```
import os
import datetime
from datetime import timedelta
import urllib.request
from matplotlib.patheffects import withStroke
from matplotlib.pyplot import fill_between
from pyparsing import withClass
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import SparkSession, Window
import sys
from collections import namedtuple
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.functions import *
import builtins
import random
import urllib.request

python_path = sys.executable
os.environ['PYSPARK_PYTHON'] = python_path
os.environ['JAVA_HOME'] = r'C:\Users\LENOVO\.jdks\corretto-1.8.0_422'
os.environ["HADOOP_HOME"] = "C:\Program Files\Hadoop"

conf = (SparkConf()
          .setAppName("pyspark")
          .setMaster("local[*]")
          .set("spark.driver.host", "localhost")
          .set("spark.default.parallelism", "1")
       )
sc = SparkContext(conf=conf)
spark = SparkSession.builder.getOrCreate()


# SPARK BOILERPLATE
# """
#
# """
# # input_df1
# +------+-------+-------------+----------+------+---+
# |emp_id|   name|department_id| hire_date|salary|age|
# +------+-------+-------------+----------+------+---+
# |     1|  Alice|          101|2020-01-15| 70000| 29|
# |     2|    Bob|          102|2018-06-23| 60000| 35|
# |     3|Charlie|          101|2019-07-01| 90000| 42|
# |     4|  David|          103|2021-03-12| 50000| 24|
# |     5|    Eve|          102|2020-09-10| 75000| 30|
# +------+-------+-------------+----------+------+---+
# input_df2
# +-------------+---------------+-------------+
# |department_id|department_name|     location|
# +-------------+---------------+-------------+
# |          101|    Engineering|     New York|
# |          102|      Marketing|San Francisco|
# |          103|          Sales|      Chicago|
# +-------------+---------------+-------------+
# result_df
#note: find the average salary  and the max_age by department
# +---------------+----------+-------+
# |department_name|avg_salary|max_age|
# +---------------+----------+-------+
# |    Engineering|   80000.0|     42|
# |      Marketing|   67500.0|     35|
# |          Sales|   50000.0|     24|
# +---------------+----------+-------+
#
# """
```

```
# ## Sample DataFrames
employees = [(1, 'Alice', 101, '2020-01-15', 70000, 29),
           (2, 'Bob', 102, '2018-06-23', 60000, 35),
           (3, 'Charlie', 101, '2019-07-01', 90000, 42),
           (4, 'David', 103, '2021-03-12', 50000, 24),
           (5, 'Eve', 102, '2020-09-10', 75000, 30)]

departments = [(101, 'Engineering', 'New York'),
             (102, 'Marketing', 'San Francisco'),
             (103, 'Sales', 'Chicago')]

# Creating DataFrames
df_employees = spark.createDataFrame(
                      employees,
                      ['emp_id', 'name', 'department_id', 'hire_date', 'salary', 'age']
                )
df_departments = spark.createDataFrame(departments, ['department_id', 'department_name', 'location'])

joined_df = (df_departments
           .join(df_employees, "department_id", "left")
           .groupby("department_name")
           .agg(
             avg("salary").alias("avg_salary"),
             max("age").alias("max_age")
           ).orderBy("department_name"))  #.select("department_name", "avg_salary", "max_age")
joined_df.show()
#note: with spark.sql
df_employees.createOrReplaceTempView("employees")
df_departments.createOrReplaceTempView("departments")
spark.sql("""
    select
       d.department_name, avg(e.salary), max(e.age)
    from departments d
       left join employees e
       on d.department_id = e.department_id
    group by department_name
    order by department_name
""").show()



#
# # info:  scenario 2
# """
# # Convert price from string to double.
# # Group by product_name and aggregate the total sales (quantity * price).
# # Separate aggregation based on customer_type: calculate total sales for
# # Regular and Premium customers.
# # product_name    total_sales_regular    total_sales_premium
# +--------------+------------+--------+------+----------+-------------+
# |transaction_id|product_name|quantity| price| sale_date|customer_type|
# +--------------+------------+--------+------+----------+-------------+
# |             1|      Laptop|       1|1200.5|2023-01-10|      Regular|
# |             2|       Mouse|       3|  25.0|2023-01-11|      Regular|
# |             3|      Laptop|       2|1200.5|2023-01-12|      Premium|
# |             4|    Keyboard|       1|  75.0|2023-01-13|      Regular|
# |             5|       Mouse|       5|  25.0|2023-01-14|      Premium|
# +--------------+------------+--------+------+----------+-------------+
#
# +------------+-------------------+-------------------+
# |product_name|total_sales_regular|total_sales_Premium|
# +------------+-------------------+-------------------+
# |      Laptop|             1200.5|             2401.0|
# |       Mouse|               75.0|              125.0|
# |    Keyboard|               75.0|                0.0|
# +------------+-------------------+-------------------+
```

```
transactions = [(1, 'Laptop', 1, '1200.50', '2023-01-10', 'Regular'),
             (2, 'Mouse', 3, '25.00', '2023-01-11', 'Regular'),
             (3, 'Laptop', 2, '1200.50', '2023-01-12', 'Premium'),
             (4, 'Keyboard', 1, '75.00', '2023-01-13', 'Regular'),
             (5, 'Mouse', 5, '25.00', '2023-01-14', 'Premium')]

# Create DataFrame
df_transactions = (spark
                .createDataFrame(
                   transactions,
                   ['transaction_id', 'product_name', 'quantity', 'price', 'sale_date',
                             'customer_type']))
df_transactions = df_transactions.withColumn('price', col('price').cast('double'))

df_transactions.show()


sales_groupby_df = (df_transactions.groupby("product_name").agg(
    sum(
       when(col("customer_type") == 'Regular', col("quantity") * col("price"))
       .otherwise(0))
    .alias("total_sales_regular"),
    sum(
       when(col("customer_type") == 'Premium', col("quantity") * col("price"))
       .otherwise(0)).alias("total_sales_Premium")
))
sales_groupby_df.show()

print(df_transactions.dtypes)

df_transactions.createOrReplaceTempView("transactions")

df = spark.sql("""
    select
       product_name,
       sum(case
             when customer_type='Regular'
                then quantity * cast(price as double) else 0 end) as Regular_sales,
       sum(case
             when customer_type='Premium'
                then quantity * cast(price as double) else 0 end) as Premium_sales
    from
       transactions
    group by
       product_name
""")
df.show()
print(df.dtypes)

#
# #note: scenario-3
# # Join sales with products on product_id.
# # Handle NULL values by replacing NULL in quantity with 1 and NULL in price with 100.
# # Group by category and aggregate the total sales and count of distinct stores.
# # category,   total_sales,   distinct_stores
# """
# Two input DF's
# +----------+------------+-----------+------------+
# |product_id|product_name|   category|manufacturer|
# +----------+------------+-----------+------------+
# |      1001|       Phone|Electronics|    XYZ Corp|
# |      1002|     Charger|Electronics|     ABC Ltd|
# |      1003|  Headphones|Electronics|    XYZ Corp|
# |      1004| Phone Cover|Accessories|     PQR Inc|
# |      1005|      Tablet|Electronics|    XYZ Corp|
# +----------+------------+-----------+------------+
#
```

```
# +-------+----------+--------+-----+----------+--------+
# |sale_id|product_id|quantity|price| sale_date|store_id|
# +-------+----------+--------+-----+----------+--------+
# |      1|      1001|       2|500.0|2023-08-01|     201|
# |      2|      1002|    NULL| 20.0|2023-08-02|    NULL|
# |      3|      1003|       4| NULL|2023-08-03|     202|
# |      4|      1004|       1|300.0|2023-08-04|     201|
# |      5|      1005|    NULL| NULL|2023-08-05|     203|
# +-------+----------+--------+-----+----------+--------+
#
# the Resultant df
# +-----------+-----------+---------------+
# |   category|total_sales|distinct_stores|
# +-----------+-----------+---------------+
# |Electronics|     1520.0|              3|
# |Accessories|      300.0|              1|
# +-----------+-----------+---------------+
# """
sales = [(1, 1001, 2, 500.00, '2023-08-01', 201),
        (2, 1002, None, 20.00, '2023-08-02', None),
        (3, 1003, 4, None, '2023-08-03', 202),
        (4, 1004, 1, 300.00, '2023-08-04', 201),
        (5, 1005, None, None, '2023-08-05', 203)]

products = [(1001, 'Phone', 'Electronics', 'XYZ Corp'),
          (1002, 'Charger', 'Electronics', 'ABC Ltd'),
          (1003, 'Headphones', 'Electronics', 'XYZ Corp'),
          (1004, 'Phone Cover', 'Accessories', 'PQR Inc'),
          (1005, 'Tablet', 'Electronics', 'XYZ Corp')]

# Create DataFrames
df_sales = spark.createDataFrame(
                   sales,
                   ['sale_id', 'product_id', 'quantity', 'price', 'sale_date', 'store_id']
                )
df_products = spark.createDataFrame(
                products,
                ['product_id', 'product_name', 'category', 'manufacturer']
    )

df_products.show()
df_sales.show()

joined_df = df_sales.join(df_products, "product_id", "left")
null_df = joined_df.fillna({"quantity": 1, "price": 100})
null_df.show()
res1 = (null_df
       .groupby("category")
       .agg(
          sum(col("price") * col("quantity")).alias("total_sales"),
          countDistinct("store_id").alias("distinct_stores")
       )
)
res1.show()

df_products.createOrReplaceTempView("products")
df_sales.createOrReplaceTempView("sales")
```

```
spark.sql("""
    select
       p.category,
       sum(coalesce(s.quantity, 1)* coalesce(s.price,100)) as total_sales,
       count(distinct store_id) as distinct_stores
    from sales s
       left join products p
    on s.product_id = p.product_id
    group by p.category
    order by distinct_stores desc
""").show()
```

```
# # note: scenario 4
# # Apply window functions to calculate the cumulative sum of page_views for each user_id,
#   ordered by visit_time.
#
# # Calculate the difference between the current page_views and the previous value.
# """
# i/p
# +--------+-------+----------+-------------------+
# |visit_id|user_id|page_views|         visit_time|
# +--------+-------+----------+-------------------+
# |       1|    101|         5|2024-01-10 09:00:00|
# |       2|    101|         7|2024-01-10 10:00:00|
# |       3|    102|         3|2024-01-10 09:30:00|
# |       4|    101|         4|2024-01-10 11:00:00|
# |       5|    103|        10|2024-01-10 10:30:00|
# +--------+-------+----------+-------------------+
# o/p
# +--------+-------+----------+-------------------+-------------+-------------------+
# |visit_id|user_id|page_views|         visit_time|cum_sum_pages|diff_from_last_view|
# +--------+-------+----------+-------------------+-------------+-------------------+
# |       1|    101|         5|2024-01-10 09:00:00|            5|               NULL|
# |       2|    101|         7|2024-01-10 10:00:00|           12|                  2|
# |       4|    101|         4|2024-01-10 11:00:00|           16|                 -3|
# |       3|    102|         3|2024-01-10 09:30:00|            3|               NULL|
# |       5|    103|        10|2024-01-10 10:30:00|           10|               NULL|
# +--------+-------+----------+-------------------+-------------+-------------------+
# """
#
# # Sample DataFrame
web_traffic = [(1, 101, 5, '2024-01-10 09:00:00'),
             (2, 101, 7, '2024-01-10 10:00:00'),
             (3, 102, 3, '2024-01-10 09:30:00'),
             (4, 101, 4, '2024-01-10 11:00:00'),
             (5, 103, 10, '2024-01-10 10:30:00')]

# Create DataFrame
df_web_traffic = spark.createDataFrame(
             web_traffic,
             ['visit_id', 'user_id', 'page_views', 'visit_time']
)
df_web_traffic.show()

window_spec = Window.partitionBy("user_id").orderBy("visit_time")
res = (df_web_traffic
       .withColumn("cum_sum_pages",sum("page_views").over(window_spec))
       .withColumn(
    "diff_from_last_view",
    col("page_views") - lag("page_views").over(window_spec)
      )
)
```

```
res.show()

df_web_traffic.createOrReplaceTempView("traffic")
spark.sql("""
    select
       visit_id, user_id, visit_time, page_views,
       sum(page_views) over(partition by user_id order by visit_time) as cum_sum_pages,
       page_views-lag(page_views) over( partition by user_id order by visit_time) as diff_time
    from traffic
""").show()



# # note: scenario 5
# #  You want to assign ranks to sales representatives based on their total sales,
# #  partitioned by department.
# """
# i/p
# +-------+----------+--------+------------+
# |sale_id|department|rep_name|sales_amount|
# +-------+----------+--------+------------+
# |      1|     Sales|   Alice|         200|
# |      2|     Sales|     Bob|         150|
# |      3| Marketing| Charlie|         300|
# |      4| Marketing|   David|         100|
# |      5|     Sales|     Eve|         250|
# +-------+----------+--------+------------+
#
# o/p
# +-------+----------+--------+------------+----+----------+
# |sale_id|department|rep_name|sales_amount|rank|row_number|
# +-------+----------+--------+------------+----+----------+
# |      3| Marketing| Charlie|         300|   1|         1|
# |      4| Marketing|   David|         100|   2|         2|
# |      5|     Sales|     Eve|         250|   1|         1|
# |      1|     Sales|   Alice|         200|   2|         2|
# |      2|     Sales|     Bob|         150|   3|         3|
# +-------+----------+--------+------------+----+----------+
# """
# # Sample DataFrame
sales_data = [(1, 'Sales', 'Alice', 200),
            (2, 'Sales', 'Bob', 150),
            (3, 'Marketing', 'Charlie', 300),
            (4, 'Marketing', 'David', 100),
            (5, 'Sales', 'Eve', 250)]

df_sales = spark.createDataFrame(sales_data, ['sale_id', 'department', 'rep_name', 'sales_amount'])

df_sales.show()
window_spec = Window.partitionBy("department").orderBy(desc("sales_amount"))

res = (df_sales
       .withColumn("rank", rank().over(window_spec))
       .withColumn("row_number", row_number().over(window_spec))
    )
res.show()


df_sales.createOrReplaceTempView("sales")
spark.sql("""
    select
       sale_id, department, rep_name, sales_amount,
       rank()over(partition by department order by sales_amount desc) as rank,
       dense_rank()over(partition by department order by sales_amount desc) as dense_rank,
       row_number()over(partition by department order by sales_amount desc) as row_num
    from sales
""").show()
```

```
# #note: scenario 6
# # Calculate a moving average of daily sales over a 3-day window.
# """
# i/p
# +-------+----------+------------+
# |sale_id|      date|sales_amount|
# +-------+----------+------------+
# |      1|2024-01-01|         100|
# |      2|2024-01-02|         150|
# |      3|2024-01-03|         200|
# |      4|2024-01-04|         250|
# |      5|2024-01-05|         300|
# +-------+----------+------------+
#
# o/p
# +-------+----------+------------+----------+
# |sale_id|      date|sales_amount|moving_avg|
# +-------+----------+------------+----------+
# |      1|2024-01-01|         100|     100.0|
# |      2|2024-01-02|         150|     150.0|
# |      3|2024-01-03|         200|     200.0|
# |      4|2024-01-04|         250|     250.0|
# |      5|2024-01-05|         300|     300.0|
# +-------+----------+------------+----------+
# """
daily_sales = [(1, '2024-01-01', 100),
             (2, '2024-01-02', 150),
             (3, '2024-01-03', 200),
             (4, '2024-01-04', 250),
             (5, '2024-01-05', 300)]

# # Calculate a moving average of daily sales over a 3-day window.
df_daily_sales = spark.createDataFrame(daily_sales, ['sale_id', 'date', 'sales_amount'])

df_daily_sales.show()
window_spec = Window.orderBy("date").rowsBetween(-2,0)
res = df_daily_sales.withColumn("moving_avg", avg('sales_amount').over(window_spec))
res.show()

df_daily_sales.createOrReplaceTempView("sales")
spark.sql("""
    select sale_id, date, sales_amount,
    avg(sales_amount) over(order by date rows between 2 preceding and current row) as moving_avg
    from sales
""").show()
```

```
# #note: scenario 7
# # Calculate the percentile rank of each employee's salary within their department.
# """
# i/p
# +------+----------+--------+------+
# |emp_id|department|emp_name|salary|
# +------+----------+--------+------+
# |     1|        IT|   Alice| 90000|
# |     2|        IT|     Bob| 75000|
# |     3|        HR| Charlie| 70000|
# |     4|        HR|   David| 80000|
# |     5|        IT|     Eve| 95000|
# |     6|        IT|     Eve|  5000|
# +------+----------+--------+------+
#
```

```
# o/p
# +------+----------+--------+------+-----------------+
# |emp_id|department|emp_name|salary|             rank|
# +------+----------+--------+------+-----------------+
# |     4|        HR|   David| 80000|              0.0|
# |     3|        HR| Charlie| 70000|            100.0|
# |     5|        IT|     Eve| 95000|              0.0|
# |     1|        IT|   Alice| 90000|33.33333333333333|
# |     2|        IT|     Bob| 75000|66.66666666666666|
# |     6|        IT|     Eve|  5000|            100.0|
# +------+----------+--------+------+-----------------+
# """
employee_salaries = [(1, 'IT', 'Alice', 90000),
                 (2, 'IT', 'Bob', 75000),
                 (3, 'HR', 'Charlie', 70000),
                 (4, 'HR', 'David', 80000),
                 (5, 'IT', 'Eve', 95000),
                   (6, 'IT', 'Eve', 5000)]

df_salaries = spark.createDataFrame(
          employee_salaries,
          ['emp_id', 'department', 'emp_name', 'salary']
)
df_salaries.show()

window_spec = Window.partitionBy("department").orderBy(desc("salary"))
res = df_salaries.withColumn("rank", percent_rank().over(window_spec)*100)
res.show()

df_with_percentile_rank = df_salaries.withColumn(
'percentile_rank',
((rank().over(window_spec) - 1) / (count('emp_id').over(Window.partitionBy('department')) - 1)) * 100
)
df_with_percentile_rank.show(truncate=8)

df_salaries.createOrReplaceTempView("salaries")
spark.sql("""
    select
       emp_id, department, emp_name, salary,
       percent_rank()over(partition by department order by salary desc) as percentile_rank,

       ((RANK() OVER (PARTITION BY department ORDER BY salary) - 1) * 100.0 /
       (COUNT(emp_id) OVER (PARTITION BY department) - 1)) AS math_percentile_rank
    from salaries
""").show()




# # note: scenario 8
# #  Compute the difference in sales between the current and previous day.
# """
# i/p
# +-------+----------+------------+
# |sale_id|      date|sales_amount|
# +-------+----------+------------+
# |      1|2024-01-01|         100|
# |      2|2024-01-02|         150|
# |      3|2024-01-03|         200|
# |      4|2024-01-04|         250|
# |      5|2024-01-05|         300|
# +-------+----------+------------+
#
```

```
# o/p
# +-------+----------+------------+-------------+-------------+----------+
# |sale_id|      date|sales_amount|prev_day_sales|next_day_sales|sales_diff|
# +-------+----------+------------+-------------+-------------+----------+
# |      1|2024-01-01|         100|         NULL|          150|      NULL|
# |      2|2024-01-02|         150|          100|          200|        50|
# |      3|2024-01-03|         200|          150|          250|        50|
# |      4|2024-01-04|         250|          200|          300|        50|
# |      5|2024-01-05|         300|          250|         NULL|        50|
# +-------+----------+------------+-------------+-------------+----------+
# """
daily_sales = [(1, '2024-01-01', 100),
             (2, '2024-01-02', 150),
             (3, '2024-01-03', 200),
             (4, '2024-01-04', 350),
             (5, '2024-01-05', 300)]

df_daily_sales = spark.createDataFrame(daily_sales, ['sale_id', 'date', 'sales_amount'])
df_daily_sales.show()
window_spec = Window.orderBy("date")

res = df_daily_sales.withColumn(
    "prev_day_sales", lag("sales_amount").over(window_spec)
).withColumn(
    "next_day_sales", lead("sales_amount").over(window_spec)
).withColumn("sales_diff", col("sales_amount") - col("prev_day_sales")
           )
res.show()

df_daily_sales.createOrReplaceTempView("sales")
spark.sql("""
    select
       sale_id, date, sales_amount,
       lag(sales_amount) over(order by date) as prev_day_sales,
       lead(sales_amount) over(order by date) as next_day_sales,
       (sales_amount - prev_day_sales) as sales_diff
    from sales
""").show()





# #note: scenario 9
# """
# i/p
# +-------+--------+----------+------------+
# |sale_id|rep_name|      date|sales_amount|
# +-------+--------+----------+------------+
# |      1|   Alice|2024-01-01|         100|
# |      2|   Alice|2024-01-02|         150|
# |      3|     Bob|2024-01-01|         200|
# |      4|     Bob|2024-01-02|         250|
# +-------+--------+----------+------------+
# o/p
# +-------+--------+----------+------------+-------------+
# |sale_id|rep_name|      date|sales_amount|running_total|
# +-------+--------+----------+------------+-------------+
# |      1|   Alice|2024-01-01|         100|          100|
# |      2|   Alice|2024-01-02|         150|          250|
# |      3|     Bob|2024-01-01|         200|          200|
# |      4|     Bob|2024-01-02|         250|          450|
# +-------+--------+----------+------------+-------------+
# """
```

```
sales_data = [(1, 'Alice', '2024-01-01', 100),
            (2, 'Alice', '2024-01-02', 150),
            (3, 'Bob', '2024-01-01', 200),
            (4, 'Bob', '2024-01-02', 250)]

df_sales = spark.createDataFrame(sales_data, ['sale_id', 'rep_name', 'date', 'sales_amount'])
df_sales.show()
# Define window specification
window_spec = (Window
             .partitionBy('rep_name')
             .orderBy('date')
             .rowsBetween(Window.unboundedPreceding, Window.currentRow)
          )

# Calculate running total
df_with_running_total = (
    df_sales.withColumn('running_total', sum('sales_amount').over(window_spec))
)

df_with_running_total.show()

df_sales.createOrReplaceTempView("sales")
spark.sql("""
    select
       sale_id, rep_name, date, sales_amount,
       sum(sales_amount) over(partition by rep_name order by date) as running_total
    from sales
""")
```

```
# #note: Scenario 10: Calculate the difference in sales between the current day and the next day
#  for each representative.
# """
# i/p
# +-------+--------+----------+------------+
# |sale_id|rep_name|      date|sales_amount|
# +-------+--------+----------+------------+
# |      1|   Alice|2024-01-01|         100|
# |      2|   Alice|2024-01-02|         150|
# |      3|     Bob|2024-01-01|         200|
# |      4|     Bob|2024-01-02|         250|
# +-------+--------+----------+------------+
#
"""
# o/p
+-------+--------+----------+------------+-------------+-------------+---------------+---------------+
|sale_id|rep_name|      date|sales_amount|next_day_sales|prev_day_sales|sales_diff_next|sales_diff_prev|
+-------+--------+----------+------------+-------------+-------------+---------------+---------------+
|      1|   Alice|2024-01-01|         100|          150|         NULL|             50|           NULL|
|      2|   Alice|2024-01-02|         150|         NULL|          100|           NULL|             50|
|      3|     Bob|2024-01-01|         200|          250|         NULL|             50|           NULL|
|      4|     Bob|2024-01-02|         250|         NULL|          200|           NULL|             50|
+-------+--------+----------+------------+-------------+-------------+---------------+---------------+
"""
# """
#
# # note:
```

```
sales_data = [(1, 'Alice', '2024-01-01', 100),

            (2, 'Alice', '2024-01-02', 150),
            (3, 'Bob', '2024-01-01', 200),
            (4, 'Bob', '2024-01-02', 250)]

df_sales = spark.createDataFrame(sales_data, ['sale_id', 'rep_name', 'date', 'sales_amount'])
df_sales.show()

# Define window specification
window_spec = Window.partitionBy('rep_name').orderBy('date')
res = ((df_sales
       .withColumn("next_day_sales", lead("sales_amount").over(window_spec)))
       .withColumn("prev_day_sales", lag("sales_amount").over(window_spec))
       .withColumn("sales_diff_next", col("next_day_sales")-col("sales_amount"))
       .withColumn("sales_diff_prev", col("sales_amount")-col("prev_day_sales"))
    )
res.show()

df_sales.createOrReplaceTempView("sales")
spark.sql("""
    select
       sale_id, rep_name, date, sales_amount,
       lead(sales_amount) over( partition by rep_name order by date) as next_day_sales,
       lag(sales_amount) over(partition by rep_name order by date) as prev_day_sales,
       ( next_day_sales - sales_amount) as sales_diff_next_day,
       ( sales_amount-prev_day_sales) as sales_diff_next_day
    from sales
""").show()






#NOTE: scenario 11: handling data skewness with Broadcast join
# Generate transactions data using Spark directly
"""
+--------------+-----------+------+
|transaction_id|customer_id|amount|
+--------------+-----------+------+
|             0|     cust77| 84.87|
|             1|     cust76| 72.29|
|             2|     cust63| 67.45|
|             3|     cust42| 48.53|
|             4|     cust34| 47.76|
|             5|      cust0|  0.96|
|             6|     cust52|  73.6|
|             7|     cust96| 45.53|
|             8|      cust4|  3.16|
+--------------+-----------+------+
only showing top 20 rows

+-----------+-----+---+
|customer_id| city|age|
+-----------+-----+---+
|      cust0|city0| 20|
|      cust1|city1| 21|
|      cust2|city2| 22|
|      cust3|city3| 23|
|      cust4|city4| 24|
|      cust5|city0| 25|
|      cust9|city4| 29|
+-----------+-----+---+
only showing top 20 rows
```

```
+-----------+--------------+------+-----+---+
|customer_id|transaction_id|amount| city|age|
+-----------+--------------+------+-----+---+
|     cust77|             0| 84.87|city2| 47|
|     cust76|             1| 72.29|city1| 46|
|     cust63|             2| 67.45|city3| 33|
|     cust42|             3| 48.53|city2| 62|
|     cust34|             4| 47.76|city4| 54|
|      cust0|             5|  0.96|city0| 20|
|     cust52|             6|  73.6|city2| 22|
|     cust96|             7| 45.53|city1| 66|
+-----------+--------------+------+-----+---+
only showing top 20 rows

"""
transaction_df = (
    spark.range(1000000)
    .withColumn("customer_id", (100 * rand()).cast("int"))
    .withColumn("amount", round(100 * rand(), 2))
    .selectExpr("id as transaction_id",
             "concat('cust', customer_id) as customer_id", "amount")
)

# Generate customer demographic data with 100 unique customers
customer_data = [(f"cust{j}", f"city{j % 5}", 20 + j % 50) for j in range(100)]
customer_df = spark.createDataFrame(customer_data, ["customer_id", "city", "age"])

transaction_df.show()
customer_df.show()
broadcast_joined_df = (transaction_df
                   .join(broadcast(customer_df), "customer_id", "inner"))
broadcast_joined_df.show()
transaction_df.createOrReplaceTempView("trans")
customer_df.createOrReplaceTempView("cust")
spark.sql("""
    select /*+ BROADCAST(c) */
       t.customer_id, t.transaction_id, t.amount, c.city, c.age
    from trans t
    join cust c
       on t.customer_id = c.customer_id
""").show()



#note scenario 12
#note: complex aggregations with windowing functions
# calculate the moving average of sales for each product over the last 30 days
# and rank the top 5 customers by their total purchase value over the same period.

"""
+-----+----------+--------------+------+-----------+----------+------------------+
|   id|product_id|transaction_id| price|customer_id|      date|        moving_avg|
+-----+----------+--------------+------+-----------+----------+------------------+
|17407|     prod1|    trans17407| 499.1|     cust16|2023-01-01|             499.1|
|20723|     prod1|    trans20723|231.37|      cust3|2023-01-01|           365.235|
|25783|     prod1|    trans25783|458.32|      cust4|2023-01-01| 396.2633333333333|
|47100|     prod1|    trans47100| 25.72|     cust90|2023-01-01|          303.6275|
|57201|     prod1|    trans57201|220.04|     cust86|2023-01-01|286.90999999999997|
|43268|     prod1|    trans43268|106.56|     cust93|2023-01-02|256.85166666666663|
|27820|     prod1|    trans27820|417.18|     cust81|2023-01-03| 279.7557142857143|
+-----+----------+--------------+------+-----------+----------+------------------+
```

```
o/p df required
+-----------+------------------+----+
|customer_id|    total_purchase|rank|
+-----------+------------------+----+
|     cust13|274669.94000000035|   1|
|     cust86| 273267.0899999998|   2|
|     cust65|273192.32000000024|   3|
|     cust51|270988.78000000026|   4|
|     cust72|270844.58999999997|   5|
+-----------+------------------+----+
"""


# Create DataFrame
sales_df = spark.read.format("csv").options(header=True).load("output_file.csv")
sales_df.show()
print(sales_df.dtypes)
sales_df = sales_df.withColumn("price", round(sales_df["price"], 2))
sales_df = sales_df.withColumn("date", to_date("date"))
# sales_df = sales_df.withColumn()
print(sales_df.dtypes)

# Show sample data

window_spec = Window.partitionBy("product_id").orderBy("date").rowsBetween(-30,0)
mov_avg_30_days = sales_df.withColumn("moving_avg", avg("price").over(window_spec))

mov_avg_30_days.show()

window_spec_customer = Window.orderBy(col("total_purchase").desc())

# Group by customer and calculate total purchase in the last 30 days
customer_purchase_df = (sales_df
                   .groupBy("customer_id")
                   .agg(sum("price").alias("total_purchase"))
                   .withColumn("rank", dense_rank().over(window_spec_customer))
                )
#
top_5_customers = customer_purchase_df.filter(col("rank") <= 5)
top_5_customers.show()

sales_df.createOrReplaceTempView("sales")
spark.sql("""
    with ranked_sales as (
        select
           customer_id,
            sum(price) as total_purchases,
            dense_rank() over (order by sum(price) desc) as rank
        from
            sales
        group by
            customer_id
    )
    select *
    from ranked_sales
    where rank <= 5
""").show()
```

```
# note: below is the same query as above but we are specifying the columns which are not part of the
#  aggregation
#  we would get below error
#  [MISSING_AGGREGATION] The non-aggregating expression "id" is based on columns which are not
#  participating in the GROUP BY clause.
    # with ranked_sales as (
    # select
    #  id,product_id, transaction_id, price, customer_id, date,
    #  sum(price) as total_purchases,
    #  dense_rank() over(order by sum(price) desc) as rank
    # from sales
    # group by customer_id
    # )
    # select *
    # from ranked_sales
    # where rank<=5




# NOTE: scenario 13:  sample json structure
"""
[
    {
       "employee_id": 101,
       "name": "John Doe",
       "department": "Engineering",
       "details": {
          "age": 29,
          "address": {
             "street": "123 Elm St",
             "city": "Springfield",
             "state": "IL",
             "postal_code": "62704"
          },
          "skills": [
             {"name": "Python", "level": "Expert", "experience": 5},
             {"name": "Spark", "level": "Intermediate", "experience": 3}
          ]
       },
       "salary": 75000.50,
       "projects": [
          {"name": "Project Alpha", "budget": 100000, "duration_months": 12},
          {"name": "Project Beta", "budget": 50000, "duration_months": 6}
       ]
    },
]
"""
# info: flatten the data
#  calculate the average salary by the department
#  filter the employees with  skill_exp>=5 and skill_level='Expert'
#  filter the projects wth project_budget>60000 and project_duration>6
#  add a column as salary_bane when salary>=100000 then 'High' when salary>=80000  then 'Medium'
#  else 'Low' # find the  avg_project_duration(months) # rank the employees by salary and then by
#  department
#  find the total experience include the skill_experience and sum them and
#  filter the employees with more than 10 years experience
"""
```

root

```
root

|-- department: string (nullable = true)
|-- details: struct (nullable = true)
|    |-- address: struct (nullable = true)
|    |    |-- city: string (nullable = true)
|    |    |-- postal_code: string (nullable = true)
|    |    |-- state: string (nullable = true)
|    |    |-- street: string (nullable = true)
|    |-- age: long (nullable = true)
|    |-- skills: array (nullable = true)
|    |    |-- element: struct (containsNull = true)
|    |    |    |-- experience: long (nullable = true)
|    |    |    |-- level: string (nullable = true)
|    |    |    |-- name: string (nullable = true)
|-- employee_id: long (nullable = true)
|-- name: string (nullable = true)
|-- projects: array (nullable = true)
|    |-- element: struct (containsNull = true)
|    |    |-- budget: long (nullable = true)
|    |    |-- duration_months: long (nullable = true)
|    |    |-- name: string (nullable = true)
|-- salary: double (nullable = true)
"""
df = spark.read.option("multiline", "true").json("gpt2.json")
df.show()
df.printSchema()
trr=(df
     .withColumn("skills", expr("explode(details.skills)"))
     .withColumn("projects", expr("explode(projects)"))
    )
exploded_df = trr.selectExpr(
    "employee_id",
    "name",
    "department",
    "salary",
    "skills.level as skill_level",
    "skills.experience as skill_exp",
    "skills.name as skill_name",
    "projects.budget as project_budget",
    "projects.duration_months as project_duration",
    "projects.name as project_name",
)
exploded_df.show()
exploded_df.printSchema()

avg_salary = exploded_df.groupBy("department").avg("salary")
print("average salary df")
avg_salary.show()

sep_df = exploded_df.select(
    "project_name",
    "project_duration",
    "project_budget",
    "skill_name",
    "skill_level",
    "skill_exp"
)

sep_df.show()


fil_com_df = exploded_df.filter("skill_exp>=5 and skill_level='Expert'")
print("filter exp & salary df")
fil_com_df.show()

fil_com_df_2 = exploded_df.filter("project_budget>60000 and project_duration>6")
print("filter duration>6 & budget > 60k df")
fil_com_df_2.show()
```

```
high_low_df = exploded_df.withColumn(
    "salary_band",
    expr("""
    case
          when salary>=100000 then 'High'
          when salary>=80000  then 'Medium'
          else 'Low'
    end
    """)
)
print("salary band")
high_low_df.show()
total_budget_df = exploded_df.groupBy("department").agg(sum("project_budget").alias("total_budget"))
print("total_budget df")
total_budget_df.show()


avg_project_duration = (exploded_df
                   .groupBy("department")
                   .agg(avg("project_duration").alias("avg_project_duration(months)")))
print("average project duration")
avg_project_duration.show()

window_spec = Window.partitionBy("department").orderBy("salary")
rank_emp_by_dept = (exploded_df
                .withColumn("rank_by_dept", dense_rank().over(window_spec))
             )
print("employee rank by the department")
rank_emp_by_dept.show()


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

salary_rank_df = (exploded_df
               .withColumn("salary_rank", dense_rank().over(window_spec)))
print("over all salary rank")
salary_rank_df.show()


total_years_df = exploded_df.groupBy("name").agg( sum("skill_exp").alias("total_exp") )
print("total+skill experience")
total_years_df.show()

max_exp_df = (exploded_df
            .groupBy("name")
            .agg(sum("skill_exp").alias("total_exp"))
            .orderBy(col("total_exp").desc())
            .filter("total_exp>10")
          )
print("df with exp > 10")
max_exp_df.show()

exploded_df.createOrReplaceTempView("exploded")
spark.sql("""
    select
       department,
       avg(salary) as avg_salary
    from
       exploded
    group by
       department
    order by
       department
""").show()
```

```
# # fil_com_df = exploded_df.filter("skill_exp>=5 and skill_level='Expert'")
spark.sql("""
    select
       employee_id, name, department, salary, skill_level,
       skill_exp, skill_name, project_budget, project_duration,
       project_name
    from
       exploded
    where
       skill_exp>=5
       and
       skill_level='Expert'
""").show()

spark.sql("""
    select
       employee_id, name, department, salary, skill_level,
       skill_exp, skill_name, project_budget, project_duration,
       project_name
    from
       exploded
    where
       project_duration>6
       and
       project_budget>60000
""").show()

spark.sql("""
    select
       employee_id, name, department, salary, skill_level,
       skill_exp, skill_name, project_budget, project_duration,
       project_name,
       case
          when salary >=100000 then 'High'
          when salary>=80000 then 'Medium'
          else 'Low'
       end as salary_band
    from
       exploded
""").show()


spark.sql("""
    select
       department,
       sum(project_budget) total_budget
    from
       exploded
    group by
       department
""").show()

spark.sql("""
    select
       department,
       avg(project_duration) as avg_project_duration_months
    from
       exploded
    group by
       department
""").show()
```

```
spark.sql("""
    select
       employee_id, name, department, salary, skill_level,
       skill_exp, skill_name, project_budget, project_duration,
       project_name,
       dense_rank()over(partition by department order by salary desc) as rank_by_dept
    from
       exploded
""").show()

spark.sql("""
    select
       employee_id, name, department, salary, skill_level,
       skill_exp, skill_name, project_budget, project_duration,
       project_name,
       dense_rank()over(order by salary desc) as rank_by_salary
    from
       exploded
""").show()
# the same query but if you want second or third highest filter the rank for that

spark.sql("""
    select
       name,
       sum(skill_exp) as total_exp
    from
       exploded
    group by
       name
    having
       sum(skill_exp)>10
""").show()


#==============================================================================================
#note: GPT-2 data
#note scenario 14
df = spark.read.option("multiline","true").json("gpt_data.json")
# df.show()
# df.printSchema()
df = df.withColumn("products", expr("explode(products)"))

# cols = df.select("products.shipping.*")
# print(cols.columns)
```

```
exploded_df  = df.selectExpr(
    "comments",
    "order_date",
    "order_id",
    "order_total",
    "products.category",
    "products.discount",
    "products.name as product_name",
    "products.price",
    "products.product_id",
    "products.quantity",
    "products.shipping.company",
    "products.shipping.estimated_delivery",
    "products.shipping.shipping_cost",
    "products.shipping.tracking_number",
    "shipped",
    "user.account_balance",
    "user.email",
    "user.loyalty_points",
    "user.name as user_name",
    "user.payment_method",
    "user.user_id",
    "user.address.city",
    "user.address.street_address",
    "user.address.zip_code",
    "user.address.geo.lat",
    "user.address.geo.lon",
)
exploded_df.printSchema()
print("EXPLODED_DF PRINTING")
exploded_df.show()

# note: 15. Customer Segmentation and Loyalty Analysis
#  Scenario: Group customers based on their total spending (order_total) and loyalty points
#  (loyalty_points). Create custom segments like:
#  High-value, high-loyalty (e.g., customers with order totals > $5000 and loyalty points > 8000).
#  Low-value, low-loyalty (e.g., order totals < $500 and loyalty points < 2000).
#  Objective: Write Spark code to identify and categorize customers into these segments.
#  Additionally, find the top 5 cities with the highest number of high-value customers.

loyalty_value_df = exploded_df.withColumn("Value", expr("""
          case
             when order_total>3500 then 'High'
             when order_total<500 then 'Low'
             else 'Moderate'
          end
""")).withColumn("Loyalty", expr("""
          case
             when loyalty_points>8000 then 'High'
             when loyalty_points<2000 then 'Low'
             else 'Moderate'
          end
"""))
loyalty_value_df.show()
High_value_df = loyalty_value_df.filter("Value='High'")
High_value_df.show()
# to find the top ranking cities first assign high-low-moderate to the columns based on that
# filter out the High value rows
# group them by the city and then dense rank them which will be the expected result
window_spec = Window.orderBy(col("count").desc())
high_value_df =  (
    High_value_df.groupBy("city").agg(count('*').alias("count"))
               .withColumn("rank", dense_rank().over(window_spec))
               .filter("rank<=5")
    )
high_value_df.show()
```

```
exploded_df.createOrReplaceTempView("exploded")
high_value_sql=spark.sql("""
    with ranking as (
       select
          *,
          case
             when order_total>3500 then 'High'
             when order_total<500 then 'Low'
             else 'Moderate'
          end as Value,
          case
             when loyalty_points>8000 then 'High'
             when loyalty_points<2000 then 'Low'
             else 'Moderate'
          end as Loyalty
       from
          exploded
    )
    select *
    from
       ranking
    where
       Value='High'
""")
high_value_sql.show()
high_value_sql.createOrReplaceTempView("high_value")
spark.sql("""
    with ranking as(
       select
          city,
          count(*) as count,
          dense_rank() over(order by count(*) desc) as rank
       from
          high_value
       group by
          city
    )
    select
       *
    from
       ranking
    where
       rank<=5
""").show()
# # note 16: below is the attempted solution for which i had to do the groupBy which i was not
#  able to perform
# #INFO: keep in mind that when ever you want to perform aggregations on even windows you
#  should do the groupby
#  over the columns of any sort
# # or else it would not work grouping is mandatory
#
high_value_df = (
    High_value_df
           .withColumn("count", count("*").over(Window.partitionBy("city")))
           .withColumn("Cities_rank", dense_rank().over(window_spec))
           .filter("Cities_rank <= 5")
    )
high_value_df.show()
```

```
#note:17. Time-Series Analysis for Shipping Efficiency
# Scenario: Calculate the difference between order_date and estimated_delivery for all orders and
# categorize them based on shipping time
# (e.g., "fast" for less than 5 days, "normal" for 5-10 days, and "slow" for more than 10 days).
# Objective: Analyze the shipping performance by shipping company and identify which company
# provides the fastest delivery on average.
# Also, calculate the average shipping cost per company.
#
raw_df = exploded_df.selectExpr(
    "order_date",
    "estimated_delivery",
    "company",
    "shipping_cost",
)
raw_df.show()
raw_df.printSchema()
days_df = raw_df.withColumn("order_date", to_timestamp(col("order_date"))) \
    .withColumn("estimated_delivery", to_timestamp(col("estimated_delivery"))) \
    .withColumn(
    "days_difference",
       datediff(col("estimated_delivery"), col("order_date"))) \
    .withColumn("delivery_category", expr("""
          case
             when days_difference> 200 then 'slow'
             when days_difference>100 then 'normal'
             when days_difference>50 then 'fast'
             when days_difference>5 then 'super_fast'
             else 'within four days'
          end
    """))
days_df.show()
print("the days printed schema")
days_df.printSchema()
num_orders_df = days_df.groupBy("company").agg(count("*").alias("num_orders")).filter("num_orders>20")
num_orders_df.show()
joined_df = days_df.join(num_orders_df, "company", "inner")
print("JOINED DF")
joined_df.show()

window_spec = Window.orderBy("avg_days_to_ship")
company_df = (
    joined_df
       .groupBy("company")
       .agg(avg("days_difference").alias("avg_days_to_ship"))
       .withColumn("rank_by_avg_delivery_time(days)",dense_rank().over(window_spec))
    )
company_df.show()
print("company df showing now below\n\n\n")

avg_shipping_cost_df = (days_df
                   .groupBy("company")
                   .agg(avg("shipping_cost").alias("avg_shipping_cost_company"))
                   .orderBy("avg_shipping_cost_company"))
avg_shipping_cost_df.show()

total_df = company_df.join(avg_shipping_cost_df, "company", 'inner')
total_df.show()
exploded_df.createOrReplaceTempView("raw")
```

```
order = spark.sql("""
    with date_conversion as (
       select
          to_timestamp(replace(order_date, 'T', ' '), 'yyyy-MM-dd HH:mm:ss') AS order_date,
          to_timestamp(estimated_delivery, 'yyyy-MM-dd') AS estimated_delivery,
          company,
          shipping_cost
       from
          raw
    ),
    days_diff as (
       select
          *,
          datediff(estimated_delivery, order_date) as days_to_deliver
       from
          date_conversion
    )
    select
       *,
       case
          when days_to_deliver> 200  then 'slow'
          when days_to_deliver>100   then 'normal'
          when days_to_deliver>50    then 'fast'
          when days_to_deliver>5        then 'super_fast'
          else 'within four days'
       end as order_speed
    from days_diff
""")
order.show()
print("Joined data frame")

order.createOrReplaceTempView("order")

min_orders=spark.sql("""
    with company as (
       select
          company,
          count(*) as num_orders
       from
          order
       group by
          company
    )
    select
       *
    from
       company
    where
       num_orders>=20
""")


min_orders.createOrReplaceTempView("min_orders")

joined= spark.sql("""
    select
       o.*,
       m.num_orders
    from
       order o
    inner join
       min_orders  m
    on
       m.company=o.company
""")
joined.createOrReplaceTempView("joined")
```

```
company = spark.sql("""
    with avg as (
       select
          company,
          avg(days_to_deliver) as avg_days_to_deliver
       from
          joined
       group by
          company
    )
    select
       a.*,
       dense_rank()over(order by avg_days_to_deliver) as rank_by_avg_to_ship_days
    from
       avg a
""")
company.show()




#note: 18. Product Sales and Discount Optimization
#   Scenario: Analyze product sales by category, taking into account the discounts applied (discount),
#   and assess whether higher discounts lead to higher sales.
#   Objective: Write a Spark query to find the correlation between discount and quantity sold.
#   Find the top 3 product categories where higher discounts
#   drive significantly higher sales. Additionally, analyze which product category provides the
#   highest revenue (considering both price and quantity).

raw_df = exploded_df.selectExpr(
    "order_id",
    "category",
    "discount",
    "product_name",
    "price",
    "quantity",
    "order_total"
)
raw_df.show()
raw_df.printSchema()


correlation_df = (raw_df
               .groupBy("category")
               .agg(corr("discount", "quantity").alias("correlation")))

correlation_df.show()

top_3_categories = correlation_df.orderBy(col("correlation").desc()).limit(3)
top_3_categories.show()
revenue_df = (raw_df
            .groupBy("category")
            .agg(sum(col("price") * col("quantity")).alias("total_revenue")))
top_revenue_category = revenue_df.orderBy(col("total_revenue").desc())
top_revenue_category.show()

# Assuming your DataFrame is named 'df'
average_order_total = raw_df.agg(avg("order_total"))
average_order_total.show()
```

```
raw_df.createOrReplaceTempView("raw")
spark.sql("""
    select
       category,
       sum(price * quantity) as total_revenue
    from
       raw
    group by
       category
    order by
       total_revenue desc
""").show()

spark.sql("""
    select
       avg(order_total) as avg_order_price
    from
       raw
""").show()

#note: 19. Fraud Detection with Anomalous Orders
#    Scenario: Identify potentially fraudulent transactions where the account_balance is
#    suspiciously low (e.g., < $50),
#    but the order_total is abnormally high (e.g., > $1000),
#    especially when the payment_method is unconventional (e.g., bitcoin).
#    Objective: Write a Spark query to detect such anomalies, and analyze trends
#    (e.g., are certain product categories more susceptible to these transactions?).

anomaly_df = (exploded_df
            .filter("account_balance<50 and order_total>1000 and payment_method='bitcoin'")
            .select("account_balance", "order_total", "category", "payment_method"))
anomaly_df.show()
anomaly_df = (anomaly_df
            .groupBy("category")
            .agg(count("category").alias("count"))
            .orderBy(col("count").desc()))
print(anomaly_df)
print()
print(f"""The category with got affected by frauds the most is
          {anomaly_df.collect()[0][0]} with  {anomaly_df.collect()[0][1]} of frauds""")
print()
# "account_balance", "order_total", "category", "payment_method"
spark.sql("""
    select
       account_balance, order_total, category, payment_method
    from
       exploded
    where
       account_balance<50 and
       order_total>1000   and
       payment_method='bitcoin'
""").show()
spark.sql("""
    select
       category,
       count(*) as frauds
    from
       exploded
    where
       account_balance<50 and
       order_total>1000   and
       payment_method='bitcoin'
    group by
       category
    order by
       frauds desc
""").show()
```

```
# note: 20. Order Retention and Cancellation Insights
#    Scenario: Analyze whether shipped (shipped = True) orders are more likely to have customer
#    comments or issues, compared to non-shipped orders.
#
shipped_df = (exploded_df
            .filter("shipped=true")
            .select("comments", "shipped")
            .groupBy("comments")
            .agg(count("comments").alias("count"))
            .orderBy(col("count").desc()))
shipped_df.show()
print(shipped_df.count())


spark.sql("""
    select
       comments,
       count(comments) as count
    from
       exploded
    where
       shipped=true
    group by
       comments
    order by
       count desc
""").show()
```

```python
# note: gpt-3 Data
# note: 21
df = spark.read.option("multiline","true").json("Ecommerce_data.json")
# df.printSchema()
df = df.withColumn("items", expr("explode(order.items)"))
df.printSchema()

# x = df.select("order.shipping.*")
# print(x.columns)
#
"""
root
 |-- order: struct (nullable = true)
 |    |-- customer: struct (nullable = true)
 |    |    |-- account_balance: double (nullable = true)
 |    |    |-- email: string (nullable = true)
 |    |    |-- loyalty: struct (nullable = true)
 |    |    |    |-- loyalty_points: long (nullable = true)
 |    |    |    |-- tier: string (nullable = true)
 |    |    |-- name: struct (nullable = true)
 |    |    |    |-- first_name: string (nullable = true)
 |    |    |    |-- last_name: string (nullable = true)
 |    |    |-- user_id: string (nullable = true)
 |    |-- items: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- category: string (nullable = true)
 |    |    |    |-- name: string (nullable = true)
 |    |    |    |-- price: double (nullable = true)
 |    |    |    |-- product_id: string (nullable = true)
 |    |    |    |-- quantity: long (nullable = true)
 |    |-- order_date: string (nullable = true)
 |    |-- order_id: string (nullable = true)
 |    |-- payment: struct (nullable = true)
 |    |    |-- amount: double (nullable = true)
 |    |    |-- discount: double (nullable = true)
 |    |    |-- method: string (nullable = true)
 |    |-- shipping: struct (nullable = true)
 |    |    |-- address: struct (nullable = true)
 |    |    |    |-- city: string (nullable = true)
 |    |    |    |-- coordinates: struct (nullable = true)
 |    |    |    |    |-- lat: double (nullable = true)
 |    |    |    |    |-- lon: double (nullable = true)
 |    |    |    |-- street_address: string (nullable = true)
 |    |    |    |-- zip_code: string (nullable = true)
 |    |    |-- estimated_delivery: string (nullable = true)
 |    |    |-- shipped: string (nullable = true)
 |    |    |-- shipping_cost: double (nullable = true)
 |    |    |-- tracking_number: string (nullable = true)
 |-- items: struct (nullable = true)
 |    |-- category: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- price: double (nullable = true)
 |    |-- product_id: string (nullable = true)
 |    |-- quantity: long (nullable = true)
"""
```

```
exploded_df = df.selectExpr(
    "order.customer.account_balance",
    "order.customer.email",
    "order.customer.loyalty.loyalty_points",
    "order.customer.loyalty.tier",
    "order.customer.name.first_name",
    "order.customer.name.last_name",
    "order.customer.user_id",
    "items.category",
    "items.name",
    "items.price",
    "items.product_id",
    "items.quantity",
    "order.order_date",
    "order.order_id",
    "order.payment.amount",
    "order.payment.discount",
    "order.payment.method",
    "order.shipping.address.city",
    "order.shipping.address.coordinates.lat",
    "order.shipping.address.coordinates.lon",
    "order.shipping.address.street_address",
    "order.shipping.address.zip_code",
    "order.shipping.estimated_delivery",
    "order.shipping.shipped",
    "order.shipping.shipping_cost",
    "order.shipping.tracking_number",
)
# exploded_df.show()
exploded_df.printSchema()
#note: 1.Calculate the Average Shipping Time per User Tier
# You need to calculate the average number of days it takes for an order to be delivered for each
# user based on their tier (standard, premium, etc.).
# Use the order_date and estimated_delivery columns to calculate the number of days for delivery,
# and then aggregate the results by tier.
# Transformation Steps:
# Calculate the difference between estimated_delivery and order_date to get days_for_delivery.
# Group by tier and calculate the average delivery time for each tier.
#   Filter orders that have been marked as shipped = true.
exploded_df = (exploded_df
             .withColumn(
                   "order_date",
                   to_date(exploded_df["order_date"],"yy-MM-dd")
              )
             .withColumn(
                   "estimated_delivery",
             to_date(exploded_df["estimated_delivery"], "yy-MM-dd")
             )
             .withColumn(
                "days_for_delivery",
                date_diff(col('estimated_delivery'), col('order_date'))
             )
)
# exploded_df = exploded_df.filter("days_for_delivery>0")
exploded_df = exploded_df.withColumn("rating", (rand()*5).cast("int"))
exploded_df.printSchema()
# exploded_df.cache()
exploded_df.show()
exploded_df.createOrReplaceTempView("exploded")


raw_df = (exploded_df
         .filter(exploded_df["shipped"]=='true')
         .groupBy("tier")
         .agg(avg("days_for_delivery").alias("avg_days_for_delivery_by_tier"))
       )
raw_df.show()
```

```
spark.sql("""
    select
       tier,
       avg(days_for_delivery) as avg_days_to_ship_by_tier
    from
       exploded
    where
       shipped='true'
    group by
       tier
""").show()


#note: 2. Join Orders with User Information and Perform Aggregations
# Create a new DataFrame by joining the user's profile information with their orders.
# Then, compute the following metrics:#
# Total order_amount for each user.
#   Total loyalty_points gained for each purchase.
#   Average discount received per user across all their orders.
# Transformation Steps:
#
# Perform an inner join between the user information (using user_id) and the order details.
# Create new columns for total order amount (price * quantity - discount).
# Aggregate data at the user level to compute total order_amount, loyalty_points, and average discount.

raw_df = exploded_df.selectExpr(
    "user_id",
    "account_balance",
    "category",
    "loyalty_points",
    "order_id",
    "product_id",
    "quantity",
    "discount",
    "price",
    "shipped",
    "tier"
)
raw_df.printSchema()
cust_df = (raw_df
          .groupBy("user_id")
          .agg(
             avg("discount").alias("avg_discount"),
             sum(col("price")*col("quantity")-col("discount")).alias("total_order_amount")
          )
    )
cust_df.show()
tier_df = raw_df.groupBy("tier").agg(avg("discount").alias("avg_discount_by_tier"))
tier_df.show()
raw_df.createOrReplaceTempView("raw")
spark.sql("""
    select
       user_id,
       avg(discount)as avg_discount,
       sum(price*quantity-discount) as total_order_amount
    from
       raw
    group by
       user_id
""").show()
```

```
spark.sql("""
    select
       tier,
       avg(discount) as avg_discount_by_tier
    from
       raw
    group by
       tier
""").show()


#note: 3. Identify Users Who Are Eligible for Tier Upgrade Based on Spending Patterns
# Assume that users are eligible for a tier upgrade if their total order amount across all
# orders exceeds a
# threshold (e.g., $5,000). Generate a list of users who are eligible for a tier upgrade
# and mark their tier as premium.
#
# Transformation Steps:
#
# Aggregate orders by user_id to calculate total spending.
# Filter users whose total spending exceeds the threshold.
# Update the tier column in the resulting DataFrame to reflect the new tier status for eligible users.

raw_df = exploded_df.selectExpr(
    "user_id",
    "account_balance",
    "category",
    "loyalty_points",
    "order_id",
    "product_id",
    "quantity",
    "discount",
    "price",
    "shipped",
    "tier"
)
cust_df = (raw_df
          .groupBy("user_id")
          .agg(sum(col("price")*col("quantity")-col("discount")).alias("total_order_amount")))
cust_df = cust_df.join(raw_df.select("user_id", "tier").distinct(), on="user_id", how="left")

# Step 3: Update the 'tier' based on the condition
# cust_df = cust_df.withColumn(
#   "tier", when(col("total_order_amount") > 5000, "Premium").otherwise(col("tier"))
# )
cust_df = cust_df.withColumn("tier", expr("""
       case
          when total_order_amount>5000 then 'Premium'
          else tier
       end
"""))
cust_df.show()

# Extract the user_id column and collect the values into a list
Premium_user_id_list = cust_df.filter("tier='Premium'").select("user_id").rdd.flatMap(lambda x: x).collect()

# Show the list of user_id values
print(len(Premium_user_id_list))
raw_df.createOrReplaceTempView("raw")
```

```
#note: 4. Complex Nested JSON Scenario

# Assume you receive new data in JSON format with nested fields, containing user reviews for products.
# The JSON schema is as follows:
# Flatten the nested structure, extracting user_id, order_id, product_id, rating, and comments.
# Aggregate the average rating per product_id and identify the products with the highest average rating.
# Transformation Steps:
# Group by product_id to compute the average rating and rank products based on ratings.

raw_df = exploded_df.selectExpr("product_id", "user_id", "rating")
raw_df = (raw_df
         .groupBy("product_id")
         .agg(avg("rating").alias("avg_rating"))
         .orderBy(col("avg_rating").desc())
       )
raw_df.show()
```

```
# note: 5. Identify Top Performing Cities by Revenue
# Calculate the total revenue generated from orders for each city and then rank the cities
# based on their revenue.
# Only include orders where the product was shipped (shipped = true). Transformation Steps:
# Filter orders where shipped = true.
# Compute total revenue per city (quantity * price - discount).
# Rank cities based on total revenue and return the top 10.

raw_df = exploded_df.filter("shipped='true'")
window_spec = Window.orderBy(col("total_order_amount_by_city").desc())
cities_df = (raw_df
           .withColumn("total_order_amount", expr("price*quantity-discount"))
           .groupBy("city")
           .agg(sum("total_order_amount").alias("total_order_amount_by_city"))
           .withColumn("rank",dense_rank().over(window_spec))
    )
cities_df.show()
```