databricks 24 December

(https://databricks.com)

```
%sql
PySpark the python library for Apache Spark , offers two fundamental data structures that serve as the building blocks of distributed
DataFrames.
 Resilient Distributed Datasets:-

      At the core of pyspark data model lies RDD immutable distributed collections of objects that can be processed in parallel across
      tolerance through lineage information, enabling efficient re-computation of lost data partitions. RDD's are perfect for low leve
      making then ideal for complex data manipulations and custom computations.
Example :-

       rdd = sc.parallelize([1,2,3,4,5])
       -- perform a Square
       squared_rdd = rdd.map(lmbda x:x**2)
       result = squared_rdd.collect()
       print(result)



%sql
dataFrames :-

          Dataframes provide a higher-level abstracion in PySpark, offering a more user-friendly way to work with distributed data.
          PySpark dataframe organize data into named columns, making querying and manipulation a breeze. They leverage Spark's Catalys
          queries and seamless integration with populaer data formats like JSON, Parquet and CSV. Dataframes are well-suited for struc
          learning tasks and data exploration.




from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

data = [{'name':'Nitya', 'Age': 25},
{'name':'Nityaa', 'Age': 35},
{'name':'Nityaaa', 'Age': 34}
]
df = spark.createDataFrame(data)
filtered_df = df.filter(df.Age>30)
filtered_df.show()
```

```
+---+-------+
|Age|   name|
+---+-------+
| 35| Nityaa|
| 34|Nityaaa|
+---+-------+
```

```
Spark Architecture :-
                    At the head of Spark efficiency lies it's powerful architecture, designed to handled complex big data loads seamle

Cluster Manager :-
                    Spark Architecture operates on a master- slave model, where a central manager oversees the distribution of tasks a
                    cluster manager ensures fault tolerance, load balancing and resource allocation, making it the backbone of Spark p
```

Transformation:-

PySpark RDD Transformations are lazy evaluation and is used to transform/update from one RDD into another. When executed on RDD, it re
new RDD. transformations always create a new RDD without updating an existing one hence, a chain of RDD transformations creates an RDD

RDD Transformations are Lazy:-

RDD Transformations are lazy operations meaning none of the transformations get executed until you call an action on PySpark RDD. Sinc
transformations on it result in a new RDD leaving the current one unchanged.

Narrow Transformation:

Narrow transformations are the result of map() and filter() functions and these compute data that live on a single partition meaning t
movement between partitions to execute narrow transformations.

Wider Transformation:

Wider transformations are the result of groupByKey() and reduceByKey() functions and these compute data that live on many partitions m
movements between partitions to execute wider transformations. Since these shuffles the data, they also called shuffle transformations

Use of StructType:-
It acts as a blueprint for creating structured data. It allows us to define a schema by specifying a sequence of StructField objects.
Each StructField represents a column with name, data-type and a optional flag indicating nullability.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
spark= SparkSession.builder.appName("Demo").getOrCreate()
schema = StructType([StructField("id", IntegerType(), False),
                     StructField("name", StringType(), True),
                     StructField("age", IntegerType(), True)])
df = spark.createDataFrame([], schema)
df.show()
```

```
+---+----+---+
| id|name|age|
+---+----+---+
+---+----+---+
```

Use of StructField:

Column specification StructField helps us specify the characteristics of each column. Here's how we can use it to defi

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
spark = SparkSession.builder.appName("demo").getOrCreate()
name_field = StructField("name", StringType(), True)
schema = StructType([name_field])
df = spark.createDataFrame([], schema)
df.show()
```

```
+----+
|name|
+----+
+----+
```

VACUUM:-

Vacuum is more than just tidying up; it reclaims storage space by physically removing files that are no longer needed due to d
delta Lake tables, this process is known as data compaction, helps keep your storage efficient and query performance snappy.

Where to use it:

Version Retention:- Delta lake retains multiple versions of data for auditing and time travel. However, over time unus
can be your friend here by removing older versions that are no longer relevant.

Small File CleanUp :-
As data evolves, small files can accumulate leading to overhead and performance issues.VACUUM consolidates small f
efficiency.

Deleted data Cleanup:-
When data is deleted or updated, the old files are retained for a long time. USE VACUUM to clean up these files.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("VacuumDemo").getOrCreate()
spark.sql("VACUUM <table_name>")
```

Handling row dublication in pyspark

Dropping dublicate:-
The simplest approach is to drop dublicate rows based on a subset of columns. this can be done using the
dropDublicates() method.

```
from pyspark.sql import SparkSession
data = [("Alice", 25), ("Bob", 30), ("Alice", 25), ("Katy", 35)]
columns = ["Name", "Age"]
spark = SparkSession.builder.appName("handling").getOrCreate()
df = spark.createDataFrame(data, columns)
df.show()
# dropping dublicate based on a name and age
dff = df.dropDuplicates(['Name', 'Age'])
dff.show()

# using distinct
df.distinct().show()
```

```
|  Bob| 30|
|Alice| 25|
| Katy| 35|
+-----+---+

+-----+---+
| Name|Age|
+-----+---+
|Alice| 25|
|  Bob| 30|
| Katy| 35|
+-----+---+

+-----+---+
| Name|Age|
+-----+---+
|Alice| 25|
|  Bob| 30|
| Katy| 35|
+-----+---+
```

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
data = [("Alice", 25), ("Bob", 30), ("Alice", 25), ("Katy", 35)]
columns = ["Name", "Age"]
spark = SparkSession.builder.appName("handling").getOrCreate()
df = spark.createDataFrame(data, columns)
window_spec = Window.partitionBy("Name", "Age").orderBy("Name")

# adding a row number column
df_with_row_number = df.withColumn("row_number", row_number().over(window_spec))

# filter out duplicate row
deduplicated_windowfn_df = df_with_row_number.filter(df_with_row_number.row_number == 1).drop("row_number")

deduplicated_windowfn_df.show()
```

```
+-----+---+
| Name|Age|
+-----+---+
|Alice| 25|
|  Bob| 30|
| Katy| 35|
+-----+---+
```

Pyspark UDF

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
data = [("Kolkata", 19),
        ("Mumbai", 25),
        ("Delhi", 30)]
columns = ["City", "Temperatures"]
df = spark.createDataFrame(data, columns)
def fahre_to_Celsisu(fareh_temp):
    celsisu_temp = (fareh_temp - 32)*5/9
    return round(celsisu_temp, 2)

# register_udf
convert_to_celsius_udf = udf(fahre_to_Celsisu, FloatType())

# apply UDF to create a new columns
df_with_celsius = df.withColumn("Temp_C",convert_to_celsius_udf("Temperatures"))
df_with_celsius.show()
```

```
+-------+------------+------+
|   City|Temperatures|Temp_C|
+-------+------------+------+
|Kolkata|          19| -7.22|
| Mumbai|          25| -3.89|
|  Delhi|          30| -1.11|
+-------+------------+------+
```

Unveiling the Power of PySpark Writer API and Its Dynamic Options!

PySpark, the Python library behind Apache Spark's magic, has completely transformed the landscape of big data processing. The Writer A offers an elegant solution for writing data to diverse storage systems, while granting you an array of dynamic options to fine-tune yo

1. Adaptable Data Formats: The Writer API effortlessly handles an array of formats - think Parquet, Avro, JSON, and more. It's like a storage possibilities!
2. Optimized Performance: Engineered for speed, this API lets you optimize performance with features like partitioning, compression, a sluggish data writes and hello to precision!
3. Dynamic Partitioning: Forget the limitations of static partitioning. With the Writer API, you can dynamically partition data based storage efficiency and query performance.
4. Flexible Schema Evolution: Embrace changing data structures with grace. The PySpark Writer API seamlessly adapts to evolving schema remains robust as your information grows.
5. Transactional Confidence: Ensure data integrity with transactional writes. The API ensures that either the entire write operation s maintaining the integrity of your precious data.

- mode: Command the writing behavior - choose 'overwrite', 'append', 'ignore', or 'error', based on your needs.
- compression: Compress data like a pro. Opt for codecs such as 'snappy', 'gzip', or 'none' to optimize space and performance.
- partitionBy: Embrace dynamic data partitioning by columns, streamlining organization and boosting query efficiency.
- bucketBy: Distribute data into buckets for a smooth querying experience in Hive-based systems.
- dateFormat: Define date and timestamp formats for consistent and structured data representation.

```
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder.appName("PySparkWriterExample").getOrCreate()

# Sample data
data = [("Alice", 28), ("Bob", 22), ("Charlie", 24)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Write data using PySpark Writer API with dynamic options
df.write.mode("overwrite") \
 .format("parquet") \
 .option("compression", "snappy") \
 .option("partitionBy", "Age") \
 .save("/path/to/output")
```

when we can use selectExpr in PySpark?

```
''' selectExpr() comes in handy when you need to select particular columns while at the same time you also need to apply some sort of
column(s) '''


from pyspark.sql import SparkSession

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

data = [(1, "Alice", "2021-01-15", 100),
 (2, "Bob", "2021-03-20", 200),
 (3, "Charlie", "2021-02-10", 150)]

columns = ["id", "name", "birthdate", "salary"]

df = spark.createDataFrame(data, columns)
df.show()
```

```
+---+-------+----------+------+
| id|   name| birthdate|salary|
+---+-------+----------+------+
```

```
|  1|  Alice|2021-01-15|   100|
|  2|    Bob|2021-03-20|   200|
|  3|Charlie|2021-02-10|   150|
+---+-------+----------+------+
```

```
# 🔹 Selecting Columns with Alias:
df.selectExpr("name AS full_name", "salary * 1.1 AS updated_salary").show()

# 🔹 Mathematical Transformations:
df.selectExpr("salary", "salary * 1.5 AS increased_salary").show()

# 🔹 String Manipulation:
df.selectExpr("name", "substring(birthdate, 1, 4) AS birth_year", "concat(name, ' - ', birth_year) AS name_year").show()

# 🔹 Conditional Expressions:
df.selectExpr("name", "CASE WHEN salary > 150 THEN 'High Salary' ELSE 'Low Salary' END AS salary_category").show()

# 🔹 Type Casting:
df.selectExpr("name", "cast(salary AS double) AS double_salary").show()
```

```
|  Alice|      2021|  Alice - 2021|
|    Bob|      2021|    Bob - 2021|
|Charlie|      2021|Charlie - 2021|
+-------+----------+--------------+

+-------+---------------+
|   name|salary_category|
+-------+---------------+
|  Alice|     Low Salary|
|    Bob|    High Salary|
|Charlie|     Low Salary|
+-------+---------------+

+-------+-------------+
|   name|double_salary|
+-------+-------------+
|  Alice|        100.0|
|    Bob|        200.0|
|Charlie|        150.0|
+-------+-------------+
```

String Manipulation in PySpark

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import trim

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

# Sample data
data = [(" Apple ",), (" Banana ",), (" Cherry ",)]
df = spark.createDataFrame(data, ["fruits"])
df.show()
```

```
+--------+
|  fruits|
+--------+
|  Apple |
| Banana |
| Cherry |
+--------+
```

```
# 1. Using trim.
from pyspark.sql.functions import trim
# Trim leading and trailing spaces
df = df.withColumn("cleaned_data", trim(df["fruits"]))
df.show()
```

```
+--------+------------+
|  fruits|cleaned_data|
+--------+------------+
|  Apple |       Apple|
| Banana |      Banana|
| Cherry |      Cherry|
+--------+------------+
```

```
# 2. ltrim: Leading Character Whisperer
# ltrim specializes in removing those pesky leading characters (spaces or any you specify) from your strings.

from pyspark.sql.functions import ltrim

# Remove leading spaces
df = df.withColumn("cleaned_data", ltrim(df["fruits"]))
df.show()

from pyspark.sql.functions import rtrim
df = df.withColumn("cleaned_data", rtrim(df["fruits"]))
df.show()
```

```
+--------+------------+
|  fruits|cleaned_data|
+--------+------------+
|  Apple |      Apple |
| Banana |     Banana |
| Cherry |     Cherry |
+--------+------------+

+--------+------------+
|  fruits|cleaned_data|
+--------+------------+
|  Apple |       Apple|
| Banana |      Banana|
| Cherry |      Cherry|
+--------+------------+
```

In a PySpark DataFrame named "orders_df" with columns (OrderID, CustomerID

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

# Create a Spark session
spark = SparkSession.builder.appName("DeduplicateOrders").getOrCreate()
data = [(1, 101, 201, "2023-01-15"),
 (2, 102, 202, "2023-01-16"),
 (3, 101, 201, "2023-01-17"),
 (4, 103, 203, "2023-01-18")]
columns = ["OrderID", "CustomerID", "ProductID", "OrderDate"]
orders_df = spark.createDataFrame(data, columns)

# Deduplicate based on  CustomerID and ProductID
window_spec = Window.partitionBy(orders_df["CustomerID"], orders_df["ProductID"]).orderBy('OrderID')
deduplicated_df = orders_df.withColumn("RowNum", row_number().over(window_spec))
deduplicated_df = deduplicated_df.filter(deduplicated_df["RowNum"] == 1).drop("RowNum")

deduplicated_df.show()
```

```
+-------+----------+---------+----------+
|OrderID|CustomerID|ProductID| OrderDate|
+-------+----------+---------+----------+
|      1|       101|      201|2023-01-15|
|      2|       102|      202|2023-01-16|
|      4|       103|      203|2023-01-18|
+-------+----------+---------+----------+
```

```
''' Imagine you work for a retail company that sells a wide range of products across different categories. You have a massive dataset
with the following columns: "Date," "ProductID," "Category," "QuantitySold," and "Revenue."

Your task is to perform sales analysis to identify trends and patterns within each product category. Specifically, you want to calcula
for each product category to understand how sales are evolving over time '''

from pyspark.sql import Window
from pyspark.sql import functions as func

# Define a window specification partitioned by "Category" and ordered by "Date"
window = Window.partitionBy("Category").orderBy("Date").rowsBetween(Window.currentRow, 2)

# Calculate the rolling sum of revenue for each product category
sales_df = sales_df.withColumn("RollingRevenueSum", func.sum("Revenue").over(window))
```

```
NameError: name 'sales_df' is not defined
```

```
'''
Imagine you're managing sales data for an e-commerce platform. Your dataset contains information about products, the quantity sold, and the quantity returned. However, not all data is perfect, and some quantities are missing, represented as NaN (Not-a-Number). Find net Quantity sold from the data.
'''
from pyspark.sql import SparkSession
from pyspark.sql.functions import nanvl
from pyspark.sql.functions import lit
data = [(1,10.0, 2.0), (2,8.0, float('nan')), (3,12.0, 3.0), (4,float('nan'), 5.0)]
df = spark.createDataFrame(data, ["product_id","quantity_sold", "quantity_returned"])

# Calculate net quantity sold, handling NaN values
net_sales_df = df.withColumn(
 "quantity_sold_withoutNull", nanvl(df["quantity_sold"], lit(0.0))
).withColumn(
 "quantity_returned_withoutNull", nanvl(df["quantity_returned"], lit(0.0))
)
result_df = net_sales_df.withColumn(
 "net_quantity_sold", net_sales_df["quantity_sold_withoutNull"] - net_sales_df["quantity_returned_withoutNull"]
)

# Show the result
result_df.drop('quantity_sold','quantity_returned').show()
```

```
+----------+------------------------+----------------------------+-----------------+
|product_id|quantity_sold_withoutNull|quantity_returned_withoutNull|net_quantity_sold|
+----------+------------------------+----------------------------+-----------------+
|         1|                    10.0|                         2.0|              8.0|
|         2|                     8.0|                         0.0|              8.0|
|         3|                    12.0|                         3.0|              9.0|
|         4|                     0.0|                         5.0|             -5.0|
+----------+------------------------+----------------------------+-----------------+
```

```
%sql
/* '''
Leveraging Managed and External Tables for Real-World Data Management
''' */

CREATE TABLE managed_product_data (
 product_id INT,
 product_name STRING,
 price DECIMAL,
 stock_quantity INT
);
```

OK

```
How and when to use broadcast function in pyspark?

The broadcast function in PySpark should be used when you want to optimize join operations between DataFrames, particularly when one DataFrame is significantly smaller than the other. Broadcasting the smaller DataFrame can greatly improve query performance by reducing data shuffling and network overhead.

Suppose we have two dataframe sales_data (with millions of records) and customer_info (small with few thousand records)

In this scenario, the customer_info DataFrame is relatively small compared to the sales_data DataFrame. Broadcasting the smaller DataFrame (customer_info) is beneficial when:
🔹 Joining Large and Small DataFrames: You are joining a large DataFrame (e.g., sales_data) with a significantly smaller DataFrame (customer_info).
🔹 Reducing Data Shuffling: Broadcasting helps reduce the amount of data that needs to be shuffled across worker nodes during the join operation, improving performance.
```

```
Explain the use of df.explain() and its parameters?

Spark provides a powerful tool called df.explain() that gives you a backstage pass to the inner workings of your DataFrame operations.
Optimization Insights: Understand how Spark optimizes your queries to boost performance.
Bottleneck Detection: Spot potential bottlenecks and fine-tune your code for speed.
Shuffle and Partitioning: Get a grip on data shuffling and partitioning strategies.
Efficiency Boost: Ensure your code runs efficiently, especially with large-scale datasets.

from pyspark.sql.functions import broadcast

there are four types of plans: Logical Plan, Analyzed Logical Plan, Optimized Logical Plan and Physical Plan
 🔹 Logical Plan: Represents the abstract representation of a query without optimization.
 🔹 Analyzed Logical Plan: Represents the query plan after parsing and semantic analysis but before optimization.
 🔹 Optimized Logical Plan: Incorporates query optimizations to improve query efficiency.
 🔹 Physical Plan: Specifies how the query will be executed physically, including details about data shuffling, joins, and partitioning strategies.
```

Mastering Sorting in PySpark: nulls in focus

```
from pyspark.sql import SparkSession
data = [
 (1, "Alice", "2023-01-15"),
 (2, "Bob", "2022-12-10"),
 (3, "Charlie", None),
 (4, "David", "2023-02-20"),
 (5, "Eve", None),
]
columns = ["customer_id", "customer_name", "subscription_start_date"]
df = spark.createDataFrame(data, columns)
df.orderBy(df.subscription_start_date.asc_nulls_first()).show()
df.orderBy(df.subscription_start_date.asc_nulls_last()).show()
```

```
+-----------+-------------+----------------------+
|customer_id|customer_name|subscription_start_date|
+-----------+-------------+----------------------+
|          3|      Charlie|                  null|
|          5|          Eve|                  null|
|          2|          Bob|            2022-12-10|
|          1|        Alice|            2023-01-15|
|          4|        David|            2023-02-20|
+-----------+-------------+----------------------+

+-----------+-------------+----------------------+
|customer_id|customer_name|subscription_start_date|
+-----------+-------------+----------------------+
|          2|          Bob|            2022-12-10|
|          1|        Alice|            2023-01-15|
|          4|        David|            2023-02-20|
|          3|      Charlie|                  null|
|          5|          Eve|                  null|
+-----------+-------------+----------------------+
```

pyspark.sql.Window function, the rowsBetween method.

```
'''
Window is used to specify the range of rows considered in a windowed operation. It determines the set of rows relative to the
current row that should be included in the window frame. The frame is used for performing calculations like aggregations,
ranking, and other window functions.

The rowsBetween method accepts two arguments: start and end, which define the boundaries of the frame. These boundaries are
relative to the current row and are specified using specific constants.

Here are the main constants you can use with rowsBetween:
 🔹 Window.unboundedPreceding: Represents the earliest possible row. It means all rows from the beginning of the partition up to
and including the current row.
 🔹 Window.unboundedFollowing: Represents the latest possible row. It means all rows from the current row up to the end of the
partition.
 🔹 Window.currentRow: Represents the current row.
 🔹 Any integer values
'''
```

```
from pyspark.sql.window import Window
from pyspark.sql.functions import sum
data = [("Aman", 10),
 ("Bahadur", 20),
 ("Anjali", 30),
 ("Babita", 40),
 ("Aditya", 50)]

columns = ["category", "value"]
df = spark.createDataFrame(data, columns)

# Define a window specification
window_spec = Window.partitionBy("category").orderBy("value")

# Calculate a cumulative sum considering all rows from the start of the partition up to the current row
df.withColumn("cumulative_sum", sum("value").over(window_spec.rowsBetween(Window.unboundedPreceding, Window.currentRow))).show()
df.withColumn("cumulative_sum", sum("value").over(window_spec.rowsBetween(Window.currentRow, Window.unboundedFollowing))).show()
df.withColumn("cumulative_sum", sum("value").over(window_spec.rowsBetween(Window.unboundedPreceding,
Window.unboundedFollowing))).show()
df.withColumn("cumulative_sum", sum("value").over(window_spec.rowsBetween(Window.currentRow,1))).show()
```

```
+--------+-----+--------------+
|category|value|cumulative_sum|
+--------+-----+--------------+
|  Aditya|   50|            50|
|    Aman|   10|            10|
|  Anjali|   30|            30|
|  Babita|   40|            40|
| Bahadur|   20|            20|
+--------+-----+--------------+

+--------+-----+--------------+
|category|value|cumulative_sum|
+--------+-----+--------------+
|  Aditya|   50|            50|
|    Aman|   10|            10|
|  Anjali|   30|            30|
|  Babita|   40|            40|
| Bahadur|   20|            20|
+--------+-----+--------------+
```

how and when to use datediff() in pyspark?

```
'''
The datediff function in PySpark is used to calculate the difference in days between two dates. It is a valuable tool in various real-life scenarios where you need to perform date-based calculations and analysis. Here are some common use cases for datediff in PySpark:

🔹 Employee Tenure Analysis: You can use datediff to calculate the tenure of employees in an organization. By subtracting the hire date from the current date, you can determine how long each employee has been with the company.
🔹 Customer Churn Analysis: When analyzing customer behavior, datediff can help calculate the time elapsed between a customer's first and last purchase. This information is essential for identifying and predicting customer churn.
🔹 Loan and Mortgage Calculations: In the financial sector, you can use datediff to calculate the duration of loans or mortgages. This helps in determining interest accrued over time and remaining payment periods.
🔹 Event Scheduling: When scheduling events or appointments, datediff can be used to calculate the time remaining until an event or the time passed since an event occurred.
🔹 Inventory Aging: For managing inventory, you can calculate the age of each item in stock using datediff. This helps in identifying and managing aging or obsolete inventory.
🔹 Healthcare Analytics: In healthcare, datediff can be used to calculate the length of hospital stays, the time between medical procedures, or the duration of treatment plans.
'''
```

```
from pyspark.sql.functions import datediff, current_date, lit
from pyspark.sql.types import DateType,StructField,  StructType, StringType

data = [
 ('2023-04-08',),
 ('2023-04-09',),
 ('2023-04-10',),
 ('2023-04-11',),
 ('2023-04-12',),
 ('2023-04-13',)
]
columns = ['d1']

schema = StructType([
 StructField("d1", StringType(), True)
])

df = spark.createDataFrame(data, schema=schema)

# Convert 'd1' to DateType
df = df.withColumn("d1", df["d1"].cast(DateType()))

# Create a new DataFrame with the current date in 'd2'
df_with_current_date = df.withColumn("d2", lit(current_date()))
df_with_current_date.show()
# Calculate the difference in days
df_with_current_date. select(datediff(df_with_current_date.d2, df_with_current_date.d1).alias('diff')).show()
# Calculate the difference in days
df_with_current_date. select(datediff(df_with_current_date.d2, df_with_current_date.d1).alias('diff')).show()
```

```
|diff|
```

```
| 280|
| 279|
| 278|
| 277|
| 276|
| 275|
+----+
```

1️⃣ Ingest Data in Real-Time: As users browse your site, their actions are immediately ingested into Delta Live Tables, creating a real-time data stream.
2️⃣ Transform Data on the Fly: Using Databricks' user-friendly interface, you can apply transformations to this data stream in real-time. For instance, you can enrich user profiles with up-to-the-second information.
3️⃣ Make Instant Decisions: With this enriched data, you can power real-time dashboards that show which products are trending, personalize product recommendations instantly, and even detect unusual behavior indicative of fraud, all in the blink of an eye.
4️⃣ Ensure Data Reliability: Delta Live Tables ensures that your data is reliable and transactional, maintaining data integrity even as you process it in real-time.

```
%sql
CREATE table orders(

 order_id INT,
 order_date STRING,
 customer_id INT,
 order_status STRING
)
using DELTA
TBLPROPERTIES (delta.enableChangeDataFeed = True)
```

OK

```
from pyspark.sql import SparkSession
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22), ("David", 28), ("Eve", 35)]
df = spark.createDataFrame(data, ["Name", "Age"])

# Transformation 1: Group by Age
grouped_data = df.groupBy("Age").count()

# Transformation 2: Filter Ages above 25
filtered_data = grouped_data.filter(grouped_data.Age > 25)

# Action 1: Show the grouped data (Lazy Evaluation)
print("Grouped Data (Lazy):")
grouped_data.show()

# Action 2: Show the filtered data (Lazy Evaluation)
print("Filtered Data (Lazy):")
filtered_data.show()
```

```
Grouped Data (Lazy):
+---+-----+
|Age|count|
+---+-----+
| 25|    1|
| 30|    1|
| 22|    1|
| 28|    1|
| 35|    1|
+---+-----+

Filtered Data (Lazy):
+---+-----+
|Age|count|
+---+-----+
| 30|    1|
```

```
| 28|    1|
| 35|    1|
+---+-----+
```

n PySpark, transformations are categorized into two types: narrow transformations and wide transformations. These categories are based on how they impact the execution plan and data shuffling in a Spark job.

Narrow Transformations:
Narrow transformations are those transformations where each output partition depends on a single input partition.
They do not require data shuffling or data movement across partitions, making them more efficient.
Examples of narrow transformations include map, filter, and union.

```
# Sample DataFrame
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22), ("David", 28), ("Eve", 35)]
df = spark.createDataFrame(data, ["Name", "Age"])

# Narrow Transformation: Filtering ages above 25
filtered_df = df.filter(df.Age > 25)
```

Wide Transformations:
Wide transformations are those transformations where each output partition depends on multiple input partitions.
They require data shuffling or redistribution across partitions, which can be resource-intensive and time-consuming.
Examples of wide transformations include groupByKey and join.

```
# Sample DataFrames
df1 = spark.createDataFrame([(1, "Alice"), (2, "Bob"), (3, "Charlie")], ["ID", "Name"])
df2 = spark.createDataFrame([(1, "Math"), (2, "Science"), (3, "History")], ["ID", "Subject"])

# Wide Transformation: Joining two DataFrames
joined_df = df1.join(df2, "ID")
```

How to handle out of memory error in databricks?

Here are some steps you can take to handle and mitigate out-of-memory errors in Databricks:
🔹 Increase Cluster Memory:
You can try scaling up your cluster by adding more worker nodes or increasing the instance types of the existing nodes. This can provide more memory to your Spark jobs.
🔹 Optimize Your Code:
Review your Spark code and optimize it to use memory efficiently. Make use of Spark transformations and actions that minimize data shuffling and memory usage, such as filter, map, and reduce.
🔹 Partition Your Data:
Ensure that your data is properly partitioned. Well-distributed and properly-sized partitions can significantly reduce memory pressure during processing.
🔹 Use Caching and Persisting:
Cache or persist intermediate DataFrames or RDDs that you need to reuse. This can help avoid recomputation and reduce memory pressure.
🔹 Increase Spark Driver Memory:
If you're running into driver memory issues, consider increasing the driver memory configuration for your Spark job.
🔹 Monitor and Tune Memory Settings:
Use Databricks' built-in monitoring tools to track the memory usage of your Spark jobs. Adjust Spark memory configurations like spark.driver.memory and spark.executor.memory based on your cluster's available resources and job requirements.
🔹 Data Sampling and Filtering:
If your dataset is too large to fit in memory, consider sampling or filtering it to work with smaller subsets. This may be necessary for exploratory data analysis.
🔹 Use Off-Heap Memory:
Spark allows you to use off-heap memory for certain data structures, which can help avoid Java heap space issues. You can configure this using the spark.memory.offHeap.enabled configuration.
🔹 Consider Cluster Autoscaling:
Enable cluster autoscaling in Databricks so that your cluster can automatically add or remove nodes based on workload, ensuring you have the necessary resources when needed.
🔹 Use External Storage:
Consider using external storage solutions like Delta Lake or Data Lakes to store and manage large datasets efficiently without consuming too much memory.
🔹 Regularly Clean Up Unused Data and Resources:
Periodically clean up temporary tables, cached DataFrames, and other resources that are no longer needed to free up memory.

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

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

data = [("Product A: $19.99!",),
 ("Special Offer on Product B - $29.95",),
 ("Product C (Limited Stock)",)]

df = spark.createDataFrame(data, ["description"])

# Clean and preprocess the descriptions using regex_replace
cleaned_df = df.withColumn("cleaned_description",
 regexp_replace(col("description"), r'[^a-zA-Z0-9\s]', ''))
cleaned_df.show(truncate=False)
```

```
+----------------------------------+------------------------------+
|description                       |cleaned_description           |
+----------------------------------+------------------------------+
|Product A: $19.99!                |Product A 1999                |
|Special Offer on Product B - $29.95|Special Offer on Product B  2995|
|Product C (Limited Stock)         |Product C Limited Stock       |
+----------------------------------+------------------------------+
```

```
# . Using na.fill() Method:
# Replace null values in a specific column with a constant value. Here's an example
from pyspark.sql import SparkSession
data = [(1, None), (2, None), (3, "value")]
columns = ["ID", "column_name"]
df = spark.createDataFrame(data, columns)
filledDF = df.na.fill("replacement_value", subset=["column_name"])
filledDF.show()
```

```
+---+-----------------+
| ID|      column_name|
+---+-----------------+
|  1|replacement_value|
|  2|replacement_value|
|  3|            value|
+---+-----------------+
```

```
from pyspark.sql import SparkSession

data = [("Product A", "2023-01"), ("Product B", "2023-02"), ("Product C", "2023-03")]
columns = ["product", "sale_date"]

# Create a DataFrame
df = spark.createDataFrame(data, columns)

# Extract the year and month from the sale_date
result_df = df.withColumn("year_month", df["sale_date"].substr(1, 7))
result_df.show()
```

```
+---------+---------+----------+
|  product|sale_date|year_month|
+---------+---------+----------+
|Product A|  2023-01|   2023-01|
|Product B|  2023-02|   2023-02|
|Product C|  2023-03|   2023-03|
+---------+---------+----------+
```

```
Dealing with Data Skewness in PySpark 📊

Data skewness can be a silent performance killer in our PySpark data processing jobs. When a few partitions or keys in our data set have significantly more data than others, it can lead to imbalanced workloads, slower processing times, and sometimes even out-of-memory errors.

🔍 Identifying Data Skewness
Before solving a problem, we must first recognize it. In PySpark, you can identify data skewness by monitoring the distribution of data across partitions using df.groupBy().count(). If some partitions have substantially more records than others, you likely have a skewness issue.

🔧 Solving Data Skewness in PySpark
Here are a few strategies to address data skewness in PySpark:
🔹 Salting Your Data: Add a random value (salt) to your data using functions like rand() to distribute the data more evenly across partitions. Then, repartition the DataFrame.
🔹 Bucketing: Use bucketing to pre-organize your data into a fixed number of buckets based on a specific column. This can help evenly distribute data and improve join performance.
🔹 Custom Partitioning: Implement custom partitioning logic based on your domain knowledge to evenly distribute the data.
🔹 Use Appropriate Joins: Choose the appropriate join type, like broadcast joins or bucketed joins, depending on your data and query requirements.
🔹 Sampling: In some cases, you might consider using random sampling to reduce the data size, making it more manageable and balanced.
🔹 Caching: Caching heavily accessed DataFrames or tables can reduce the overhead of repeatedly computing the same data, improving query performance.
🔹 Data skewness is a common challenge in distributed data processing, but with these strategies and careful monitoring, we can keep our PySpark jobs running smoothly. 🚀
```

GroupBy

```
'''
GroupBy is a fundamental operation in PySpark that allows you to group rows of a DataFrame based on one or more columns and perform aggregate functions on each group. This operation is essential for summarizing, analyzing, and transforming data
'''

from pyspark.sql.functions import avg, sum, count
data = [("Movie_A", "Drama", 4.5),
 ("Movie_B", "Comedy", 3.8),
 ("Movie_C", "Drama", 4.2),
 ("Movie_D", "Action", 4.0),
 ("Movie_E", "Comedy", 3.5)]

schema = ["movie", "genre", "rating"]
df = spark.createDataFrame(data, schema=schema)

df.groupBy("genre").agg(avg("rating").alias("avg_rating"),count("genre").alias("Movie in each genre")).show()
```

```
+------+----------+------------------+
| genre|avg_rating|Movie in each genre|
+------+----------+------------------+
| Drama|      4.35|                 2|
|Comedy|      3.65|                 2|
|Action|       4.0|                 1|
+------+----------+------------------+
```

```
from pyspark.sql.functions import col, to_timestamp , year, month,datediff,current_date, date_add,  date_trunc,date_format

data = [("EventA", "2023-11-15 08:30:00"),
 ("EventB", "2023-12-20 15:45:30"),
 ("EventC", "2023-12-10 12:00:00")]

schema = ["event_name", "timestamp_str"]

df = spark.createDataFrame(data, schema)
df_timestamps = df.withColumn("event_time", to_timestamp(col("timestamp_str"), "yyyy-MM-dd HH:mm:ss"))
df_timestamps = df_timestamps.withColumn("year", year(col("event_time")))
df_timestamps = df_timestamps.withColumn("month", month(col("event_time")))
df_timestamps = df_timestamps.withColumn("days_diff", datediff(current_date(), col("event_time")))
df_timestamps = df_timestamps.withColumn("next_week", date_add(col("event_time"), 7))
df_timestamps = df_timestamps.withColumn("truncated_hour", date_trunc("hour", col("event_time")))
df_timestamps = df_timestamps.withColumn("formatted_date", date_format(col("event_time"), "dd/MM/yyyy HH:mm:ss"))
df_timestamps.show()
```

```
+----------+-------------------+-------------------+----+-----+---------+----------+-------------------+-------------------+
|event_name|      timestamp_str|         event_time|year|month|days_diff| next_week|     truncated_hour|     formatted_date|
+----------+-------------------+-------------------+----+-----+---------+----------+-------------------+-------------------+
|    EventA|2023-11-15 08:30:00|2023-11-15 08:30:00|2023|   11|       59|2023-11-22|2023-11-15 08:00:00|15/11/2023 08:30:00|
|    EventB|2023-12-20 15:45:30|2023-12-20 15:45:30|2023|   12|       24|2023-12-27|2023-12-20 15:00:00|20/12/2023 15:45:30|
|    EventC|2023-12-10 12:00:00|2023-12-10 12:00:00|2023|   12|       34|2023-12-17|2023-12-10 12:00:00|10/12/2023 12:00:00|
+----------+-------------------+-------------------+----+-----+---------+----------+-------------------+-------------------+
```

```
filter or where Transformation:
Purpose: Filters rows based on specified conditions.
•Syntax: df.filter(condition)
•Example: filtered_df = df.filter(df["age"] > 25)
•Use Cases:
 •Filtering rows based on specific column values.
 •Complex conditions involving multiple columns.

•dropDuplicates Transformation:
 Purpose: Removes duplicate rows based on specified columns.
•Syntax: df.dropDuplicates(subset=columns)
•Example:
•unique_department_df = df.dropDuplicates(subset=["department"])
•Use Cases:
 •Ensuring unique values in specific columns.
 •Preprocessing data before aggregation.
```

```
from pyspark.sql import SparkSession
data = [("Alice", 28, 60000, "HR"),
 ("Bob", 35, 75000, "Engineering"),
 ("Charlie", 22, 50000, "Marketing"),
 ("Alice", 28, 60000, "HR"),  # Duplicate row
 ("David", 40, 90000, "Engineering")]

# Define the schema
schema = ["name", "age", "salary", "department"]

df = spark.createDataFrame(data, schema)

#Filtering Employees with Age Over 30:
senior_employees_df = df.where(df["age"] > 30)
senior_employees_df.show
()

#Dropping Duplicate Rows Based on All Columns:

unique_records_df = df.dropDuplicates()
unique_records_df.show()
```

```
+-------+---+------+-----------+
|   name|age|salary| department|
+-------+---+------+-----------+
|  Alice| 28| 60000|         HR|
|    Bob| 35| 75000|Engineering|
|Charlie| 22| 50000|  Marketing|
|  David| 40| 90000|Engineering|
+-------+---+------+-----------+
```

```
'''
You have a DataFrame containing information about products, including their names and prices. You are tasked with creating a new
column, "PriceCategory," based on the following conditions:
If the price is less than 50, categorize it as "Low."
If the price is between 50 (inclusive) and 100 (exclusive), categorize it as "Medium."
If the price is 100 or greater, categorize it as "High."
'''
```

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

# Sample data
data = [("ProductA", 30),
 ("ProductB", 75),
 ("ProductC", 110)]

# Define the schema
schema = ["ProductName", "Price"]

# Create the DataFrame
productsData = spark.createDataFrame(data, schema)

# Use when and otherwise to categorize product prices
result_df = productsData.withColumn("PriceCategory",
 when(col("Price") < 50, "Low")
 .when((col("Price") >= 50) & (col("Price") < 100), "Medium")
 .otherwise("High")
)

result_df.show()
```

```
+-----------+-----+-------------+
|ProductName|Price|PriceCategory|
+-----------+-----+-------------+
|   ProductA|   30|          Low|
|   ProductB|   75|       Medium|
|   ProductC|  110|         High|
+-----------+-----+-------------+
```

•ArrayType is a data type in PySpark that represents an array or a list of elements.
•It's commonly used when dealing with structured data where a column needs to contain multiple values.

You've been provided with a dataset containing information about stock transactions for an investment portfolio.
Question:
You've been provided with a dataset containing information about stock transactions for an investment portfolio.
1) Calculate the total transaction amount for each transaction. Create a new column named total_transaction in the DataFrame.
2) Compute the cumulative transaction amount for each stock symbol. Create a new column named cumulative_transaction for each stock symbol, representing the sum of total transaction amounts for all transactions of that stock.
3)Identify the most traded stock symbol for each month. Create a new column named top_stock_monthly that contains the stock symbol with the highest total quantity traded in each month.
4)Determine the average unit price for each stock symbol.
5)Identify the stocks with the highest lifetime transaction value (LTV).

Unlocking Data Transformation: Using explode Function in PySpark

```
from pyspark.sql import SparkSession

# Instantiate a Spark
spark = SparkSession.builder.appName("PySparkExplodeFunctionUsage").getOrCreate()
from pyspark.sql.functions import explode

# Sample DataFrame
data = [("Alice", ["apple", "banana", "cherry"]),
        ("Bob", ["orange", "peach"]),
        ("Cathy", ["grape", "kiwi", "pineapple"])]

df = spark.createDataFrame(data, ["Name", "Fruits"])

# Using explode function
exploded_df = df.select("Name", explode("Fruits").alias("Fruit"))
exploded_df.show()
```

```
+-----+---------+
| Name|    Fruit|
+-----+---------+
|Alice|    apple|
|Alice|   banana|
|Alice|   cherry|
|  Bob|   orange|
|  Bob|    peach|
|Cathy|    grape|
|Cathy|     kiwi|
|Cathy|pineapple|
+-----+---------+
```