleaning and Transformation

Question: You have a DataFrame of product data with columns (product\_id, name, price, category). Some price values are missing, and you want to fill them with the median price of their respective category. Write a PySpark script to accomplish this.

Answer:

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, median, when, lit, rand

from pyspark.sql.types import StructType, StructField, StringType, DoubleType, IntegerType

# Create a SparkSession

spark = SparkSession.builder.appName("DataCleaning").getOrCreate()

# Define the schema for the products DataFrame

**schema** = StructType([

StructField("product\_id", IntegerType(), False),

StructField("name", StringType(), False),

StructField("price", DoubleType(), True), # Allow null values

StructField("category", StringType(), False)

])

# Create a sample products DataFrame

**data** = [

(1, "Laptop", 1200.0, "Electronics"),

(2, "Smartphone", 800.0, "Electronics"),

(3, "Headphones", None, "Electronics"), # Missing price

(4, "T-shirt", 25.0, "Clothing"),

(5, "Jeans", 50.0, "Clothing"),

(6, "Dress", None, "Clothing"), # Missing price

(7, "Book", 15.0, "Books"),

(8, "Magazine", 5.0, "Books"),

(9, "Cookbook", None, "Books"), # Missing price

(10, "Tablet", 300.0, "Electronics")

]

# Create the DataFrame

products = spark.createDataFrame(data, schema)

# Show the original DataFrame

print("Original Products DataFrame:")

products.show()

# Calculate median price for each category

category\_medians = products.groupBy("category") \

.agg(median("price").alias("median\_price"))

print("Category Medians:")

category\_medians.show()

# Join original data with category medians

products\_with\_median = products.join(category\_medians, "category")

# Fill missing prices with category median

cleaned\_products = products\_with\_median.withColumn("cleaned\_price",

when(col("price").isNull(), col("median\_price")).otherwise(col("price"))

)

# Show results

print("Cleaned Products DataFrame:")

cleaned\_products.select("product\_id", "name", "category", "price", "cleaned\_price") \

.show()

# Stop the SparkSession

spark.stop()

# Scenario 5: Collaborative Filtering for Recommendation System

Question: You have two DataFrames: ratings (user\_id, movie\_id, rating) and movies (movie\_id, title, genre). Implement a simple item-based collaborative filtering system to recommend movies. For a given movie, find the top 5 most similar movies based on user ratings.

Answer:

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, udf, when, rand

from pyspark.sql.types import FloatType, IntegerType, StringType

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.linalg import Vectors

# Create a SparkSession

spark = SparkSession.builder.appName("MovieRecommendation").getOrCreate()

# Generate sample data

def generate\_sample\_data():

# Generate ratings data

ratings\_data = [(i, j, float(rand().cast(IntegerType()) % 5 + 1))

for i in range(1, 101) for j in range(1, 21)]

ratings = spark.createDataFrame(ratings\_data, ["user\_id", "movie\_id", "rating"])

# Generate movies data

movies\_data = [(i, f"Movie {i}", ["Action", "Comedy", "Drama"][i % 3])

for i in range(1, 21)]

movies = spark.createDataFrame(movies\_data, ["movie\_id", "title", "genre"])

return ratings, movies

# Generate sample data

ratings, movies = generate\_sample\_data()

# Pivot the ratings DataFrame to create a user-movie matrix

user\_movie\_matrix = ratings.groupBy("movie\_id").pivot("user\_id").agg({"rating": "first"})

# Fill null values with 0

user\_movie\_matrix = user\_movie\_matrix.na.fill(0)

# Create feature vectors

assembler = VectorAssembler(

inputCols=[col for col in user\_movie\_matrix.columns if col != "movie\_id"],

outputCol="features"

)

movie\_vectors = assembler.transform(user\_movie\_matrix).select("movie\_id", "features")

# Define a UDF to compute cosine similarity

def cosine\_similarity(v1, v2):

return float(v1.dot(v2) / (v1.norm(2) \* v2.norm(2)))

similarity\_udf = udf(cosine\_similarity, FloatType())

# Compute pairwise similarities

movie\_sims = movie\_vectors.alias("i").crossJoin(movie\_vectors.alias("j")) \

.select(

col("i.movie\_id").alias("movie\_i"),

col("j.movie\_id").alias("movie\_j"),

similarity\_udf(col("i.features"), col("j.features")).alias("similarity")

) \

.filter(col("movie\_i") < col("movie\_j")) # Avoid duplicate pairs

# Function to get top 5 similar movies

**def get\_similar\_movies(movie\_id):**

return movie\_sims.filter((col("movie\_i") == movie\_id) | (col("movie\_j") == movie\_id)) \

.select(

when(col("movie\_i") == movie\_id, col("movie\_j")).otherwise(col("movie\_i")).alias("similar\_movie\_id"),

col("similarity")

) \

.orderBy(col("similarity").desc()) \

.limit(5)

# Example: Get similar movies for movie\_id 1

similar\_movies = get\_similar\_movies(1)

print("Top 5 movies similar to Movie 1:")

similar\_movies.join(movies, similar\_movies.similar\_movie\_id == movies.movie\_id) \

.select("similar\_movie\_id", "title", "similarity") \

.show()

# Show some sample data

print("\nSample of ratings data:")

ratings.show(5)

print("\nSample of movies data:")

movies.show(5)

# Stop the SparkSession

spark.stop()

Question 1: Data Transformation and Aggregation

Problem:

You are given a PySpark DataFrame representing sales data with columns: product\_id, category, sales\_amount, date.

Calculate the total sales for each category for the most recent date in the dataset.

Expected Solution Outline:

from pyspark.sql.functions import col, max, sum

# Find the most recent date

latest\_date = df.agg(max("date")).collect()[0][0]

# Filter data for the latest date and aggregate sales by category

result = df.filter(col("date") == latest\_date) \

.groupBy("category") \

.agg(sum("sales\_amount").alias("total\_sales"))

result.show()

Problem:

You have a PySpark DataFrame with customer transactions: customer\_id, transaction\_amount, transaction\_date.

For each customer, calculate the 3-day rolling average of their transaction amounts.

Expected Solution Outline:

from pyspark.sql.window import Window

from pyspark.sql.functions import avg, col

# Define the window specification

window\_spec = Window.partitionBy("customer\_id") \

.orderBy("transaction\_date") \

.rowsBetween(-2, 0) # 3-day window including current row

# Calculate the rolling average

result = df.withColumn("rolling\_avg", avg("transaction\_amount").over(window\_spec))

result.show()

Question 3: Data Cleaning and Filtering

Problem:

You are given a PySpark DataFrame containing user data: user\_id, email, age, country.

Filter out rows where the email is invalid (doesn't contain '@') or the age is less than 18.

from pyspark.sql.functions import col

# Filter based on email validity and age

result = df.filter((col("email").contains("@")) & (col("age") >= 18))

result.show()

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, max, sum, avg

from pyspark.sql.window import Window

from pyspark.sql.types import \*

# Create a SparkSession

spark = SparkSession.builder.appName("KaratPySparkInterview").getOrCreate()

# Sample DataFrames (replace with actual data if available)

sales\_data = [

("P1", "Electronics", 1000, "2023-09-10"),

("P2", "Electronics", 500, "2023-09-09"),

("P3", "Clothing", 200, "2023-09-10"),

("P4", "Clothing", 300, "2023-09-08"),

]

sales\_schema = StructType([

StructField("product\_id", StringType(), True),

StructField("category", StringType(), True),

StructField("sales\_amount", IntegerType(), True),

StructField("date", StringType(), True),

])

sales\_df = spark.createDataFrame(sales\_data, sales\_schema)

transaction\_data = [

(1, 100, "2023-09-08"),

(1, 200, "2023-09-09"),

(1, 150, "2023-09-10"),

(2, 50, "2023-09-07"),

(2, 80, "2023-09-09"),

]

transaction\_schema = StructType([

StructField("customer\_id", IntegerType(), True),

StructField("transaction\_amount", IntegerType(), True),

StructField("transaction\_date", StringType(), True),

])

transaction\_df = spark.createDataFrame(transaction\_data, transaction\_schema)

user\_data = [

(1, "user1@example.com", 25, "USA"),

(2, "invalid\_email", 30, "Canada"),

(3, "user3@example.com", 15, "UK"),

]

user\_schema = StructType([

StructField("user\_id", IntegerType(), True),

StructField("email", StringType(), True),

StructField("age", IntegerType(), True),

StructField("country", StringType(), True),

])

user\_df = spark.createDataFrame(user\_data, user\_schema)

# Question 1: Total Sales by Category for Latest Date

latest\_date = sales\_df.agg(max("date")).collect()[0][0]

result\_q1 = sales\_df.filter(col("date") == latest\_date) \

.groupBy("category") \

.agg(sum("sales\_amount").alias("total\_sales"))

print("Question 1: Total Sales by Category for Latest Date")

result\_q1.show()

# Question 2: 3-Day Rolling Average of Transaction Amounts

window\_spec = Window.partitionBy("customer\_id") \

.orderBy("transaction\_date") \

.rowsBetween(-2, 0)

result\_q2 = transaction\_df.withColumn("rolling\_avg", avg("transaction\_amount").over(window\_spec))

print("\nQuestion 2: 3-Day Rolling Average of Transaction Amounts")

result\_q2.show()

# Question 3: Filter Invalid Emails and Underage Users

result\_q3 = user\_df.filter((col("email").contains("@")) & (col("age") >= 18))

print("\nQuestion 3: Filter Invalid Emails and Underage Users")

result\_q3.show()

# Stop the SparkSession

spark.stop()

Question 4: Joins and Data Enrichment

Problem:

You have two DataFrames:

orders\_df with columns: order\_id, customer\_id, order\_date, total\_amount

customers\_df with columns: customer\_id, customer\_name, city

Enrich the orders\_df by joining it with customers\_df to include customer\_name and city information.

Expected Solution Outline:

# Perform an inner join on 'customer\_id'

enriched\_orders\_df = orders\_df.join(customers\_df, on="customer\_id", how="inner")

# Select the desired columns

result\_q4 = enriched\_orders\_df.select("order\_id", "customer\_name", "city", "order\_date", "total\_amount")

print("\nQuestion 4: Joins and Data Enrichment")

result\_q4.show()

Question 5: Handling Missing Values

Problem:

You have a DataFrame with potentially missing values in the age column.

Fill the missing age values with the average age calculated from the non-missing values.

Expected Solution Outline:

from pyspark.sql.functions import avg, when, col, isnull

# Calculate the average age from non-missing values

average\_age = df.agg(avg(when(~isnull("age"), col("age")))).collect()[0][0]

# Fill missing 'age' values with the calculated average

result\_q5 = df.withColumn("age", when(isnull("age"), average\_age).otherwise(col("age")))

print("\nQuestion 5: Handling Missing Values")

result\_q5.show()

Question 6: User-Defined Functions (UDFs)

Problem:

You have a DataFrame with a text column containing product descriptions.

Create a UDF to extract the first word from each product description and add it as a new column first\_word.

Expected Solution Outline:

Python

from pyspark.sql.functions import udf

# Define the UDF to extract the first word

def extract\_first\_word(text):

if text:

return text.split()[0]

else:

return None

extract\_first\_word\_udf = udf(extract\_first\_word, StringType())

# Apply the UDF to create the new column

result\_q6 = df.withColumn("first\_word", extract\_first\_word\_udf(col("text")))

print("\nQuestion 6: User-Defined Functions (UDFs)")

result\_q6.show()