
PYSPARK - SCENARIO
BASED Q&A
BY - SHUBHAM WADEKAR

Question 1: Find the top N most frequent words in a large text file

Problem Explanation:

You are given a large text file. Your task is to count how frequently each word appears and return the top N most common words.

PySpark Code:

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import explode, split, col, desc

# Create Spark session

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

# Load text file

text_df = spark.read.text("path/to/large_text_file.txt")

# Split lines into words, flatten list, count frequency

words_df = text_df.select(explode(split(col("value"), "\\s+")).alias("word")) \

            .filter(col("word") != "") \

            .groupBy("word") \

            .count() \

            .orderBy(desc("count"))


# Get Top N (e.g., 10)

top_n = 10

top_n_words = words_df.limit(top_n)

top_n_words.show()
```

Explanation:

- **split** splits each line into words.
- **explode** flattens nested word lists.
- **groupBy().count()** gets word frequency.
- **orderBy(desc("count"))** sorts by frequency.
- **limit(n)** gets top N.

Question 2: Calculate the average salary and count of employees for each department

PySpark Code:

```
from pyspark.sql.functions import avg, count

# Assuming DataFrame name is df with columns: department, salary
result_df = df.groupBy("department") \
        .agg(avg("salary").alias("average_salary"),
          count("*").alias("employee_count"))

result_df.show()
```

Explanation:

- groupBy("department") groups employees by their department.
- agg(avg(), count()) calculates average salary and employee count.
- Alias names are set for clarity in output.

Question 3: Remove duplicate rows based on specific columns

Approach 1: Using dropDuplicates

```
# Drop duplicates based on specific columns (e.g., 'id' and 'name')
deduped_df = df.dropDuplicates(["id", "name"])
```

Approach 2: Keep the latest record using Window

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
window_spec = Window.partitionBy("id", "name").orderBy(col("timestamp").desc())
latest_df = df.withColumn("row_num", row_number().over(window_spec)) \
        .filter(col("row_num") == 1) \
        .drop("row_num")
```

Question 4: Filter rows where salary > 5000 and select only the name column

PySpark Code:

```
filtered_df = df.filter(col("salary") > 5000).select("name")

filtered_df.show()
```

Explanation:

- filter() applies row-level condition.
- select() narrows the output to only the name column.

Question 5: Drop rows with nulls in the age column

PySpark Code:

```
cleaned_df = df.dropna(subset=["age"])

cleaned_df.show()
```

Alternate Option: Fill with default value

```
filled_df = df.fillna({"age": 0})
```

Explanation:

- dropna() removes rows where the specified column has a null.
- fillna() is useful when you want to keep rows but replace nulls with defaults.

Question 6: Add a New Column to a DataFrame

Problem Explanation:
You are asked to add a new column to a PySpark DataFrame. The new column may have a constant value or be derived from other columns.

Optimized PySpark Code:

```
from pyspark.sql.functions import lit

# Add a constant value column

df_new = df.withColumn("bonus", lit(1000))

# Add a derived column (e.g., total_compensation)

df_new = df_new.withColumn("total_compensation", df_new["salary"] + df_new["bonus"])
```

Step-by-step Explanation:

1. Use withColumn() to create or overwrite a column.
2. lit() is used to insert a literal constant value.
3. You can chain another withColumn() to create derived columns.
4. The new DataFrame will include the added column(s).

Question 7: Perform an Inner Join on Two DataFrames

Problem Explanation:
You are given two DataFrames: employee_df and department_df. You need to join them on dept_id to find which department each employee belongs to.

Optimized PySpark Code:

```
joined_df = employee_df.join(department_df, on="dept_id", how="inner")

joined_df.show()
```

Step-by-step Explanation:

1. Use the join() method and pass the join column (dept_id).
2. Specify how="inner" for inner join (default behavior).
3. The resulting DataFrame will include only matching rows from both tables.

Question 8: Perform a Left Join to Include All Employees

Problem Explanation:

You are given employees and departments DataFrames. Perform a left join to include all employees, even those without departments.

Optimized PySpark Code:

```
left_joined_df = employees.join(departments, on="dept_id", how="left")

left_joined_df.show()
```

Step-by-step Explanation:

1. Use join() with how="left" to perform a left outer join.
2. This includes all records from the left DataFrame (employees).
3. If an employee doesn't belong to any department, the dept_name will be null.

Question 9: Perform a Right Join to Get All Customers Including Those Without Orders

Problem Explanation:
Given two datasets — orders and customers — you need to perform a right join so that all customers are included, even if they didn't place any orders.

Optimized PySpark Code:

```
right_joined_df = orders.join(customers, on="customer_id", how="right")

right_joined_df.show()
```

Step-by-step Explanation:

1. Use join() with how="right" to ensure all customer data is preserved.
2. Matching order data will be attached where available.
3. Unmatched rows from orders will show null values for order-related columns.

Question 10: Calculate Running Total of Stock Prices for Each Symbol

Problem Explanation:
Given a dataset of daily stock prices with stock_symbol and price, calculate the running total (cumulative sum) of prices for each stock.

Optimized PySpark Code:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import sum as _sum

window_spec =
Window.partitionBy("stock_symbol").orderBy("date").rowsBetween(Window.unboundedPrece
ding, 0)

df_running_total = df.withColumn("running_total", _sum("price").over(window_spec))

df_running_total.show()
```

Step-by-step Explanation:

1. Define a window partitioned by stock_symbol and ordered by date.
2. Use .rowsBetween(Window.unboundedPreceding, 0) for cumulative sum.
3. Apply _sum().over(window_spec) to calculate the running total.
4. The withColumn() adds the running_total to the DataFrame.

Question 11: Rename Columns in a DataFrame

Problem Explanation:
You need to rename one or more columns in a PySpark DataFrame, either for clarity, standardization, or schema alignment.

Optimized PySpark Code:

```
# Rename a single column
df_renamed = df.withColumnRenamed("old_name", "new_name")
# Rename multiple columns
df_renamed = df.withColumnRenamed("id", "employee_id") \
    .withColumnRenamed("dept", "department")
```

Step-by-Step Explanation:

1. Use withColumnRenamed() to rename columns one at a time.
2. You can chain multiple renaming operations.
3. This returns a new DataFrame with updated column names.

Question 12: Create a New Column Derived from Existing Columns

Problem Explanation:
You need to add a new column that is a combination or transformation of other columns — e.g., full name from first and last name.

Optimized PySpark Code:

```
from pyspark.sql.functions import concat, lit
df_fullname = df.withColumn("full_name", concat(col("first_name"), lit(" "), col("last_name")))
```

Step-by-Step Explanation:

1. Use withColumn() to add a new column.
2. Use concat() to combine strings.
3. Use lit() to add literal values like a space.
4. Resulting column full_name will contain combined names.

Question 13: Sort a DataFrame Based on One or More Columns

Problem Explanation:
You are asked to sort a DataFrame based on one or multiple columns — for example, sort employees by department and descending salary.

Optimized PySpark Code:

```
df_sorted = df.orderBy("department", col("salary").desc())

df_sorted.show()
```

Step-by-Step Explanation:

1. Use orderBy() to sort.
2. Pass column names or col().desc() for descending order.
3. Multiple columns can be passed for multi-level sorting.
4. orderBy returns a sorted DataFrame.

Question 14: Write a PySpark DataFrame to a CSV File

Problem Explanation:
You need to export your transformed or final DataFrame to a CSV file on disk.

Optimized PySpark Code:

```
df.write.option("header", True).csv("/path/to/output_directory")
```

Step-by-Step Explanation:

1. Use .write on the DataFrame.
2. Use .option("header", True) to write column names.
3. .csv(path) specifies the output location.
4. Output will be a folder with part files (CSV chunks per partition).

Question 15: Use rank() Function to Rank Employees Based on Salary Within Department

Problem Explanation:
You need to rank employees based on their salary within each department using PySpark's window functions.

Optimized PySpark Code:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import rank

window_spec = Window.partitionBy("department").orderBy(col("salary").desc())

df_ranked = df.withColumn("rank", rank().over(window_spec))

df_ranked.show()
```

Step-by-Step Explanation:

1. Define a WindowSpec partitioned by department and ordered by descending salary.
2. Use the rank() function with .over(window_spec).
3. withColumn() adds a rank column to the DataFrame.
4. Ranking resets for each department and accounts for ties.

Question 16: Perform a Simple Arithmetic Operation on DataFrame Columns

Problem Explanation:
You want to perform arithmetic operations like addition, subtraction, etc., between two columns (e.g., calculating total compensation = salary + bonus).

Optimized PySpark Code:

```
df = df.withColumn("total_compensation", col("salary") + col("bonus"))
```

Step-by-Step Explanation:

1. Use withColumn() to create a new column.
2. Perform arithmetic using col("column_name") from pyspark.sql.functions.
3. Operations like +, -, *, / can be used directly.
4. Resulting column holds the computed value for each row.

Question 17: Use coalesce() to Reduce the Number of Partitions

Problem Explanation:
Your DataFrame has too many partitions (e.g., after reading from a large source). You need to reduce the number of partitions before writing to optimize performance.

Optimized PySpark Code:

```
df_repartitioned = df.coalesce(4)
```

Step-by-Step Explanation:

1. coalesce(n) merges partitions without shuffling — ideal before writing.
2. Use when reducing partition count (e.g., from 8 → 4).
3. More efficient than repartition() in this case.
4. Improves performance during file writes and small data loads.

Question 18: Customer Transaction Aggregation and Filtering

Problem Explanation:
Given a DataFrame with customer_id, transaction_date, and amount, calculate total, average, and count of transactions per customer. Then filter those with more than 5 transactions.

Optimized PySpark Code:

```
from pyspark.sql.functions import sum, avg, count
agg_df = df.groupBy("customer_id").agg(
  sum("amount").alias("total_amount"),
  avg("amount").alias("average_amount"),
  count("*").alias("transaction_count")
)
filtered_df = agg_df.filter(col("transaction_count") > 5)
filtered_df.show()
```

Step-by-Step Explanation:

1. Group by customer_id.
2. Aggregate total, average, and count using sum(), avg(), count().
3. Rename columns using alias().
4. Use filter() to retain customers with more than 5 transactions.

Question 19: Different Ways to Read Data into PySpark

Problem Explanation:
You want to read data from various sources like CSV, JSON, Parquet, ORC, and relational databases using PySpark.

Optimized PySpark Code:

```
# CSV

df_csv = spark.read.option("header", True).csv("path/to/file.csv")


# JSON

df_json = spark.read.json("path/to/file.json")


# Parquet

df_parquet = spark.read.parquet("path/to/file.parquet")


# MySQL (JDBC)

df_mysql = spark.read \

  .format("jdbc") \

  .option("url", "jdbc:mysql://localhost:3306/db") \

  .option("dbtable", "table_name") \

  .option("user", "username") \

  .option("password", "password") \

  .load()
```

Step-by-Step Explanation:

1. Use appropriate format like .csv(), .json(), .parquet() to load files.
2. Use .option() to configure headers, delimiters, etc.
3. For databases, use .format(\"jdbc\") with connection details.
4. Always validate schema post load using df.printSchema().

Question 20: Create a SparkSession and Explain Its Uses

Problem Explanation:
You need to create a SparkSession, which is the entry point to use DataFrame and SQL APIs in PySpark.

Optimized PySpark Code:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \

  .appName("MySparkApp") \

  .getOrCreate()
```

Step-by-Step Explanation:

1. SparkSession is the gateway to all Spark functionality (DF, SQL, Streaming).
2. builder.appName() gives a name to the Spark application.
3. .getOrCreate() creates a session or reuses an existing one.
4. Once created, use spark.read, spark.sql(), spark.createDataFrame() etc.

Question 21: Filter Customers Whose Names Start with 'A'

Problem Explanation:
You need to filter records in a DataFrame to retrieve only those customers whose names begin with the letter 'A'.

Optimized PySpark Code:

```
filtered_df = df.filter(col("name").startswith("A"))

filtered_df.show()
```

Step-by-Step Explanation:

1. Use the filter() function to apply conditions.
2. Use .startswith("A") from the Column class to filter names.
3. The result includes only those rows where the name column starts with 'A'.

Question 22: Calculate Each Employee's Salary Percentage Contribution to Their Department

Problem Explanation:
You need to compute what percentage of the total departmental salary each employee contributes.

Optimized PySpark Code:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import sum, col

window_spec = Window.partitionBy("department")

df_with_total = df.withColumn("total_dept_salary", sum("salary").over(window_spec))

df_percentage = df_with_total.withColumn("salary_pct", (col("salary") /
col("total_dept_salary")) * 100)

df_percentage.select("employee_id", "department", "salary", "salary_pct").show()
```

Step-by-Step Explanation:

1. Define a window partitioned by department for group-wise aggregation.
2. Use sum() over the window to compute total salary per department.
3. Compute salary percentage using division and multiplication.
4. Select relevant columns for the final output.

Question 23: Replace Department Name 'Finance' with 'Financial Services'

Problem Explanation:
You need to update department names by replacing all instances of "Finance" with "Financial Services".

Optimized PySpark Code:

```
from pyspark.sql.functions import when

df_updated = df.withColumn("department", when(col("department") == "Finance", "Financial
Services").otherwise(col("department")))
```

Step-by-Step Explanation:

1. Use withColumn() to overwrite the department column.
2. Use when() to check if department is "Finance".
3. Replace it with "Financial Services" using otherwise() to retain other values.

Question 24: Optimize PySpark Jobs for Performance

Problem Explanation:
You need to apply best practices and configuration changes to optimize the performance of PySpark jobs.

Optimized PySpark Code (Best Practices):

```
# 1. Repartitioning before heavy shuffles
df_repart = df.repartition("key_column")


# 2. Caching intermediate results
df_cached = df_repart.cache()


# 3. Broadcast join
from pyspark.sql.functions import broadcast
df_joined = df_cached.join(broadcast(small_df), "id")
```

Step-by-Step Explanation:

1. Use repartition() or coalesce() to balance partitions.
2. Use cache() to persist intermediate results in memory.
3. Apply broadcast() for small lookup DataFrames to avoid large shuffles.
4. Avoid using UDFs unless built-in functions can't solve the problem.
5. Monitor stages using Spark UI to identify bottlenecks.

Question 25: Calculate the Correlation Between Columns

Problem Explanation:
You want to measure the linear correlation (Pearson) between two numerical columns in a DataFrame.

Optimized PySpark Code:

```
from pyspark.sql.functions import corr
df.select(corr("column1", "column2").alias("correlation")).show()
```

Step-by-Step Explanation:

1. Use corr() function from pyspark.sql.functions.
2. Pass the two numeric columns as arguments.
3. It returns a scalar value between -1 and 1 representing correlation.
4. 1 means strong positive, -1 strong negative, 0 means no correlation.

Question 26: Handle Time-Series Data in PySpark

Problem Explanation:
You are working with time-series data and want to perform operations like sorting, windowing, or calculating time-based metrics.

Optimized PySpark Code:

```
from pyspark.sql.functions import to_timestamp

from pyspark.sql.window import Window

from pyspark.sql.functions import lag

df_ts = df.withColumn("timestamp", to_timestamp("event_time", "yyyy-MM-dd HH:mm:ss"))

window_spec = Window.partitionBy("sensor_id").orderBy("timestamp")

df_lag = df_ts.withColumn("prev_value", lag("reading").over(window_spec))

df_lag.show()
```

Step-by-Step Explanation:

1. Convert string-based time column to timestamp using to_timestamp().
2. Define a window over sensor_id ordered by time.
3. Use lag() to fetch previous readings (e.g., for calculating deltas).
4. This setup is commonly used in trend analysis, forecasting, etc.

Question 27: Update Nested Columns in PySpark

Problem Explanation:
Your DataFrame contains a complex/nested column (struct). You need to update one of the nested fields without affecting others.

Optimized PySpark Code:

```
from pyspark.sql.functions import col, struct
df_updated = df.withColumn("address",
        struct(
          col("address.street"),
          col("address.city"),
          col("address.zip"),
          col("address.country").alias("new_country")
        ))
```

Step-by-Step Explanation:

1. Access nested fields using dot notation (e.g., address.city).
2. Use struct() to rebuild the nested column with updated values.
3. Assign the new struct back to the parent field (e.g., "address").
4. This ensures only the desired nested field is modified.

Question 28: Explain PySpark UDF with an Example

Problem Explanation:
You want to apply custom logic that can't be achieved with built-in functions, using a User Defined Function (UDF).

Optimized PySpark Code:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
def convert_case(name):
  return name.upper()
convert_case_udf = udf(convert_case, StringType())
df_udf = df.withColumn("upper_name", convert_case_udf(col("name")))
df_udf.show()
```

Step-by-Step Explanation:

1. Define a function (convert_case).
2. Register it as a UDF with return type using udf() and StringType().
3. Apply it to a DataFrame column with withColumn().
4. Avoid UDFs for performance-sensitive logic — use built-ins when possible.

Question 29: Load a File with Custom Delimiter (~|)

Problem Explanation:
Your data file uses a complex delimiter like ~| and needs to be loaded into a PySpark DataFrame.

Optimized PySpark Code:

```
df_custom = spark.read.option("delimiter", "~|") \
            .option("header", True) \
            .csv("path/to/file.txt")
df_custom.show()
```

Step-by-Step Explanation:

1. Use .option("delimiter", "~|") to specify the custom delimiter.
2. Include .option("header", True) if the first row is a header.
3. Load using .csv() — PySpark will split based on your custom delimiter.
4. Always check schema using df.printSchema().

Question 30: Cover All PySpark Concepts and Commands for Data Engineering

Problem Explanation:
You want to summarize essential PySpark commands for data engineering — including I/O, transformations, aggregations, joins, and performance tuning.

Optimized PySpark Concepts Summary:

```
# I/O
df = spark.read.option("header", True).csv("file.csv")
df.write.mode("overwrite").parquet("path/")
```

```python
# Transformations
df = df.withColumn("age_plus_5", col("age") + 5).drop("temp_col")

# Aggregations
df.groupBy("dept").agg(avg("salary"), max("age"))

# Joins
df1.join(df2, "emp_id", "inner")

# Window Functions
window_spec = Window.partitionBy("dept").orderBy("salary")
df.withColumn("rank", rank().over(window_spec))

# UDFs
spark.udf.register("upper_case", lambda x: x.upper())

# Performance
df.cache()
df.repartition("dept")
```

Step-by-Step Explanation:

1. Focus on core areas: I/O (CSV, Parquet), transformation (withColumn, drop), aggregation (groupBy + agg()), joins, window functions, UDFs, and performance tuning.
2. Learn to read/write data efficiently.
3. Prefer built-in functions over UDFs for performance.
4. Tune partitions and use caching wisely in heavy jobs.

Question 31. Filter rows where the email domain is gmail.com and the last_login is within the past 30 days

Problem Explanation:
Filter users whose email ends with @gmail.com and who logged in within the last 30 days.

PySpark Code:

```
from pyspark.sql.functions import col, current_date, to_date, expr

df_filtered = df.filter(

  (col("email").endswith("@gmail.com")) &

  (to_date(col("last_login")) >= expr("date_sub(current_date(), 30)"))

)
```

Explanation:

- endswith() checks the domain of email.
- to_date() ensures the last_login is treated as a date.
- expr("date_sub(...)") gets the date 30 days before today.
- Combined filter ensures only recent Gmail users are returned.

Question 32. Replace nulls in multiple columns with default values in a single operation

Problem Explanation:
Fill missing/null values in specific columns with user-defined defaults.

PySpark Code:

```
df_filled = df.fillna({

  "city": "Unknown",

  "age": 0,

  "email": "not_provided@example.com"

})
```

Explanation:

- fillna() accepts a dictionary to apply default values.
- Each key is a column name and the value is the replacement for nulls.
- Efficient and concise for batch null-handling.

Question 33. Extract only alphabetic characters from a mixed alphanumeric column

Problem Explanation:
Strip out non-alphabet characters from a string field using regex.

PySpark Code:

```
from pyspark.sql.functions import regexp_replace

df_alpha = df.withColumn("alpha_only", regexp_replace(col("mixed_col"), "[^A-Za-z]", ""))
```

Explanation:

- regexp_replace() removes all characters that are not A–Z or a–z.
- This leaves only alphabetic letters in the new alpha_only column.

Question 34. Identify and mask PII like phone numbers and SSNs

Problem Explanation:
Mask sensitive values by hiding the middle digits of PII fields like SSN and phone.

PySpark Code:

```
from pyspark.sql.functions import regexp_replace

df_masked = df.withColumn("phone_masked", regexp_replace("phone", r"(\d{3})\d{4}(\d{3})", r"\1****\2")) \
        .withColumn("ssn_masked", regexp_replace("ssn", r"(\d{3})\d{2}(\d{4})", r"\1**\2"))
```

Explanation:

- Uses regex to match and mask middle digits.
- Keeps outer digits for traceability but hides sensitive parts.

Question 35. Parse a column of JSON strings and explode the nested array inside

Problem Explanation:
Parse a JSON column and flatten array data into individual rows.

PySpark Code:

```
from pyspark.sql.functions import from_json, explode, col
from pyspark.sql.types import StructType, ArrayType, StringType
schema = StructType().add("items", ArrayType(StringType()))
df_parsed = df.withColumn("json_data", from_json(col("json_col"), schema))
df_exploded = df_parsed.select(explode(col("json_data.items")).alias("item"))
```

Explanation:

- from_json() parses the raw JSON string.
- explode() flattens the array into multiple rows.
- Assumes JSON has a field like {"items": ["a", "b", "c"]}.

Question 36. Convert a CSV column string "1,2,3" into an array of integers

Problem Explanation:
Split a string by delimiter and convert to integer array.

PySpark Code:

```
from pyspark.sql.functions import split, col
from pyspark.sql.types import ArrayType, IntegerType
df_split = df.withColumn("int_array", split(col("csv_str"), ",").cast(ArrayType(IntegerType())))
```

Explanation:

- split() splits the string into an array of strings.
- .cast(ArrayType(IntegerType())) converts elements to integers.

Question 37. Generate a new column that tags users as "new" or "returning" based on first visit

Problem Explanation:
Tag users as "new" if their first visit was in the last 7 days, else "returning".

PySpark Code:

```
from pyspark.sql.functions import to_date, current_date, expr, when

df_tagged = df.withColumn("user_type", when(

  to_date("first_visit") >= expr("date_sub(current_date(), 7)"), "new"

).otherwise("returning"))
```

Explanation:

- Compares first_visit date with today's minus 7 days.
- Tags based on recency of first interaction.

Question 38. Apply conditional logic to derive a risk_score based on multiple columns

Problem Explanation:
Create a derived column using multiple if-else conditions.

PySpark Code:

```
from pyspark.sql.functions import when

df_risk = df.withColumn("risk_score", when((col("age") < 25) & (col("income") < 30000), "High")

                  .when((col("age") >= 25) & (col("income") < 50000), "Medium")

                  .otherwise("Low"))
```

Explanation:

- Nested when() statements emulate if-elif-else logic.
- Multiple column conditions used to compute a business-specific score.

Question 39. Replace any value below 0 in numeric columns with the column mean

Problem Explanation:
For each column, replace negative numbers with the mean of that column.

PySpark Code:

```
from pyspark.sql.functions import col, mean, when

mean_val = df.select(mean("amount")).first()[0]

df_replaced = df.withColumn("amount", when(col("amount") < 0,
mean_val).otherwise(col("amount")))
```

Explanation:

- Compute mean of the column.
- Replace values less than 0 using when() and otherwise().

Question 40. Derive a full_address column by combining street, city, and zip

Problem Explanation:
Concatenate address fields into a single formatted string.

PySpark Code:

```
from pyspark.sql.functions import concat_ws


df_address = df.withColumn("full_address", concat_ws(", ", "street", "city", "zip"))
```

Explanation:

- concat_ws() joins strings with a delimiter.
- Columns are combined in a readable format.

Question 41. Filter customers who placed orders but have never logged in

Problem Explanation:
Identify customers present in the orders table but absent from the login table.

PySpark Code:

```
orders_df.join(logins_df, "customer_id", "left_anti").show()
```

Explanation:

- left_anti join returns rows from the left table with no match in the right.
- Captures customers who have orders but no login activity.

Question 42. Tokenize a text column into individual words and explode into separate rows

Problem Explanation:
Split sentences into words and expand each word into a row.

PySpark Code:

```
from pyspark.sql.functions import split, explode

df_words = df.withColumn("words", split(col("text"), " "))

df_exploded = df_words.select(explode("words").alias("word"))
```

Explanation:

- split() turns sentence into an array of words.
- explode() creates a new row for each word.

Question 43. Normalize a numeric column using min-max scaling

Problem Explanation:
Transform a column to be between 0 and 1 using min-max normalization.

PySpark Code:

```
from pyspark.sql.functions import min, max

min_val = df.agg(min("score")).first()[0]

max_val = df.agg(max("score")).first()[0]

df_scaled = df.withColumn("normalized_score", (col("score") - min_val) / (max_val - min_val))
```

Explanation:

- Compute min and max separately.
- Normalize each value using formula: (val - min) / (max - min).

Question 44. Extract the top-level domain from a list of URLs

Problem Explanation:
Extract the .com, .org, etc., part from a URL.

PySpark Code:

```
from pyspark.sql.functions import regexp_extract

df_tld = df.withColumn("tld", regexp_extract(col("url"), r"\.([a-z]{2,6})$", 1))
```

Explanation:

- Uses regex to capture characters after the last period in a URL.
- Assumes the domain ends with .com, .net, etc.

Question 45. Tag rows as "weekday" or "weekend" based on a timestamp column

Problem Explanation:
Check the day of the week and label it accordingly.

PySpark Code:

```
from pyspark.sql.functions import date_format, when

df_tagged = df.withColumn("day_type", when(date_format("timestamp", "u").isin("6", "7"), "weekend")
                        .otherwise("weekday"))
```

Explanation:

- date_format(..., "u") returns day of the week (1 = Monday, 7 = Sunday).
- Weekend: Saturday (6) and Sunday (7).
- when().otherwise() applies the label.

Question 46

Problem Explanation:
Perform a semi-join to return only users who have matching records in the transactions table (i.e., users who made at least one transaction).

PySpark Code:

```
users_with_txn = users_df.join(transactions_df, "user_id", "left_semi")
```

Explanation:

- left_semi join returns records from the left DataFrame (users_df) where a match is found in the right (transactions_df).
- No duplicate or joined columns — only filters rows.

Question 47

Problem Explanation:
Join two datasets on a key and ensure nulls from the right side are replaced with default values.

PySpark Code:

```
from pyspark.sql.functions import coalesce, lit

joined_df = orders_df.join(products_df, "product_id", "left") \
    .withColumn("product_name", coalesce(col("product_name"), lit("Unknown"))) \
    .withColumn("price", coalesce(col("price"), lit(0)))
```

Explanation:

- left join brings all orders, with matching product info.
- coalesce() fills nulls with fallback values.

Question 48

Problem Explanation:
Identify mismatches between two DataFrames on key fields (e.g., user_id) and log them separately.

PySpark Code:

```
mismatches = df1.join(df2, "user_id", "outer") \
    .filter(df1["email"] != df2["email"])
```

Explanation:

- outer join brings all rows from both DataFrames.
- Filter identifies mismatched values on shared key.
- Can be saved/logged for auditing.

Question 49

Problem Explanation:
Perform a fuzzy join on names using string similarity (Levenshtein distance).

PySpark Code:

```
from pyspark.sql.functions import col, levenshtein

fuzzy_join = df1.crossJoin(df2) \
  .filter(levenshtein(col("df1.name"), col("df2.name")) < 3)
```

Explanation:

- crossJoin creates all combinations (may be heavy).
- levenshtein() computes string similarity.
- Filters out close matches (within edit distance 2).

Question 50

Problem Explanation:
Merge customer records from multiple sources and deduplicate based on email or phone.

PySpark Code:

```
from pyspark.sql.functions import row_number

from pyspark.sql.window import Window

combined_df = df1.union(df2)

windowSpec = Window.partitionBy("email", "phone").orderBy("last_updated")

deduped_df = combined_df.withColumn("rn", row_number().over(windowSpec)).filter("rn = 1")
```

Explanation:

- union() merges datasets.
- row_number() gives priority to latest records.
- Keeps only one row per email-phone pair.

Question 51

Problem Explanation:
Perform a conditional join where one column must match exactly and another within a range.

PySpark Code:

```
joined_df = df1.join(df2,
  (df1["user_id"] == df2["user_id"]) &
  (df1["timestamp"].between(df2["start_time"], df2["end_time"]))
)
```

Explanation:

- Join matches on exact user_id.
- Applies between() on a time range condition.

Question 52

Problem Explanation:
Join two DataFrames and filter out rows where join keys were null on either side.

PySpark Code:

```
df_full = df1.join(df2, "id", "outer") \
  .filter(df1["id"].isNotNull() & df2["id"].isNotNull())
```

Explanation:

- outer join includes all records.
- Filter removes unmatched rows (i.e., missing keys on either side).

Question 53

Problem Explanation:
Enrich a streaming event stream with static customer profile data using a broadcast join.

PySpark Code:

```
from pyspark.sql.functions import broadcast
enriched_df = stream_df.join(broadcast(static_df), "customer_id", "left")
```

Explanation:

- Broadcasts small static DataFrame for efficient join with streaming data.
- Ideal when static data fits in memory.

Question 54

Problem Explanation:
Join a streaming DataFrame with a static reference DataFrame and update in near real-time.

PySpark Code:

```
enriched_stream = stream_df.join(static_df, "device_id", "left")
```

Explanation:

- Static DF must be periodically refreshed if required.
- Works in structured streaming mode without breaking continuity.

Question 55

Problem Explanation:
Join product catalog with sales data, but only include active products.

PySpark Code:

```
active_sales = sales_df.join(products_df.filter("status = 'active'"), "product_id", "inner")
```

Explanation:

- Filters products_df to only active records before the join.
- Ensures only current/valid product sales are analyzed.

Question 56

Problem Explanation:
Assign a row number to each transaction per user ordered by transaction date.

PySpark Code:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import row_number

windowSpec = Window.partitionBy("user_id").orderBy("transaction_date")

df_ranked = df.withColumn("txn_rank", row_number().over(windowSpec))
```

Explanation:

- Ranks transactions within each user group.
- Can be used to get latest or N-th transaction per user.

Question 57

Problem Explanation:
Calculate running average transaction amount per customer using a window.

PySpark Code:

```
from pyspark.sql.functions import avg

windowSpec =
Window.partitionBy("customer_id").orderBy("txn_date").rowsBetween(Window.unboundedPre
ceding, 0)

df_avg = df.withColumn("running_avg", avg("amount").over(windowSpec))
```

Explanation:

- Cumulative average up to current row.
- Uses dynamic window frame unboundedPreceding.

Question 58

Problem Explanation:
Detect consecutive failed login attempts using lag() and lead().

PySpark Code:

```
from pyspark.sql.functions import lag

from pyspark.sql.window import Window

windowSpec = Window.partitionBy("user_id").orderBy("login_time")

df_flagged = df.withColumn("prev_status", lag("status").over(windowSpec)) \
        .filter((col("status") == "failed") & (col("prev_status") == "failed"))
```

Explanation:

- Uses lag() to compare current and previous row.
- Filters for repeated failures.

Question 59

Problem Explanation:
Identify the first and last transaction per customer using window functions.

PySpark Code:

```
from pyspark.sql.functions import first, last

windowSpec = Window.partitionBy("customer_id").orderBy("transaction_date")

df_bounds = df.withColumn("first_txn", first("transaction_id").over(windowSpec)) \
    .withColumn("last_txn", last("transaction_id").over(windowSpec))
```

Explanation:

- Window ordered by transaction date gives transaction range per customer.

Question 60

Problem Explanation:
Compute percent change in price compared to previous day for each product.

PySpark Code:

```
from pyspark.sql.functions import lag, col

windowSpec = Window.partitionBy("product_id").orderBy("date")

df_delta = df.withColumn("prev_price", lag("price").over(windowSpec)) \
    .withColumn("pct_change", ((col("price") - col("prev_price")) / col("prev_price")) * 100)
```

Explanation:

- Computes percentage difference between current and previous price.

Question 61

Problem Explanation:
Tag each row as "increasing" or "decreasing" based on comparison with previous row.

PySpark Code:

```
from pyspark.sql.functions import when

df_trend = df.withColumn("prev_value", lag("value").over(windowSpec)) \
    .withColumn("trend", when(col("value") > col("prev_value"), "increasing")
                .when(col("value") < col("prev_value"), "decreasing")
                .otherwise("same"))
```

Explanation:

- Checks value trend row by row.
- Great for time-series or pricing data.

Question 62

Problem Explanation:
Assign ranks to employees based on salary within their department using dense_rank().

PySpark Code:

```
from pyspark.sql.functions import dense_rank

windowSpec = Window.partitionBy("dept_id").orderBy(col("salary").desc())

df_ranked = df.withColumn("rank", dense_rank().over(windowSpec))
```

Explanation:

- Ranks employees within each department.
- dense_rank() doesn't skip numbers on tie.

Question 63

Problem Explanation:
Create a cumulative sum of monthly revenue per region.

PySpark Code:

```
from pyspark.sql.functions import sum

windowSpec =
Window.partitionBy("region").orderBy("month").rowsBetween(Window.unboundedPreceding,
0)

df_cum = df.withColumn("cumulative_revenue", sum("revenue").over(windowSpec))
```

Explanation:

- Calculates running total revenue for a region across time.

Question 64

Problem Explanation:
Compute the difference between max and min in a sliding 7-day window per product.

PySpark Code:

```
from pyspark.sql.functions import max, min

windowSpec = Window.partitionBy("product_id").orderBy("date").rowsBetween(-6, 0)

df_windowed = df.withColumn("price_range", max("price").over(windowSpec) -
min("price").over(windowSpec))
```

Explanation:

- Sliding window of 7 days (current and past 6).
- Tracks volatility in price.

Question 65

Problem Explanation:
Flag abnormal values that are 2 standard deviations above rolling average.

PySpark Code:

```
from pyspark.sql.functions import avg, stddev

windowSpec = Window.partitionBy("metric").orderBy("timestamp").rowsBetween(-6, 0)

df_anomaly = df.withColumn("rolling_avg", avg("value").over(windowSpec)) \

        .withColumn("rolling_std", stddev("value").over(windowSpec)) \

        .withColumn("is_anomaly", (col("value") > col("rolling_avg") + 2 * col("rolling_std")))
```

Explanation:

- Flags outliers using statistical threshold.
- Uses rolling window for dynamic detection.

Question 66

Problem Explanation:
Calculate the median salary per department using approxQuantile() since PySpark doesn't support an exact median in aggregation.

PySpark Code:

```
median_salary = df.groupBy("department").agg(

   expr("percentile_approx(salary, 0.5)").alias("median_salary")

)
```

Step-by-Step Explanation:

- percentile_approx() is used to approximate quantiles, including median (0.5).
- Grouping by department allows per-group aggregation.
- It's much faster and scalable than collecting and sorting manually.

Question 67

Problem Explanation:
Group customer orders by month and calculate total spend and order count.

PySpark Code:

```
from pyspark.sql.functions import month, sum, count

monthly_summary = orders_df.groupBy(month("order_date").alias("month")) \
    .agg(sum("amount").alias("total_spent"), count("*").alias("order_count"))
```

Step-by-Step Explanation:

- Extracts month from order_date.
- Aggregates total spend and number of orders per month.

Question 68

Problem Explanation:
Identify customers with total purchases over $10,000 in the past year.

PySpark Code:

```
from pyspark.sql.functions import col, current_date, date_sub

past_year_orders = orders_df.filter(col("order_date") >= date_sub(current_date(), 365))

high_value_customers = past_year_orders.groupBy("customer_id") \
    .agg(sum("amount").alias("total_purchases")) \
    .filter(col("total_purchases") > 10000)
```

Step-by-Step Explanation:

- Filters for orders in the last 365 days.
- Aggregates spend by customer and filters based on threshold.

Question 69

Problem Explanation:
Find the top 3 products sold in each category by total revenue.

PySpark Code:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import sum, dense_rank

revenue_df = sales_df.groupBy("category", "product_id") \

  .agg(sum("amount").alias("total_revenue"))

windowSpec = Window.partitionBy("category").orderBy(col("total_revenue").desc())

top3_products = revenue_df.withColumn("rank", dense_rank().over(windowSpec)) \

  .filter(col("rank") <= 3)
```

Step-by-Step Explanation:

- First computes revenue per product-category.
- Then uses window function to rank products and filter top 3.

Question 70

Problem Explanation:
Count number of distinct visitors per hour on a website.

PySpark Code:

```
from pyspark.sql.functions import hour, countDistinct

visits_by_hour = logs_df.groupBy(hour("timestamp").alias("hour")) \

  .agg(countDistinct("user_id").alias("unique_visitors"))
```

Step-by-Step Explanation:

- Extracts hour from timestamp.
- Counts distinct users in each hourly bucket.

Question 71

Problem Explanation:
Determine the most common product bought per region using grouping and ranking.

PySpark Code:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import count, col, dense_rank

product_counts = sales_df.groupBy("region", "product_id") \

  .agg(count("*").alias("product_count"))

windowSpec = Window.partitionBy("region").orderBy(col("product_count").desc())

most_common_product = product_counts.withColumn("rank",
dense_rank().over(windowSpec)) \

  .filter(col("rank") == 1)
```

Step-by-Step Explanation:

- Group sales by region and product, then count occurrences.
- Use window function to rank products per region.
- Filter top-ranked product(s).

Question 72

Problem Explanation:
Aggregate sales data and format it as a JSON string per region.

PySpark Code:

```
from pyspark.sql.functions import collect_list, to_json, struct

json_per_region = sales_df.groupBy("region") \

  .agg(to_json(collect_list(struct("product_id", "amount"))).alias("sales_json"))
```

Step-by-Step Explanation:

- Use collect_list and struct to combine columns per row.
- Convert collected rows into a JSON string with to_json.

Question 73

Problem Explanation:
Calculate the average number of days between orders per customer.

PySpark Code:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col, lag, avg, datediff
windowSpec = Window.partitionBy("customer_id").orderBy("order_date")
df_lagged = orders_df.withColumn("prev_order", lag("order_date").over(windowSpec))
df_diff = df_lagged.withColumn("days_between", datediff("order_date", "prev_order"))
avg_days = df_diff.groupBy("customer_id") \
  .agg(avg("days_between").alias("avg_days_between_orders"))
```

Step-by-Step Explanation:

- Use lag() to access the previous order date.
- Calculate the difference in days.
- Average the difference per customer.

Question 74

Problem Explanation:
Find users who placed more than one order in a single day.

PySpark Code:

```
from pyspark.sql.functions import count
multi_orders = orders_df.groupBy("user_id", "order_date") \
  .agg(count("*").alias("order_count")) \
  .filter(col("order_count") > 1)
```

Step-by-Step Explanation:

- Group by user_id and order_date.
- Count the number of orders and filter for more than one.

Question 75

Problem Explanation:
Perform an upsert using merge() in Delta Lake to update or insert customer details.

PySpark Code:

```
from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark, "/delta/customers")

updates_df = spark.read.format("parquet").load("/data/new_customers")

delta_table.alias("target").merge(
    updates_df.alias("source"),
    "target.customer_id = source.customer_id"
).whenMatchedUpdateAll() \
 .whenNotMatchedInsertAll() \
 .execute()
```

Step-by-Step Explanation:

- Use Delta Lake merge() for transactional updates/inserts.
- Match on customer_id, update if matched, insert if not.

Question 76

Problem Explanation:
Implement a Type 2 Slowly Changing Dimension (SCD) in Delta Lake using is_current and date columns.

PySpark Code:

```
from pyspark.sql.functions import current_date, lit

new_data = updates_df.withColumn("is_current", lit(True)) \
            .withColumn("start_date", current_date()) \
            .withColumn("end_date", lit(None))

delta_table = DeltaTable.forPath(spark, "/delta/dim_customers")
```

```
# Close existing records
delta_table.alias("target").merge(
    new_data.alias("source"),
    "target.customer_id = source.customer_id AND target.is_current = true"
).whenMatchedUpdate(set={"is_current": lit(False), "end_date": current_date()}) \
 .whenNotMatchedInsertAll() \
 .execute()
```

Step-by-Step Explanation:

- Updates previous records by setting is_current = False.
- Inserts new row for each change with updated flags and timestamps.

Question 77

Problem Explanation:
Use Delta Lake's time travel to query a previous version of a dataset.

PySpark Code:

```
historical_df = spark.read.format("delta") \
    .option("versionAsOf", 5) \
    .load("/delta/sales_data")
```

Step-by-Step Explanation:

- Uses versionAsOf to read data from version 5.
- Useful for auditing and debugging.

Question 78

Problem Explanation:
Write a batch job that overwrites only affected partitions in a Delta table.

PySpark Code:

```
affected_partitions = updated_df.select("region").distinct().rdd.flatMap(lambda x: x).collect()
for region in affected_partitions:
  partition_data = updated_df.filter(col("region") == region)
  partition_data.write.format("delta").mode("overwrite") \
    .option("replaceWhere", f"region = '{region}'") \
    .save("/delta/sales")
```

Step-by-Step Explanation:

- Gets list of partitions to overwrite.
- Overwrites specific partitions using replaceWhere.

Question 79

Problem Explanation:
Compact small files in Delta Lake using an OPTIMIZE-like logic in PySpark.

PySpark Code:

```
from delta.tables import DeltaTable
df = spark.read.format("delta").load("/delta/iot_data")
df.coalesce(1).write.option("dataChange", "false").format("delta") \
 .mode("overwrite").option("replaceWhere", "1=1").save("/delta/iot_data")
```

Step-by-Step Explanation:

- Reads all data, merges it into a single partition using coalesce(1).
- Overwrites the data without triggering unnecessary logs (dataChange = false).

Question 80

Problem Explanation:
Identify rows with schema drift or mismatched data types.

PySpark Code:

```
from pyspark.sql.functions import col

invalid_rows = df.filter(~col("age").cast("int").isNotNull())
```

Step-by-Step Explanation:

- Tries to cast the column.
- Filters out rows where casting fails — likely schema drift.

Question 81

Problem Explanation:
Validate data ranges for numeric columns and log outliers separately.

PySpark Code:

```
valid = df.filter((col("salary") >= 30000) & (col("salary") <= 200000))

outliers = df.filter((col("salary") < 30000) | (col("salary") > 200000))
```

Step-by-Step Explanation:

- Separate valid vs out-of-range salary data.
- Can store outliers in a "quarantine" path for review.

Question 82

Problem Explanation:
Detect duplicated rows across ingestion batches using hash logic.

PySpark Code:

```
from pyspark.sql.functions import sha2, concat_ws

df_with_hash = df.withColumn("hash_id", sha2(concat_ws("||", *df.columns), 256))

duplicates = df_with_hash.groupBy("hash_id").count().filter("count > 1")
```

Step-by-Step Explanation:

- Creates a unique hash per row.
- Groups by hash to detect duplicates across batches.

Question 83

Problem Explanation:
Create a column-level profiling report (min, max, null %, distinct count).

PySpark Code:

```
from pyspark.sql.functions import count, countDistinct, isnan, when

profile_df = df.agg(

  count("*").alias("total_rows"),

  *[

    countDistinct(c).alias(f"{c}_distinct") for c in df.columns

  ],

  *[

    (count(when(col(c).isNull() | isnan(c), c)) / count("*")).alias(f"{c}_null_pct") for c in
df.columns

  ]

)
```

Step-by-Step Explanation:

- Generates metrics per column: distinct count, null %, etc.
- Helps in understanding data distribution and health.

Question 84

Problem Explanation:
Quarantine bad data rows into a separate path instead of failing the pipeline.

PySpark Code:

```
valid_data = df.filter("age IS NOT NULL AND age > 0")

invalid_data = df.subtract(valid_data)

valid_data.write.parquet("/mnt/clean/data")

invalid_data.write.parquet("/mnt/quarantine/data")
```

Step-by-Step Explanation:

- Separates invalid rows proactively.
- Ensures pipeline continues without interruption, logs issues for later fix.

Question 85

Problem Explanation:
Identify rows with schema drift or mismatched data types, especially when ingesting dynamic or semi-structured sources.

Optimized PySpark Code:

```
from pyspark.sql.functions import col

# Detect rows where numeric field contains non-numeric values

invalid_rows = df.filter(~col("age").cast("int").isNotNull())
```

Step-by-step Explanation:

- Cast the column to its expected data type.
- Filter out rows where cast fails (resulting in null), indicating mismatched types.

Question 86

Problem Explanation:
Validate numeric ranges (e.g., salary, age), and log outliers without stopping the pipeline.

Optimized PySpark Code:

```
valid = df.filter((col("salary") >= 30000) & (col("salary") <= 200000))

outliers = df.subtract(valid)

# Write outliers for logging

outliers.write.parquet("/quarantine/outliers")
```

Step-by-step Explanation:

- Apply range checks using filter.
- Subtract valid rows to get outliers, write them to quarantine.

Question 87

Problem Explanation:
Detect duplicate rows across ingestion batches using row-level hashing.

Optimized PySpark Code:

```
from pyspark.sql.functions import sha2, concat_ws

df_with_hash = df.withColumn("row_hash", sha2(concat_ws("||", *df.columns), 256))

duplicates = df_with_hash.groupBy("row_hash").count().filter("count > 1")
```

Step-by-step Explanation:

- Create hash for each row.
- Group by hash to find duplicates appearing more than once.

Question 88

Problem Explanation:
Generate column-level data profiling: null %, min, max, and unique count.

Optimized PySpark Code:

```
from pyspark.sql.functions import count, countDistinct, isnan, when, col, min, max

profile = df.agg(

   *[countDistinct(c).alias(f"{c}_distinct") for c in df.columns],

   *[count(when(col(c).isNull() | isnan(c), c)).alias(f"{c}_nulls") for c in df.columns],

   *[min(c).alias(f"{c}_min") for c in df.columns if str(df.schema[c].dataType) in ['IntegerType',
'DoubleType']],

   *[max(c).alias(f"{c}_max") for c in df.columns if str(df.schema[c].dataType) in ['IntegerType',
'DoubleType']]

)
```

Step-by-step Explanation:

- Computes distinct count, nulls, min, and max for all columns in one pass.

Question 89

Problem Explanation:
Quarantine bad rows (with nulls or invalid values) into a separate folder instead of failing the job.

Optimized PySpark Code:

```
valid_rows = df.filter("age IS NOT NULL AND salary IS NOT NULL")

invalid_rows = df.subtract(valid_rows)

valid_rows.write.parquet("/clean/data")

invalid_rows.write.parquet("/quarantine/bad_data")
```

Step-by-step Explanation:

- Define rules for "good" data.
- Subtract to find bad rows and write separately.

Question 90

Problem Explanation:
Detect skewed joins and apply salting to redistribute skewed keys.

Optimized PySpark Code:

```
from pyspark.sql.functions import monotonically_increasing_id, rand


# Add salt key

left_df = left_df.withColumn("salt", (rand() * 10).cast("int"))

right_df = right_df.crossJoin(spark.range(10).toDF("salt"))


# Join on both key and salt

salted_join = left_df.join(right_df, on=["key", "salt"])
```

Step-by-step Explanation:

- Randomize and expand skewed dataset.
- Cross join small side with salt range to distribute load.

Question 91

Problem Explanation:
Use coalesce() before writing to reduce output file count.

Optimized PySpark Code:

```
df.coalesce(10).write.mode("overwrite").parquet("/output/path")
```

Step-by-step Explanation:

- Combines data into fewer partitions (10 in this case) before writing.
- Reduces small file problem and improves downstream performance.

Question 92

Problem Explanation:
Persist intermediate results using .cache() and measure performance gain.

Optimized PySpark Code:

```
stage1 = df.filter("status = 'active'").cache()

stage1.count()  # Materialize cache


# Use in multiple downstream operations

result1 = stage1.groupBy("region").count()

result2 = stage1.select("id", "timestamp")
```

Step-by-step Explanation:

- Cache is useful when a DataFrame is reused.
- Avoids recomputation of the same lineage.

Question 93

Problem Explanation:
Estimate DataFrame size and tune number of partitions.

Optimized PySpark Code:

```
num_partitions = df.rdd.getNumPartitions()

approx_size = df.rdd.map(lambda x: len(str(x))).reduce(lambda x, y: x + y)


# Repartition if needed

df = df.repartition(approx_size // (128 * 1024 * 1024))
```

Step-by-step Explanation:

- Calculates approximate in-memory size.
- Adjusts partition count to keep them evenly sized.

Question 94

Problem Explanation:
Optimize wide transformation chains with checkpoint().

Optimized PySpark Code:

```
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

df = df.checkpoint(eager=True)
```

Step-by-step Explanation:

- Useful in long lineage pipelines with joins, filters, aggregations.
- Reduces recomputation and lineage overhead.

Question 95

Problem Explanation:
Read a large partitioned Parquet dataset and use partition pruning.

Optimized PySpark Code:

```
df = spark.read.parquet("/data/sales")

filtered_df = df.filter("year = 2023 AND month = 6")
```

Step-by-step Explanation:

- Filtering on partition columns allows Spark to skip reading irrelevant partitions.

Question 96

Problem Explanation:
Write data to partitioned folders by year/month/day.

Optimized PySpark Code:

```
df.write.partitionBy("year", "month", "day").parquet("/output/partitioned_sales")
```

Step-by-step Explanation:

- Automatically writes data to directory paths like /year=2023/month=07/day=01/.

Question 97

Problem Explanation:
Convert a CSV ingestion pipeline to Parquet with schema inference.

Optimized PySpark Code:

```
df = spark.read.option("header", True).option("inferSchema", True).csv("/input/sales.csv")

df.write.parquet("/output/sales_parquet")
```

Step-by-step Explanation:

- Uses built-in schema inference from CSV headers.
- Writes in optimized Parquet format.

Question 98

Problem Explanation:
Write different columns of the same DataFrame to different file formats.

Optimized PySpark Code:

```
df.select("id", "name").write.json("/output/users_json")

df.select("id", "transactions").write.parquet("/output/transactions_parquet")
```

Step-by-step Explanation:

- Useful when separate consumers need different formats or slices of data.

Question 99

Problem Explanation:
Merge multiple small Parquet files into fewer optimized files.

Optimized PySpark Code:

```
df = spark.read.parquet("/input/small_files/")

df.coalesce(5).write.mode("overwrite").parquet("/output/merged_files")
```

Step-by-step Explanation:

- Reads fragmented data and rewrites into 5 consolidated files using coalesce.

Question 100

Problem Explanation:
Read data from Kafka, parse JSON payloads, and write to Delta Lake.

Optimized PySpark Code:

```
from pyspark.sql.functions import from_json, col

from pyspark.sql.types import StructType, StringType


schema = StructType().add("user_id", StringType()).add("event_type", StringType())


df = spark.readStream.format("kafka") \

  .option("subscribe", "events") \

  .load()


json_df = df.selectExpr("CAST(value AS STRING)") \

  .select(from_json(col("value"), schema).alias("data")).select("data.*")


json_df.writeStream.format("delta").option("checkpointLocation", "/chkpt") \

  .start("/output/delta_stream")
```

Step-by-step Explanation:

- Reads Kafka stream.
- Parses JSON payload.
- Writes to Delta format with checkpointing for fault tolerance.