Given a Teams table with columns TeamID (integer) and Members (comma-separated string of names), write a query to calculate and display the total number of members in each team

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, split, size

# Create Spark session
spark = SparkSession.builder \
   .appName("TeamMemberCount") \
   .getOrCreate()

# Sample data
data = [
   (1, "Chris, Evan, Marty, Eva"),
   (2, "Jake, Olivia"),
   (3, "Sophia, Liam, Noah, Emma"),
   (4, "Ava, Lucas, Mia, Ethan, Amelia"),
   (5, "Benjamin, Charlotte"),
   (6, "Harper, Henry, Evelyn, Daniel, Ella"),
   (7, "Michael, Emily, Alexander"),
   (8, "James, Abigail, William, Isabella, Jack, Grace"),
   (9, "Sebastian, Chloe"),
   (10, "David, Lily, Samuel, Madison")]

# Create DataFrame
teams_df = spark.createDataFrame(data, ["TeamID", "Members"])

# Calculate member count
result_df = teams_df.withColumn("MemberCount", size(split(col("Members"), ",\\s*")))

# Show results
result_df.select("TeamID", "MemberCount").show()

# Register DataFrame as a temporary view
teams_df.createOrReplaceTempView("Teams")

# Execute SparkSQL query
spark.sql("""
SELECT TeamID, SIZE(SPLIT(Members, ',\\s*')) AS MemberCount FROM Teams""").show()
-------------------------------------------------------------
```

Write a SQL query to retrieve the most frequently ordered item(s) for each date from a given orders table. If multiple items have the highest order count on a particular date, include all such items in the result."

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, max, dense_rank
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder \
   .appName("FrequentItemsAnalysis") \
   .getOrCreate()
```

```
# Sample data
data = [
   ('2024-03-01', 'Apple'),
   ('2024-03-01', 'Banana'),
   ('2024-03-01', 'Apple'),
   ('2024-03-02', 'Orange'),
   ('2024-03-02', 'Orange'),
   ('2024-03-02', 'Mango'),
   ('2024-03-03', 'Banana'),
   ('2024-03-03', 'Banana'),
   ('2024-03-03', 'Mango'),
   ('2024-03-03', 'Mango')
]

# Create DataFrame
orders_df = spark.createDataFrame(data, ["order_date", "item"])

# Solution using Window Functions
window_spec = Window.partitionBy("order_date").orderBy(col("order_count").desc())

result_df = (orders_df
   .groupBy("order_date", "item")
   .agg(count("*").alias("order_count"))
   .withColumn("rank", dense_rank().over(window_spec))
   .filter(col("rank") == 1)
   .select("order_date", "item", "order_count")
   .orderBy("order_date", "item"))

# Show results
result_df.show()

# Register DataFrame as a temporary view
orders_df.createOrReplaceTempView("orders")

# Execute SparkSQL query with window function

spark.sql("""
WITH ranked_items AS (
SELECT order_date, item, COUNT(*) AS order_count, DENSE_RANK() OVER (PARTITION
BY order_date ORDER BY COUNT(*) DESC) AS rank FROM orders GROUP BY order_date,
item)
SELECT order_date, item, order_count FROM ranked_items WHERE rank = 1 ORDER BY
order_date, item""").show()
```

Write a pyspark and sparkSQL query to display the entire employee reporting hierarchy using a recursive CTE, showing each employee's level in the hierarchy.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, when

# Create Spark session
spark = SparkSession.builder \
   .appName("EmployeeHierarchy") \
   .getOrCreate()
```

```
# Sample data
data = [
   (1, 'Alice', 'CEO', None),
   (2, 'Bob', 'CTO', 1),
   (3, 'Charlie', 'CFO', 1),
   (4, 'David', 'Engineering Manager', 2),
   (5, 'Eve', 'Finance Manager', 3),
   (6, 'Frank', 'Software Engineer', 4),
   (7, 'Grace', 'Software Engineer', 4),
   (8, 'Hannah', 'Accountant', 5)
]

# Create DataFrame
employees_df = spark.createDataFrame(data, ["id", "name", "designation",
"manager_id"])

# Register DataFrame as a temporary view for recursive query
employees_df.createOrReplaceTempView("employees")

# PySpark doesn't natively support recursive CTEs, so we'll use an iterative approach
max_levels = 10  # Assuming max hierarchy depth

# Initialize with level 0 (CEO)
hierarchy_df = employees_df.filter(col("manager_id").isNull()) \
   .withColumn("level", lit(0)) \
   .withColumn("path", lit("")) \
   .select("id", "name", "designation", "manager_id", "level", "path")

for level in range(1, max_levels + 1):
   current_level_df = spark.sql(f"""
      SELECT e.id, e.name, e.designation, e.manager_id, {level} as level,
CONCAT(h.path, ' > ', e.name) as path FROM employees e JOIN hierarchy h ON
e.manager_id = h.id WHERE h.level = {level - 1}""")

   if current_level_df.count() == 0:
      break

   hierarchy_df = hierarchy_df.union(current_level_df)

# Show the complete hierarchy
hierarchy_df.orderBy("level", "name").show(truncate=False)

# SparkSQL solution (works in Databricks and Spark 3.4+ with recursive CTE support)
spark.sql("""
   WITH RECURSIVE employee_hierarchy AS (
      -- Base case: top-level employees (CEO)
    SELECT id, name, designation, manager_id, 0 AS level, CAST(name AS STRING) AS
hierarchy_path FROM employees WHERE  manager_id IS NULL
    UNION ALL
    -- Recursive case: employees with managers
     SELECT e.id, e.name, e.designation, e.manager_id, eh.level + 1 AS level,
CONCAT(eh.hierarchy_path, ' > ', e.name) AS hierarchy_path FROM employees e
JOIN  employee_hierarchy eh ON e.manager_id = eh.id)
   SELECT
      id,
      name,
      designation,
      manager_id,
      level,
      hierarchy_path,
      CONCAT(REPEAT('    ', level), name) AS visual_hierarchy FROM
employee_hierarchy ORDER BY hierarchy_path""").show(truncate=False)
```

Write a pyspark and SparkSQL query to retrieve the first and last order for each customer from the orders table

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, first, last
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder \
    .appName("FirstLastOrders") \
    .getOrCreate()

# Sample data
data = [
    (1, 101, '2024-01-05'),
    (2, 101, '2024-03-15'),
    (3, 101, '2024-05-20'),
    (4, 102, '2024-02-10'),
    (5, 102, '2024-04-25'),
    (6, 102, '2024-06-30'),
    (7, 103, '2024-01-01'),
    (8, 103, '2024-02-18'),
    (9, 103, '2024-03-25')]

# Create DataFrame
orders_df = spark.createDataFrame(data, ["order_id", "customer_id", "order_date"])

# Window function approach
window_spec = Window.partitionBy("customer_id").orderBy("order_date")

result_df = orders_df.withColumn("row_num", row_number().over(window_spec)) \
    .withColumn("reverse_row_num",
row_number().over(window_spec.orderBy(col("order_date").desc()))) \
    .filter((col("row_num") == 1) | (col("reverse_row_num") == 1)) \
    .groupBy("customer_id") \
    .agg(
        first("order_id").alias("first_order_id"),
        first("order_date").alias("first_order_date"),
        last("order_id").alias("last_order_id"),
        last("order_date").alias("last_order_date")).orderBy("customer_id")

# Show results
result_df.show()

# Register DataFrame as a temporary view
orders_df.createOrReplaceTempView("orders")
```

```
# Execute SparkSQL query
spark.sql("""
    WITH ranked_orders AS (
        SELECT
            customer_id,
            order_id,
            order_date,
            ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date) AS
first_rank,
            ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC)
AS last_rank FROM orders)
    SELECT
        customer_id,
        MAX(CASE WHEN first_rank = 1 THEN order_id END) AS first_order_id,
        MAX(CASE WHEN first_rank = 1 THEN order_date END) AS first_order_date,
        MAX(CASE WHEN last_rank = 1 THEN order_id END) AS last_order_id,
        MAX(CASE WHEN last_rank = 1 THEN order_date END) AS last_order_date FROM
ranked_orders WHERE first_rank = 1 OR last_rank = 1 GROUP BY customer_id ORDER BY
customer_id""").show()
```

How can all 10 teams play a total of 14 league matches each (resulting in 70 league matches overall), where each team faces 5 opponents twice and the remaining 4 opponents only once

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, row_number
from pyspark.sql.window import Window

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

# Create teams DataFrame
teams_data = [
    ("Mumbai Indians", "A"),
    ("Chennai Super Kings", "B"),
    ("Kolkata Knight Riders", "A"),
    ("Sun Risers Hyderabad", "B"),
    ("Rajesthan Royals", "A"),
    ("Royal Challengers Bangalore", "B"),
    ("Delhi Capitals", "A"),
    ("Punjab Kings", "B"),
    ("Lucknow Super Giants", "A"),
    ("Gujarat Titans", "B")
]
teams_df = spark.createDataFrame(teams_data, ["team", "group_name"])

# Generate intra-group matches (home and away)
intra_group = teams_df.alias("a").join(
    teams_df.alias("b"),
    (col("a.group_name") == col("b.group_name")) & (col("a.team") <
col("b.team"))).select(
    col("a.team").alias("team1"),
    col("b.team").alias("team2"),
    col("a.group_name").alias("match_group")
)
```

```
# Duplicate for home/away matches
intra_group_matches = intra_group.unionAll(
    intra_group.select(
        col("team2").alias("team1"),
        col("team1").alias("team2"),
        col("match_group")))

# Generate inter-group matches
inter_group = teams_df.alias("a").join(
    teams_df.alias("b"),
    col("a.group_name") != col("b.group_name")).select(
    col("a.team").alias("team1"),
    col("b.team").alias("team2"),
    lit("A vs B").alias("match_type"))

# Select 5 inter-group matches per team
window_spec = Window.partitionBy("team1").orderBy("team2")
selected_inter_group = inter_group.withColumn("rn",
row_number().over(window_spec)).filter(col("rn") <= 5).select("team1", "team2",
"match_type")

# Combine all matches
full_schedule = intra_group_matches.unionAll(
    selected_inter_group.select(
        col("team1"),
        col("team2"),
        col("match_type").alias("match_group")))

# Show the schedule
full_schedule.orderBy("team1", "team2").show(100)

%sql

WITH match_counts AS (
    SELECT team1 AS team, COUNT(*) AS matches
    FROM full_schedule
    GROUP BY team1
    UNION ALL
    SELECT team2 AS team, COUNT(*) AS matches
    FROM full_schedule
    GROUP BY team2)
SELECT team, SUM(matches) AS total_matches FROM match_counts GROUP BY team
ORDER BY team;
--------------------------------------------------------------------------------------------------------------------------------------------------------
```

How many cases have reached each stage of completion (Stage 1 to Stage 5) for each center?

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, when

# Create Spark session
spark = SparkSession.builder.appName("CaseProgressAnalysis").getOrCreate()

# Assuming the CaseProgress table is already available as a DataFrame
# If not, you would load it like:
```

```
# case_progress_df = spark.read.jdbc(...) or spark.read.table("CaseProgress")

# Calculate stage counts per center
result_df = case_progress_df.groupBy("Center_ID").agg(
    count(when(col("Stage1").isNotNull(), 1)).alias("Stage1_Count"),
    count(when(col("Stage2").isNotNull(), 1)).alias("Stage2_Count"),
    count(when(col("Stage3").isNotNull(), 1)).alias("Stage3_Count"),
    count(when(col("Stage4").isNotNull(), 1)).alias("Stage4_Count"),
    count(when(col("Stage5").isNotNull(), 1)).alias("Stage5_Count"),
    count("*").alias("Total_Cases")).orderBy("Center_ID")

# Show results
result_df.show()
```

```
SELECT
    Center_ID,
    COUNT(CASE WHEN Stage1 IS NOT NULL THEN 1 END) AS Stage1_Count,
    COUNT(CASE WHEN Stage2 IS NOT NULL THEN 1 END) AS Stage2_Count,
    COUNT(CASE WHEN Stage3 IS NOT NULL THEN 1 END) AS Stage3_Count,
    COUNT(CASE WHEN Stage4 IS NOT NULL THEN 1 END) AS Stage4_Count,
    COUNT(CASE WHEN Stage5 IS NOT NULL THEN 1 END) AS Stage5_Count,
    COUNT(*) AS Total_Cases FROM CaseProgress GROUP BY Center_ID ORDER BY
Center_ID;
```

Learn how to calculate monthly differences in credit card issuance using real-world data from JPMorgan Chase.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lag, round
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.appName("CreditCardAnalysis").getOrCreate()

# Window specification for previous month comparison
window_spec = Window.partitionBy("card_name").orderBy("issue_year", "issue_month")

# Calculate monthly differences
result_df = credit_card_issuance_df.withColumn(
    "previous_month_amount",
    lag("issued_amount",
1).over(window_spec)).withColumn("monthly_difference",col("issued_amount") -
col("previous_month_amount")).withColumn(
    "percentage_change",
    round((col("monthly_difference") / col("previous_month_amount")) * 100, 2)).filter(
    col("previous_month_amount").isNotNull()).select(
    "card_name",
    "issue_year",
    "issue_month",
    "issued_amount",
    "previous_month_amount",
    "monthly_difference",
   "percentage_change").orderBy("card_name", "issue_year", "issue_month")

# Show results
result_df.show()
```

```
%sql
SELECT
   card_name,
   issue_year,
   issue_month,
   issued_amount AS current_month_amount,LAG(issued_amount, 1) OVER (PARTITION
BY card_name ORDER BY issue_year, issue_month) AS previous_month_amount,
   issued_amount - LAG(issued_amount, 1) OVER (PARTITION BY card_name ORDER BY
issue_year, issue_month) AS monthly_difference,
   ROUND((issued_amount - LAG(issued_amount, 1) OVER (PARTITION BY card_name
ORDER BY issue_year, issue_month))
   LAG(issued_amount, 1) OVER (PARTITION BY card_name ORDER BY issue_year,
issue_month) * 100, 2) AS percentage_change FROM credit_card_issuance ORDER BY
card_name, issue_year, issue_month;
```

----

calculate the average transaction amount per year for each client for the years 2018 to 2022.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year, avg, round, count

# Create Spark session
spark = SparkSession.builder.appName("TransactionAnalysis").getOrCreate()

# Sample data (in a real scenario, you'd read from a source)
data = [
   (1, 269, '2018-08-15', 500.0),
   (2, 478, '2018-11-25', 400.0),
   (3, 269, '2019-01-05', 1000.0),
   (4, 123, '2020-10-20', 600.0),
   (5, 478, '2021-07-05', 700.0),
   (6, 123, '2022-03-05', 900.0)]

# Create DataFrame
columns = ["transaction_id", "user_id", "transaction_date", "transaction_amount"]
transactions_df = spark.createDataFrame(data, columns)

# Convert string date to date type
transactions_df = transactions_df.withColumn(
   "transaction_date",
   col("transaction_date").cast("date"))

# Calculate average transaction amount per year per user
result_df = transactions_df \
   .filter((year("transaction_date") >= 2018) & (year("transaction_date") <= 2022)) \
   .groupBy("user_id", year("transaction_date").alias("transaction_year")) \
   .agg(round(avg("transaction_amount"),
2).alias("avg_transaction_amount"),count("*").alias("transaction_count")).orderBy("user_
id", "transaction_year")

# Show results
result_df.show()

%sql
SELECT
    user_id,
    EXTRACT(YEAR FROM transaction_date) AS transaction_year,
    ROUND(AVG(transaction_amount), 2) AS avg_transaction_amount,
    COUNT(*) AS transaction_count FROM transactions WHERE  EXTRACT(YEAR FROM
transaction_date) BETWEEN 2018 AND 2022 GROUP BY user_id,  EXTRACT(YEAR FROM
transaction_date) ORDER BY user_id,transaction_year;
```

----

calculate the 3-month moving average of sales revenue for each month using order data. The moving average for each month should be calculated as the average of the current month's sales and the sales from the previous two months.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, date_format, sum, avg, lag
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.appName("MovingAverage").getOrCreate()

# Read data (in a real scenario, you'd read from source)
customers_data = [(1, 'John', '2023-01-10'), (2, 'Simmy', '2023-02-15'), (3, 'Iris',
'2023-03-20')]
orders_data = [
    (1, 1, '2023-01-05', 100.00),
    (2, 2, '2023-02-14', 150.00),
    (3, 1, '2023-02-28', 200.00),
    (4, 3, '2023-03-22', 300.00),
    (5, 2, '2023-04-10', 250.00),
    (6, 1, '2023-05-15', 400.00),
    (7, 3, '2023-06-10', 350.00)]

# Create DataFrames
customers_df = spark.createDataFrame(customers_data, ["Customer_id", "Name",
"Join_Date"])
orders_df = spark.createDataFrame(orders_data, ["Order_id", "Customer_id",
"Order_Date", "Amount"])

# Convert string dates to date type
orders_df = orders_df.withColumn("Order_Date", col("Order_Date").cast("date"))

# Step 1: Calculate monthly sales
monthly_sales = orders_df.groupBy(date_format("Order_Date",
"yyyy-MM").alias("month")).agg(sum("Amount").alias("monthly_revenue")).orderBy("mon
th")

# Step 2: Calculate 3-month moving average using window functions
window_spec = Window.orderBy("month").rowsBetween(-2, 0)
```

```
moving_avg =
monthly_sales.withColumn("3_month_moving_avg",avg("monthly_revenue").over(window_spec))

# Show results
moving_avg.show()

%sql

# Register DataFrames as temporary views
orders_df.createOrReplaceTempView("orders")
customers_df.createOrReplaceTempView("customers")

# Execute SparkSQL query
result = spark.sql("""
WITH monthly_sales AS (
SELECT date_format(Order_Date, 'yyyy-MM') AS month, SUM(Amount) AS
monthly_revenue FROM orders GROUP BY date_format(Order_Date, 'yyyy-MM')),
moving_averages AS (
SELECT month, monthly_revenue, AVG(monthly_revenue) OVER (ORDER BY month
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS 3_month_moving_avg FROM
monthly_sales)
SELECT month,monthly_revenue,ROUND(3_month_moving_avg, 2) AS
3_month_moving_avg FROM moving_averages ORDER BY month""")

result.show()
```

----

calculating the average monthly revenue from each sector using financial transaction data.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, date_format, avg, round

# Create Spark session
spark = SparkSession.builder.appName("SectorRevenueAnalysis").getOrCreate()

# Sample data
transactions_data = [
    (101, 1, '2020-01-15', 5000.00),
    (102, 2, '2020-01-20', 8500.00),
    (103, 1, '2020-02-10', 4500.00),
    (104, 3, '2020-02-20', 9900.00),
    (105, 2, '2020-02-25', 7500.00)
]

sectors_data = [
    (1, 'Technology'),
    (2, 'Healthcare'),
    (3, 'Technology')
]

# Create DataFrames
transactions_df = spark.createDataFrame(transactions_data,
                              ["transaction_id", "company_id", "transaction_date", "revenue"])
```

```
sectors_df = spark.createDataFrame(sectors_data, ["company_id", "sector"])

# Convert string dates to date type
transactions_df = transactions_df.withColumn("transaction_date",
col("transaction_date").cast("date"))

# Join with sectors to get sector information
joined_df = transactions_df.join(sectors_df, "company_id")

# Calculate average monthly revenue by sector
result_df = joined_df.groupBy("sector",
    date_format("transaction_date", "yyyy-MM").alias("month")).agg(
    round(avg("revenue"), 2).alias("avg_monthly_revenue"),
    count("*").alias("transaction_count")).orderBy("sector", "month")

# Show results
result_df.show()

# Register DataFrames as temporary views
transactions_df.createOrReplaceTempView("transactions")
sectors_df.createOrReplaceTempView("sectors")

# Execute SparkSQL query
result = spark.sql("""
    SELECT
        s.sector,
        date_format(t.transaction_date, 'yyyy-MM') AS month,
        ROUND(AVG(t.revenue), 2) AS avg_monthly_revenue,
        COUNT(*) AS transaction_count FROM transactions t JOIN sectors s ON
t.company_id = s.company_id GROUP BY s.sector, date_format(t.transaction_date,
'yyyy-MM') ORDER BY s.sector, month""")

result.show()
------------------------------------------------------------------------
```

Finding Managers with ≥5 Direct Reports in Departments with >10 Employees

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count

# Create Spark session
spark = SparkSession.builder.appName("ManagerAnalysis").getOrCreate()

# Sample data
employee_data = [
    (1, 'John', 'HR', None),
    (2, 'Bob', 'HR', 1),
    (3, 'Olivia', 'HR', 1),
    (4, 'Emma', 'Finance', None),
    (5, 'Sophia', 'HR', 1),
    (6, 'Mason', 'Finance', 4),
    (7, 'Ethan', 'HR', 1),
    (8, 'Ava', 'HR', 1),
    (9, 'Lucas', 'HR', 1),
    (10, 'Isabella', 'Finance', 4),
    (11, 'Harper', 'Finance', 4),
    (12, 'Hemla', 'HR', 3),
    (13, 'Aisha', 'HR', 2),
    (14, 'Himani', 'HR', 2),
    (15, 'Lily', 'HR', 2)]

# Create DataFrame
employee_df = spark.createDataFrame(employee_data, ["id", "name", "department",
"managerId"])

# Calculate department sizes (>10 employees)
department_sizes = employee_df.groupBy("department") \
    .agg(count("*").alias("total_employees")) \
    .filter(col("total_employees") > 10)

# Calculate managers with ≥5 direct reports
manager_reports = employee_df.alias("e1") \
    .join(employee_df.alias("e2"),
        col("e1.id") == col("e2.managerId")) \
    .groupBy(
        col("e1.id").alias("manager_id"),
        col("e1.name").alias("manager_name"),
        col("e1.department")) \
    .agg(count("*").alias("direct_reports_count")) \
    .filter(col("direct_reports_count") >= 5)

# Join results
result = manager_reports.join(department_sizes, "department") \
    .select(
        "manager_id",
        "manager_name",
        "department",
        "direct_reports_count",
        "total_employees") \
    .orderBy(col("direct_reports_count").desc())

# Show results
result.show()

%sql

WITH department_sizes AS (
SELECT department, COUNT(*) AS total_employees FROM Employee GROUP BY
department HAVING COUNT(*) > 10),
manager_reports AS (
    SELECT
        e1.id AS manager_id,
        e1.name AS manager_name,
        e1.department,
        COUNT(e2.id) AS direct_reports_count FROM
        Employee e1 JOIN Employee e2 ON e1.id = e2.managerId GROUP BY e1.id,
e1.name, e1.department HAVING COUNT(e2.id) >= 5)
SELECT
    m.manager_id,
    m.manager_name,
    m.department,
    m.direct_reports_count,
    d.total_employees AS department_size FROM manager_reports m JOIN
department_sizes d ON m.department = d.department ORDER BY
m.direct_reports_count DESC;
```

Identifying Numbers Appearing at Least Three Times Consecutively

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, row_number, count
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.appName("ConsecutiveNumbers").getOrCreate()

# Sample data
logs_data = [
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 2),
    (5, 1),
    (6, 2),
    (7, 2)]

# Create DataFrame
logs_df = spark.createDataFrame(logs_data, ["id", "num"])

# Window specification
window_spec = Window.partitionBy("num").orderBy("id")

# Calculate consecutive groups
consecutive_groups = logs_df.withColumn("grp",col("id") -
row_number().over(window_spec))

# Count consecutive occurrences
result = consecutive_groups.groupBy("num", "grp") \
    .agg(count("*").alias("consecutive_count")) \
    .filter(col("consecutive_count") >= 3) \
    .select("num").distinct() \
    .orderBy("num")

# Show results
result.show()
```

```
%sql

WITH consecutive_groups AS (
SELECT num,id,id - ROW_NUMBER() OVER (PARTITION BY num ORDER BY id) AS grp
FROM logs),
consecutive_counts AS (
SELECT num, grp, COUNT(*) AS consecutive_count FROM consecutive_groups GROUP
BY num, grp HAVING COUNT(*) >= 3)
SELECT DISTINCT num FROM consecutive_counts ORDER BY num;
```

----

finding the Net Present Value (NPV) for queries from two tables

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

# Create Spark session
spark = SparkSession.builder.appName("NPVCalculation").getOrCreate()

# Sample data
npv_data = [
   (1, 2018, 100),
   (7, 2020, 30),
   (13, 2019, 40),
   (1, 2019, 113),
   (2, 2008, 121),
   (3, 2009, 12),
   (11, 2020, 99),
   (7, 2019, 0)
]

queries_data = [
   (1, 2019),
   (2, 2008),
   (3, 2009),
   (7, 2018),
   (7, 2019),
   (7, 2020),
   (13, 2019)
]

# Create DataFrames
npv_df = spark.createDataFrame(npv_data, ["id", "year", "npv"])
queries_df = spark.createDataFrame(queries_data, ["id", "year"])

# Calculate NPV for queries
result_df = queries_df.join(
   npv_df,
   (queries_df.id == npv_df.id) & (queries_df.year == npv_df.year),
   "left").select(
   queries_df.id,
   queries_df.year,
   coalesce(npv_df.npv, lit(0)).alias("npv")).orderBy("id", "year")

# Show results
result_df.show()

# Register DataFrames as temporary views
npv_df.createOrReplaceTempView("npv")
queries_df.createOrReplaceTempView("queries")

# Execute SparkSQL query
result = spark.sql("""
SELECT q.id, q.year, COALESCE(n.npv, 0) AS npv FROM queries q LEFT JOIN npv n ON
q.id = n.id AND q.year = n.year ORDER BY q.id, q.year""")

result.show()
```

----

retrieve Walmart users' most recent transaction date, user ID, and the total number of products they purchased. Learn how to efficiently sort data in chronological order and calculate product counts based on user transactions.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, sum, max
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.appName("WalmartUserAnalysis").getOrCreate()

# Sample data
transaction_data = [
   (3673, 123, 68.90, '2022-07-08 10:00:00'),
   (9623, 123, 274.10, '2022-07-08 10:00:00'),
   (1467, 115, 19.90, '2022-07-08 10:00:00'),
   (2513, 159, 25.00, '2022-07-08 10:00:00'),
   (1452, 159, 74.50, '2022-07-10 10:00:00'),
   (1452, 123, 74.50, '2022-07-10 10:00:00'),
   (9765, 123, 100.15, '2022-07-11 10:00:00'),
   (6536, 115, 57.00, '2022-07-12 10:00:00'),
   (7384, 159, 15.50, '2022-07-12 10:00:00'),
   (1247, 159, 23.40, '2022-07-12 10:00:00')]

# Create DataFrame
columns = ["product_id", "user_id", "spend", "transaction_date"]
transactions_df = spark.createDataFrame(transaction_data, columns)

# Convert string date to timestamp
transactions_df = transactions_df.withColumn(
   "transaction_date",
   col("transaction_date").cast("timestamp"))

# Calculate user statistics
result_df = transactions_df.groupBy("user_id").agg(
   max("transaction_date").alias("most_recent_date"),
   count("product_id").alias("total_products"),
   sum("spend").alias("total_spend")).orderBy(col("most_recent_date").desc())

# Show results
result_df.show()

%sql
WITH user_stats AS (
   SELECT
      user_id,
      MAX(transaction_date) AS most_recent_date,
      COUNT(product_id) AS total_products,
      SUM(spend) AS total_spend FROM transactions GROUP BY user_id)
SELECT
   user_id,
   most_recent_date,
   total_products,
   total_spend FROM user_stats ORDER BY most_recent_date DESC;
```

----

calculate the average rating for each restaurant for each month. We'll be using

Zomato's review data to calculate the average rating for each restaurant on a monthly basis.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year, month, avg, round, count

# Create Spark session
spark = SparkSession.builder.appName("RestaurantRatings").getOrCreate()

# Sample data
review_data = [
   (1001, 501, '2022-01-15', 101, 4),
   (1002, 502, '2022-01-20', 101, 5),
   (1003, 503, '2022-01-25', 102, 3),
   (1004, 504, '2022-01-15', 102, 4),
   (1005, 505, '2022-02-20', 101, 5),
   (1006, 506, '2022-02-26', 101, 4),
   (1007, 507, '2022-03-01', 101, 4),
   (1008, 508, '2022-03-05', 102, 2)]

# Create DataFrame
columns = ["review_id", "user_id", "submit_date", "restaurant_id", "rating"]
reviews_df = spark.createDataFrame(review_data, columns)

# Convert string date to date type
reviews_df = reviews_df.withColumn("submit_date", col("submit_date").cast("date"))

# Calculate monthly average ratings
result_df = reviews_df.groupBy(
   "restaurant_id",
   year("submit_date").alias("year"),
   month("submit_date").alias("month")).agg(
   round(avg("rating"), 2).alias("avg_rating"),
   count("*").alias("review_count")).orderBy(
   "restaurant_id",
   "year",
   "month")

# Show results
result_df.show()
```

```
%sql
SELECT
   restaurant_id,
   EXTRACT(YEAR FROM submit_date) AS year,
   EXTRACT(MONTH FROM submit_date) AS month,
   ROUND(AVG(rating), 2) AS avg_rating,
   COUNT(*) AS review_count FROM reviews GROUP BY restaurant_id, EXTRACT(YEAR
FROM submit_date),EXTRACT(MONTH FROM submit_date) ORDER BY restaurant_id,
year,month;
```

Our goal is to find the top 5 artists whose songs have appeared most frequently in the

Top 10 of the global music charts. We'll be using three tables: artists, songs, and global_song_rank.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, desc

# Create Spark session
spark = SparkSession.builder.appName("TopArtists").getOrCreate()

# Sample data
artists_data = [
    (101, 'Ed Sheeran', 'Warner Music Group'),
    (120, 'Drake', 'Warner Music Group'),
    (125, 'Bad Bunny', 'Rimas Entertainment'),
    (130, 'Lady Gaga', 'Interscope Records'),
    (140, 'Katy Perry', 'Capitol Records')
]

songs_data = [
    (55511, 101, 'Perfect'),
    (45202, 101, 'Shape of You'),
    (22222, 120, 'One Dance'),
    (19960, 120, 'Hotline Bling'),
    (33333, 125, 'Dákiti'),
    (44444, 125, 'Yonaguni'),
    (55555, 130, 'Bad Romance'),
    (66666, 130, 'Poker Face'),
    (99999, 140, 'Roar'),
    (101010, 140, 'Firework')
]

global_rank_data = [
    (1, 45202, 5),
    (3, 45202, 2),
    (1, 19960, 3),
    (9, 19960, 6),
    (1, 55511, 8),
    (5, 22222, 7),
    (2, 33333, 4),
    (4, 44444, 8),
    (6, 55555, 1),
    (7, 66666, 10),
    (5, 99999, 5)
]

# Create DataFrames
artists_df = spark.createDataFrame(artists_data, ["artist_id", "artist_name",
"label_owner"])
songs_df = spark.createDataFrame(songs_data, ["song_id", "artist_id", "name"])
global_rank_df = spark.createDataFrame(global_rank_data, ["day", "song_id", "rank"])

# Calculate top 10 appearances
top_artists = global_rank_df.filter(col("rank") <= 10) \
    .join(songs_df, "song_id") \
    .groupBy("artist_id") \
    .agg(count("*").alias("top_10_count")) \
    .join(artists_df, "artist_id") \
    .select("artist_name", "top_10_count") \
    .orderBy(desc("top_10_count"), "artist_name") \
    .limit(5)

# Show results
top_artists.show()

%sql
WITH top_10_appearances AS (
SELECT s.artist_id, COUNT(*) AS top_10_count FROM global_song_rank gsr JOIN songs s
ON gsr.song_id = s.song_id WHERE  gsr.rank <= 10 GROUP BY  s.artist_id)
SELECT a.artist_name,t.top_10_count FROM top_10_appearances t JOIN artists a ON
t.artist_id = a.artist_id ORDER BY t.top_10_count DESC, a.artist_name LIMIT 5;
```


Calculating Delayed Orders Count for Each Delivery Partner

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, unix_timestamp

# Create Spark session
spark = SparkSession.builder.appName("DeliveryAnalysis").getOrCreate()

# Sample data
order_data = [
    (1, 101, 'Bangalore', '2024-01-01', 'PartnerA', '10:00:00', '11:30:00', 60, 100.00),
    (2, 102, 'Chennai', '2024-01-02', 'PartnerB', '12:00:00', '13:15:00', 45, 200.00),
    (3, 103, 'Bangalore', '2024-01-03', 'PartnerA', '14:00:00', '15:45:00', 60, 300.00),
    (4, 104, 'Chennai', '2024-01-04', 'PartnerB', '16:00:00', '17:30:00', 90, 400.00)
]

# Create DataFrame
columns = ["orderid", "custid", "city", "order_date", "del_partner",
           "order_time", "deliver_time", "predicted_time", "aov"]
orders_df = spark.createDataFrame(order_data, columns)

# Calculate actual delivery time in minutes
orders_df = orders_df.withColumn(
    "actual_time_minutes",
    (unix_timestamp("deliver_time") - unix_timestamp("order_time")) / 60)

# Filter delayed orders and count by partner
delayed_orders = orders_df.filter(
    col("actual_time_minutes") > col("predicted_time")).groupBy("del_partner").agg(
    count("*").alias("delayed_orders_count")).orderBy(
    col("delayed_orders_count").desc())

# Show results
delayed_orders.show()

%sql
SELECT del_partner,COUNT(*) AS delayed_orders_count FROM order_details WHERE
```

```
TIMESTAMPDIFF(MINUTE, order_time, deliver_time) > predicted_time GROUP BY
del_partner ORDER BY delayed_orders_count DESC;
```

Identifying Sellers with No Sales in 2020

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year

# Create Spark session
spark = SparkSession.builder.appName("InactiveSellers").getOrCreate()

# Sample data
orders_data = [
   (1, '2020-03-01', 1500.00, 101, 1),
   (2, '2020-05-25', 2400.00, 102, 2),
   (3, '2019-05-25', 800.00, 101, 3),
   (4, '2020-09-13', 1000.00, 103, 2),
   (5, '2019-02-11', 700.00, 101, 2)
]

sellers_data = [
   (1, 'Daniel'),
   (2, 'Ben'),
   (3, 'Frank')
]

# Create DataFrames
orders_df = spark.createDataFrame(orders_data, ["order_id", "sale_date", "order_cost",
"customer_id", "seller_id"])
sellers_df = spark.createDataFrame(sellers_data, ["seller_id", "seller_name"])

# Convert string date to date type
orders_df = orders_df.withColumn("sale_date", col("sale_date").cast("date"))

# Get sellers with 2020 sales
active_2020 = orders_df.filter(year("sale_date") == 2020).select("seller_id").distinct()

# Find inactive sellers (anti-join)
inactive_sellers = sellers_df.join(
   active_2020,
   sellers_df.seller_id == active_2020.seller_id,
   "left_anti")

# Show results
inactive_sellers.show()

%sql
SELECT
   s.seller_id,
   s.seller_name FROM sellers s LEFT JOIN (
   SELECT DISTINCT seller_id
   FROM orders
   WHERE YEAR(sale_date) = 2020) o ON s.seller_id = o.seller_id WHERE o.seller_id IS
NULL;
```

The objective is to write a SQL query that identifies and counts the number of companies that have posted duplicate job listings.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, countDistinct

# Create Spark session
spark = SparkSession.builder.appName("DuplicateJobs").getOrCreate()

# Sample data
job_data = [
   (827, 248, 'Business Analyst', 'Business analyst evaluates past and current business
data...'),
   (845, 149, 'Business Analyst', 'Business analyst evaluates past and current business
data...'),
   (345, 945, 'Data Analyst', 'Data analyst reviews data to identify key insights...'),
   (345, 164, 'Data Analyst', 'Data analyst reviews data to identify key insights...'),
   (244, 172, 'Data Engineer', 'Data engineer works in a variety of settings...')
]

# Create DataFrame
columns = ["company_id", "job_id", "title", "description"]
jobs_df = spark.createDataFrame(job_data, columns)

# Find duplicate job listings
duplicate_companies = jobs_df.groupBy(
   "company_id",
   "title",
   "description").agg(
   count("*").alias("duplicate_count")).filter(
   col("duplicate_count") > 1)

# Count distinct companies with duplicates
result = duplicate_companies.agg(
   countDistinct("company_id").alias("companies_with_duplicates"))

# Show results
result.show()

%sql
WITH duplicate_jobs AS (
   SELECT
      company_id,
      title,
      description,
      COUNT(*) AS duplicate_count FROM job_listings GROUP BY company_id, title,
description HAVING COUNT(*) > 1)
SELECT COUNT(DISTINCT company_id) AS companies_with_duplicates FROM
duplicate_jobs;
------------------------------------------------------------------------------------------------
```

Given the details of the Amazon customer, specifically focusing on the 'product_spend' table, which contains information about customer purchases, the products they bought, and how much they spent.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, dense_rank
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.appName("TopProductsByCategory").getOrCreate()

# Sample data
product_data = [
    ('appliance', 'refrigerator', 165, 26.00),
    ('appliance', 'refrigerator', 123, 3.00),
    ('appliance', 'washing machine', 123, 19.80),
    ('electronics', 'vacuum', 178, 5.00),
    ('electronics', 'wireless headset', 156, 7.00),
    ('electronics', 'vacuum', 145, 15.00),
    ('electronics', 'laptop', 114, 999.99),
    ('fashion', 'dress', 117, 49.99),
    ('groceries', 'milk', 243, 2.99),
    ('groceries', 'bread', 645, 1.99),
    ('home', 'furniture', 276, 599.99)
]

# Create DataFrame
columns = ["category", "product", "user_id", "spend"]
product_spend_df = spark.createDataFrame(product_data, columns)

# Calculate total spend by category and product
category_product_spend = product_spend_df.groupBy(
    "category", "product").agg(
    sum("spend").alias("total_spend"))

# Define window specification
window_spec = Window.partitionBy("category").orderBy(col("total_spend").desc())

# Rank products within each category
ranked_products = category_product_spend.withColumn(
    "product_rank", dense_rank().over(window_spec))

# Filter top 2 products per category
result = ranked_products.filter(
    col("product_rank") <= 2).orderBy("category", col("total_spend").desc())

# Show results
result.show()
```

```
WITH category_product_spend AS (
    SELECT
        category,
        product,
        SUM(spend) AS total_spend FROM ProductSpend GROUP BY category, product),
ranked_products AS (
    SELECT
        category,
        product,
        total_spend,
        DENSE_RANK() OVER (PARTITION BY category ORDER BY total_spend DESC) AS
product_rank FROM category_product_spend)
SELECT category,product,total_spend FROM ranked_products WHERE product_rank
<= 2 ORDER BY category, total_spend DESC;
```

Finding a User's Third Transaction

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, row_number
from pyspark.sql.window import Window

# Create Spark session
spark = SparkSession.builder.appName("ThirdTransaction").getOrCreate()

# Sample data
transaction_data = [
    (111, 100.50, '2022-01-08 12:00:00'),
    (111, 55, '2022-01-10 12:00:00'),
    (121, 36, '2022-01-18 12:00:00'),
    (145, 24.99, '2022-01-26 12:00:00'),
    (111, 89.60, '2022-02-05 12:00:00')
]

# Create DataFrame
columns = ["user_id", "spend", "transaction_date"]
transactions_df = spark.createDataFrame(transaction_data, columns)

# Convert string date to timestamp
transactions_df = transactions_df.withColumn(
    "transaction_date",
    col("transaction_date").cast("timestamp"))

# Define window specification
window_spec = Window.partitionBy("user_id").orderBy("transaction_date")

# Rank transactions for each user
ranked_transactions = transactions_df.withColumn(
    "transaction_rank",
    row_number().over(window_spec))

# Filter for third transaction
third_transactions = ranked_transactions.filter(
    col("transaction_rank") == 3).select(
    "user_id",
    "spend",
    "transaction_date")

# Show results
third_transactions.show()

%sql
WITH ranked_transactions AS (
    SELECT
        user_id,
        spend,
        transaction_date,
        ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY transaction_date) AS
transaction_rank FROM transactions)
SELECT
   user_id,
   spend,
   transaction_date FROM ranked_transactions WHERE transaction_rank = 3;
```

Identifying Customers Who Purchased from Every Product Category

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, countDistinct

# Create Spark session
spark = SparkSession.builder.appName("CustomerAnalysis").getOrCreate()

# Assuming customer_contracts_df and products_df are already created
# Join the tables
customer_categories = customer_contracts_df.join(
   products_df,
   customer_contracts_df.product_id == products_df.product_id)

# Count distinct categories per customer
customer_category_counts = customer_categories.groupBy("customer_id") \
   .agg(countDistinct("product_category").alias("categories_purchased"))

# Get total number of categories
total_categories = products_df.select(countDistinct("product_category")) \
   .first()[0]

# Filter customers who purchased all categories
result = customer_category_counts.filter(
   col("categories_purchased") ==
total_categories).select("customer_id").orderBy("customer_id")

# Show results
result.show()

%sql
# Register DataFrames as temporary views
customer_contracts_df.createOrReplaceTempView("customer_contracts")
products_df.createOrReplaceTempView("products")

# Execute SparkSQL query
result = spark.sql("""
WITH customer_category_counts AS (
SELECT cc.customer_id, COUNT(DISTINCT p.product_category) AS
categories_purchased FROM customer_contracts cc JOIN products p ON
cc.product_id = p.product_id GROUP BY  cc.customer_id),
total_categories AS (
SELECT COUNT(DISTINCT product_category) AS total_categories FROM products)
SELECT c.customer_id FROM customer_category_counts c CROSS JOIN
total_categories t WHERE c.categories_purchased = t.total_categories ORDER BY
c.customer_id""")
```

```
result.show()
```