Final-Mock-Questions

- 1. What is the difference between Map and Flatmap, when will you use it, applications
- 2. How many number of outputs does filter function give?
- 3. What is a lazy evaluation
- 4. In the azure project we wrote multiple files
  - How to Append in Data Flow
  - How to Overwrite in Data Flow
- 5. Why do we need a CTE? Whats the benefit
  - Why Do We Need a CTE?
- 6. Tell some action commands in pyspark
  - RDD Action Commands
  - DataFrame Action Commands
  - Example
- 7. Tell me difference between action and transformation in pyspark. Whats the benefit, as a user what are we getting
  - Transformation
  - Action
  - Example
- 8. Tell me transformations, with examples
  - 1. select(): Selecting Columns
  - 2. filter() or where(): Filtering Rows
  - 3. withColumn(): Adding or Modifying Columns
  - 4. withColumnRenamed(): Renaming a Column
  - 5. drop(): Dropping Columns
  - 6. orderBy() or sort(): Sorting Data
  - 7. groupBy() and agg(): Grouping and Aggregation
  - 8. union(): Combining DataFrames
  - 9. join(): Joining DataFrames
  - 10. dropDuplicates(): Removing Duplicate Rows
  - 11. fillna() or na.fill(): Handling Missing Values

        - 12. dropna() or na.drop(): Dropping Rows with Missing Values
        - 13.Map and Flatmap
        - 1\. map() Transformation
        - 2\. flatMap() Transformation
            - When to use map vs. flatMap
- 9. Broadcast join in pyspark, why would you use? Whats the use? What kind of optimization does it give?
    - What is Broadcast Join?
    - Why would you use Broadcast Join? What's the Use?
    - What kind of optimization does it give?
- 10. What is the syntax for a CTE at a high level, how to use the CTE, when will we use CTE, Why will we use it? How is it different from subquery
    - Syntax for a CTE (High Level)
    - How to Use a CTE?
    - When Will We Use CTEs?
    - How is it Different from a Subquery?
- 11. What is .format()
    - Syntax and Examples
- 12. What are the 2 arguments to map function
- 13. What are the parameters of a filter function?
- 14. When will you use a map, when will you use a flatmap
- 15. What does take command do in pyspark
    - What take() does:
    - Syntax:
    - Examples:
    - When to use take():
- 16. What is the difference between show and take?
- 17. How do you list all files in a folder in hadoop
- 18.What is a delta lake?
- 19. What are the 3 different ways of parquet file encoding
    - 1. Delta encoding
    - 2. Dictionary Encoding (RLEDICTIONARY)
    - 3. Run-Length Encoding (RLE) /

- 20. What is partitioning and bucketing
  - Partitioning
  - Bucketing
- 21. Inline dataset in dataflow?
- 22. What does collect do?
- 23. What is DAG in pyspark? How does it help
  - What is DAG?
  - Example
  - How does DAG help?
- 24. External and internal table or managed table in Hive?
  - 1. Internal Table (Managed Table)
  - 2.External Table
- 25. What If I drop an external table or managed table? What will happen?
  - 1. DROP MANAGED (Internal) Table
  - 2. DROP EXTERNAL Table
- 26. What is HDFS?
  - Features
- 27. In pyspark why do we create a DAG?
- 28. Difference between action and transformation
  - Transformation
  - Action
- 29. What is a overalapping window? When do we use it? Whats the usecase
  - Example
- 30. What are the 3 different tabs, where do you create a linked service
  - 1. Author (or Author & Monitor)
  - 2. Monitor
  - 3. Manage
  - Where to Create a Linked Service?
- 31. What is the output a flatmap would give?
- 32. What is cardinality of data
- 33. Cardinality in terms of bucketing and partitioning. What is the cardininality we choose for bucketing and partitioning?

1. What is the difference between Map and Flatmap, when will you use it, applications

Map

Transforms each element of the RDD using a function.

Returns one output for each input.

The result is an RDD of the same size.

```
rdd = sc.parallelize([1, 2, 3])
mapped = rdd.map(lambda x: [x, x*2])
print(mapped.collect())
```

```
[[1, 2], [2, 4], [3, 6]]
```

Flatmap

Also transforms each element using a function.

But it flattens the result — meaning it merges all the lists into a single RDD.

Useful when each input can produce multiple outputs.

```
rdd = sc.parallelize([1, 2, 3])
flat_mapped = rdd.flatMap(lambda x: [x, x*2])
print(flat_mapped.collect())
```

```
[1, 2, 2, 4, 3, 6]
```

Instead of a list of lists, you get a flat list.

2. How many number of outputs does filter function give?

- One or 0

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

Input: [1, 2, 3, 4, 5]
Output: [2, 4]

3. What is a lazy evaluation

Lazy evaluation means that PySpark doesn't run your code immediately when you write transformations like map, filter, or flatMap.

Instead, it waits until an action is called (like collect(), count(), saveAsTextFile(), etc.) — only then does it actually process the data.

Example

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd2 = rdd.filter(lambda x: x % 2 == 0)
rdd3 = rdd2.map(lambda x: x * 2)
```

Nothing happens yet, PySpark just builds a plan.

```
result = rdd3.collect()
```

Now PySpark executes everything — filter, then map — and gives you the result.

4. In the azure project we wrote multiple files

- How do you append in dataflow
- How do you overwrite, what option you need to give

How to Append in Data Flow

- To append data to existing files:
- Go to the Sink transformation in your data flow.
- Under Settings, look for the option:
  - "Write Behavior" or "Sink Write Behavior"
  - Choose:
    - "Append" — this will add new data to existing files without deleting them.

How to Overwrite in Data Flow

- To overwrite existing files:
- In the Sink settings of your data flow:
  - Set "Write Behavior" to:
    - "Overwrite"
    - This will delete existing files in the target folder and write new ones.

5. Why do we need a CTE? Whats the benefit

A CTE is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement.

Example

```
WITH SalesCTE AS (
    SELECT customer_id, SUM(amount) AS total_sales
    FROM sales
    GROUP BY customer_id
)
SELECT * FROM SalesCTE WHERE total_sales > 1000;
```

Why Do We Need a CTE?

1. Improves Readability
   - Breaks complex queries into smaller, logical parts.
   - Makes SQL easier to understand and maintain.

2. Avoids Repetition
    - You can reuse the same logic multiple times without repeating code.
3. Supports Recursion
    - CTEs can be recursive, which is useful for hierarchical data (like org charts or folder structures).

6. Tell some action commands in pyspark

PySpark uses **lazy evaluation**, so transformations like `map`, `filter`, etc. are not executed until an **action** is called.

RDD Action Commands

Action	Description
`collect()`	Returns all elements to the driver. ⚠️ Use only for small datasets.
`count()`	Returns the number of elements.
`first()`	Returns the first element.
`take(n)`	Returns the first `n` elements.
`reduce(func)`	Aggregates elements using a function.
`saveAsTextFile(path)`	Saves the RDD as a text file.
`foreach(func)`	Applies a function to each element (no return).

DataFrame Action Commands

Action	Description
`show(n)`	Displays the first `n` rows.
`collect()`	Returns all rows as a list of Row objects.
`count()`	Returns the number of rows.
`first()` / `head()`	Returns the first row.
`take(n)`	Returns the first `n` rows.
`toPandas()`	Converts to a Pandas DataFrame. ⚠️ Use only for small data.
`write.format(...).save(...)`	Saves the DataFrame to storage.

Action	Description
`foreach(func)`	Applies a function to each row (used for side effects).

Example

```
df = spark.read.csv("data.csv", header=True)
df_filtered = df.filter(df["age"] > 30)

# Action command
df_filtered.show()
```

7. Tell me difference between action and transformation in pyspark. Whats the benefit, as a user what are we getting

In PySpark, operations are divided into two main types: **Transformations** and **Actions**.

Transformation

- **Definition**: A transformation creates a new RDD or DataFrame from an existing one.
- **Lazy Evaluation**: Transformations are **not executed immediately**. They build a logical plan.
- **Examples**:
  - `map()`
  - `filter()`
  - `select()`
  - `groupBy()`
  - `join()`
- Efficient: PySpark can **optimize** the entire chain of transformations before execution.
- Modular: You can build complex logic step-by-step without triggering computation.

Action

- **Definition**: An action **triggers the execution** of the transformations and returns a result.

- **Execution**: This is when Spark actually **runs the job** and processes the data.
- **Examples**:
  - `collect()`
  - `count()`
  - `show()`
  - `take(n)`
  - `write.format(...).save(...)`
- Retrieves results: You get actual output or save data.
- Final step: Converts the logical plan into a physical execution.

Feature	Transformation	Action
Execution Timing	Lazy (not immediate)	Immediate
Performance	Optimized before run	Triggers computation
Use Case	Build logic	Get results/save data
User Benefit	Flexibility, efficiency	Output, insights

Example

```
# Transformation (lazy)
df_filtered = df.filter(df["age"] > 30)

# Action (triggers execution)
df_filtered.show()
```

8. Tell me transformations, with examples

1. Creating a DataFrame (for example)

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType

spark =
SparkSession.builder.appName("DataFrameTransformations").getOrCreate()
```

```
data = [
    ("Alice", 1, "Sales", 50000),
    ("Bob", 2, "Marketing", 60000),
    ("Charlie", 1, "Sales", 55000),
    ("David", 3, "Engineering", 70000),
    ("Eve", 2, "Marketing", 62000),
    ("Frank", 3, "Engineering", 75000),
    ("Grace", 1, "Sales", 58000)
]

schema = StructType([
    StructField("Name", StringType(), True),
    StructField("DepartmentID", IntegerType(), True),
    StructField("Department", StringType(), True),
    StructField("Salary", IntegerType(), True)
])

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

Output:

```
+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|    Bob|           2|  Marketing| 60000|
|Charlie|           1|      Sales| 55000|
|  David|           3|Engineering| 70000|
|    Eve|           2|  Marketing| 62000|
|  Frank|           3|Engineering| 75000|
|  Grace|           1|      Sales| 58000|
+-------+------------+-----------+------+
```

Common DataFrame Transformations:

1. `select()` : Selecting Columns

Used to select a subset of columns from a DataFrame. You can also rename columns during selection.

```
# Select specific columns
df_selected = df.select("Name", "Department", "Salary")
df_selected.show()

# Select and rename a column
df_renamed_salary = df.select(df.Name, df.Salary.alias("EmployeeSalary"))
df_renamed_salary.show()
```

Output:

```
+-------+----------+------+
|   Name| Department|Salary|
+-------+----------+------+
|  Alice|      Sales| 50000|
|    Bob|  Marketing| 60000|
|Charlie|      Sales| 55000|
|  David|Engineering| 70000|
|    Eve|  Marketing| 62000|
|  Frank|Engineering| 75000|
|  Grace|      Sales| 58000|
+-------+----------+------+

+-------+--------------+
|   Name|EmployeeSalary|
+-------+--------------+
|  Alice|         50000|
|    Bob|         60000|
|Charlie|         55000|
|  David|         70000|
|    Eve|         62000|
|  Frank|         75000|
|  Grace|         58000|
+-------+--------------+
```

2. `filter()` or `where()` : Filtering Rows

Used to filter rows based on a given condition. `filter()` and `where()` are aliases for each other.

```
# Filter employees with Salary > 60000
df_high_earners = df.filter(df.Salary > 60000)
df_high_earners.show()

# Filter employees in 'Sales' department
df_sales_employees = df.where(df.Department == "Sales")
df_sales_employees.show()
```

Output:

```
+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  David|           3|Engineering| 70000|
|    Eve|           2|  Marketing| 62000|
|  Frank|           3|Engineering| 75000|
+-------+------------+-----------+------+

+-------+------------+---------+------+
|   Name|DepartmentID|Department|Salary|
+-------+------------+---------+------+
|  Alice|           1|    Sales| 50000|
|Charlie|           1|    Sales| 55000|
|  Grace|           1|    Sales| 58000|
+-------+------------+---------+------+
```

3. `withColumn()` : Adding or Modifying Columns

Used to add a new column or modify an existing one.

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

# Add a new column 'Bonus' (10% of Salary)
df_with_bonus = df.withColumn("Bonus", col("Salary") * 0.10)
df_with_bonus.show()
```

```
# Modify an existing column (e.g., convert Salary to USD, assuming a fixed
rate)
df_salary_usd = df.withColumn("Salary", col("Salary") / 80) # Assuming 1 USD
= 80 INR
df_salary_usd.show()
```

Output:

```
+-------+------------+-----------+------+-------+
|   Name|DepartmentID| Department|Salary|  Bonus|
+-------+------------+-----------+------+-------+
|  Alice|           1|      Sales| 50000| 5000.0|
|    Bob|           2|  Marketing| 60000| 6000.0|
|Charlie|           1|      Sales| 55000| 5500.0|
|  David|           3|Engineering| 70000| 7000.0|
|    Eve|           2|  Marketing| 62000| 6200.0|
|  Frank|           3|Engineering| 75000| 7500.0|
|  Grace|           1|      Sales| 58000| 5800.0|
+-------+------------+-----------+------+-------+


+-------+------------+-----------+-------+
|   Name|DepartmentID| Department| Salary|
+-------+------------+-----------+-------+
|  Alice|           1|      Sales|  625.0|
|    Bob|           2|  Marketing|  750.0|
|Charlie|           1|      Sales|  687.5|
|  David|           3|Engineering|  875.0|
|    Eve|           2|  Marketing|  775.0|
|  Frank|           3|Engineering|  937.5|
|  Grace|           1|      Sales|  725.0|
+-------+------------+-----------+-------+
```

4. `withColumnRenamed()` : Renaming a Column

Used to rename a specific column.

```
df_renamed = df.withColumnRenamed("Department", "DepartmentName")
df_renamed.show()
```

Output:

```
+-------+------------+--------------+------+
|   Name|DepartmentID|DepartmentName|Salary|
+-------+------------+--------------+------+
|  Alice|           1|         Sales| 50000|
|    Bob|           2|     Marketing| 60000|
|Charlie|           1|         Sales| 55000|
|  David|           3|   Engineering| 70000|
|    Eve|           2|     Marketing| 62000|
|  Frank|           3|   Engineering| 75000|
|  Grace|           1|         Sales| 58000|
+-------+------------+--------------+------+
```

5. `drop()` : Dropping Columns

Used to drop one or more columns from a DataFrame.

```
# Drop a single column
df_no_dept_id = df.drop("DepartmentID")
df_no_dept_id.show()

# Drop multiple columns
df_name_salary_only = df.drop("DepartmentID", "Department")
df_name_salary_only.show()
```

Output:

```
+-------+----------+------+
|   Name|Department|Salary|
+-------+----------+------+
|  Alice|     Sales| 50000|
|    Bob| Marketing| 60000|
|Charlie|     Sales| 55000|
```

```
|  David|Engineering| 70000|
|    Eve|  Marketing| 62000|
|  Frank|Engineering| 75000|
|  Grace|      Sales| 58000|
+-------+-----------+------+

+-------+------+
|   Name|Salary|
+-------+------+
|  Alice| 50000|
|    Bob| 60000|
|Charlie| 55000|
|  David| 70000|
|    Eve| 62000|
|  Frank| 75000|
|  Grace| 58000|
+-------+------+
```

6. `orderBy()` or `sort()`: Sorting Data

Used to sort the DataFrame by one or more columns in ascending or descending order.

```
from pyspark.sql.functions import desc

# Sort by Salary in ascending order
df_sorted_salary_asc = df.orderBy("Salary")
df_sorted_salary_asc.show()

# Sort by Department (ascending) then by Salary (descending)
df_sorted_multi = df.sort("Department", desc("Salary"))
df_sorted_multi.show()
```

Output:

```
+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
```

```
|Charlie|           1|      Sales| 55000|
|  Grace|           1|      Sales| 58000|
|    Bob|           2|  Marketing| 60000|
|    Eve|           2|  Marketing| 62000|
|  David|           3|Engineering| 70000|
|  Frank|           3|Engineering| 75000|
+-------+------------+-----------+------+


+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Frank|           3|Engineering| 75000|
|  David|           3|Engineering| 70000|
|    Eve|           2|  Marketing| 62000|
|    Bob|           2|  Marketing| 60000|
|  Grace|           1|      Sales| 58000|
|Charlie|           1|      Sales| 55000|
|  Alice|           1|      Sales| 50000|
+-------+------------+-----------+------+
```

7. `groupBy()` and `agg()`: Grouping and Aggregation

Used to group rows based on one or more columns and then apply aggregation functions (like sum, count, avg, min, max).

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

# Calculate average salary per department
df_avg_salary_per_dept =
df.groupBy("Department").agg(avg("Salary").alias("AverageSalary"))
df_avg_salary_per_dept.show()

# Count employees and sum salaries per department
df_dept_summary = df.groupBy("Department").agg(
    count("*").alias("NumEmployees"),
    sum("Salary").alias("TotalSalary")
)
df_dept_summary.show()
```

Output:

```
+-----------+-----------------------+
| Department|     AverageSalary     |
+-----------+-----------------------+
|      Sales|      54333.333333333336|
|  Marketing|      61000.0          |
|Engineering|      72500.0          |
+-----------+-----------------------+


+-----------+------------+-----------+
| Department|NumEmployees|TotalSalary|
+-----------+------------+-----------+
|      Sales|           3|     163000|
|  Marketing|           2|     122000|
|Engineering|           2|     145000|
+-----------+------------+-----------+
```

8. `union()`: Combining DataFrames

Used to combine two or more DataFrames with the same schema.

```
data2 = [
    ("Peter", 4, "HR", 45000),
    ("Quinn", 4, "HR", 48000)
]
df2 = spark.createDataFrame(data2, schema)

df_combined = df.union(df2)
df_combined.show()
```

Output:

```
+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|    Bob|           2|  Marketing| 60000|
```

```
|Charlie|           1|      Sales| 55000|
|  David|           3|Engineering| 70000|
|    Eve|           2|  Marketing| 62000|
|  Frank|           3|Engineering| 75000|
|  Grace|           1|      Sales| 58000|
|  Peter|           4|         HR| 45000|
|  Quinn|           4|         HR| 48000|
+-------+------------+-----------+------+
```

9. `join()` : Joining DataFrames

Used to combine DataFrames based on a common column(s). Various join types are available (inner, outer, left, right, semi, anti).

Let's create another DataFrame for departments:

```
dept_data = [
    (1, "Sales", "East"),
    (2, "Marketing", "West"),
    (3, "Engineering", "North"),
    (4, "HR", "Central")
]

dept_schema = StructType([
    StructField("DeptID", IntegerType(), True),
    StructField("DeptName", StringType(), True),
    StructField("Location", StringType(), True)
])

df_departments = spark.createDataFrame(dept_data, dept_schema)
df_departments.show()

# Inner Join
df_joined_inner = df.join(df_departments, df.DepartmentID ==
df_departments.DeptID, "inner")
df_joined_inner.show()

# Left Outer Join
df_joined_left = df.join(df_departments, df.DepartmentID ==
```

```
df_departments.DeptID, "left_outer")
df_joined_left.show()
```

Output:

```
+------+--------+--------+
|DeptID|DeptName|Location|
+------+--------+--------+
|     1|   Sales|    East|
|     2|Marketing|    West|
|     3|Engineering|   North|
|     4|      HR| Central|
+------+--------+--------+

+-------+------------+-----------+------+------+--------+--------+
|   Name|DepartmentID| Department|Salary|DeptID|DeptName|Location|
+-------+------------+-----------+------+------+--------+--------+
|  Alice|           1|      Sales| 50000|     1|   Sales|    East|
|    Bob|           2|  Marketing| 60000|     2|Marketing|    West|
|Charlie|           1|      Sales| 55000|     1|   Sales|    East|
|  David|           3|Engineering| 70000|     3|Engineering|   North|
|    Eve|           2|  Marketing| 62000|     2|Marketing|    West|
|  Frank|           3|Engineering| 75000|     3|Engineering|   North|
|  Grace|           1|      Sales| 58000|     1|   Sales|    East|
+-------+------------+-----------+------+------+--------+--------+

+-------+------------+-----------+------+------+--------+--------+
|   Name|DepartmentID| Department|Salary|DeptID|DeptName|Location|
+-------+------------+-----------+------+------+--------+--------+
|  Alice|           1|      Sales| 50000|     1|   Sales|    East|
|    Bob|           2|  Marketing| 60000|     2|Marketing|    West|
|Charlie|           1|      Sales| 55000|     1|   Sales|    East|
|  David|           3|Engineering| 70000|     3|Engineering|   North|
|    Eve|           2|  Marketing| 62000|     2|Marketing|    West|
|  Frank|           3|Engineering| 75000|     3|Engineering|   North|
```

```
|  Grace|           1|      Sales| 58000|     1|   Sales|    East|
+-------+------------+-----------+------+------+--------+--------+
```

10. `dropDuplicates()`: Removing Duplicate Rows

Used to remove duplicate rows from a DataFrame. You can specify columns to consider for duplication.

```
# Create a DataFrame with duplicates
data_dup = [("A", 1), ("B", 2), ("A", 1), ("C", 3)]
schema_dup = StructType([StructField("Col1", StringType()),
StructField("Col2", IntegerType())])
df_dup = spark.createDataFrame(data_dup, schema_dup)
df_dup.show()

# Drop all duplicate rows
df_distinct = df_dup.dropDuplicates()
df_distinct.show()

# Drop duplicates based on a subset of columns (e.g., 'Col1' only)
df_distinct_col1 = df_dup.dropDuplicates(["Col1"])
df_distinct_col1.show()
```

Output:

```
+----+----+
|Col1|Col2|
+----+----+
|   A|   1|
|   B|   2|
|   A|   1|
|   C|   3|
+----+----+

+----+----+
|Col1|Col2|
+----+----+
|   B|   2|
|   A|   1|
```

```
|   C|   3|
+----+----+


+----+----+
|Col1|Col2|
+----+----+
|   B|   2|
|   A|   1|
|   C|   3|
+----+----+
```

11. `fillna()` or `na.fill()`: Handling Missing Values

Used to fill null values in a DataFrame.

```
from pyspark.sql.functions import mean

data_with_nulls = [
    ("Alice", 1, "Sales", 50000),
    ("Bob", 2, None, 60000),
    ("Charlie", 1, "Sales", None),
    ("David", 3, "Engineering", 70000),
]

schema_nulls = StructType([
    StructField("Name", StringType(), True),
    StructField("DepartmentID", IntegerType(), True),
    StructField("Department", StringType(), True),
    StructField("Salary", IntegerType(), True)
])

df_nulls = spark.createDataFrame(data_with_nulls, schema_nulls)
df_nulls.show()

# Fill nulls in 'Department' with "Unknown"
df_fill_dept = df_nulls.fillna("Unknown", subset=["Department"])
df_fill_dept.show()

# Fill nulls in 'Salary' with the mean salary
mean_salary = df_nulls.select(mean("Salary")).collect()[0][0]
```

```
df_fill_salary = df_nulls.fillna(mean_salary, subset=["Salary"])
df_fill_salary.show()
```

Output:

```
+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|    Bob|           2|       null| 60000|
|Charlie|           1|      Sales|  null|
|  David|           3|Engineering| 70000|
+-------+------------+-----------+------+


+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|    Bob|           2|    Unknown| 60000|
|Charlie|           1|      Sales|  null|
|  David|           3|Engineering| 70000|
+-------+------------+-----------+------+


+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|    Bob|           2|       null| 60000|
|Charlie|           1|      Sales| 60000|
|  David|           3|Engineering| 70000|
+-------+------------+-----------+------+
```

12. `dropna()` or `na.drop()` : Dropping Rows with Missing Values

Used to drop rows containing null values.

```
# Drop rows with any null value
df_drop_any_null = df_nulls.dropna()
df_drop_any_null.show()

# Drop rows where 'Department' column has a null value
df_drop_dept_null = df_nulls.dropna(subset=["Department"])
df_drop_dept_null.show()
```

Output:

```
+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|  David|           3|Engineering| 70000|
+-------+------------+-----------+------+

+-------+------------+-----------+------+
|   Name|DepartmentID| Department|Salary|
+-------+------------+-----------+------+
|  Alice|           1|      Sales| 50000|
|Charlie|           1|      Sales|  null|
|  David|           3|Engineering| 70000|
+-------+------------+-----------+------+
```

13.Map and Flatmap

```
from pyspark.sql import SparkSession

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

# Create an RDD from a Python list
data = ["hello world", "spark is great", "data processing"]
rdd = spark.sparkContext.parallelize(data)
```

```
print("Original RDD elements:")
print(rdd.collect())
```

Output:

```
Original RDD elements:
['hello world', 'spark is great', 'data processing']
```

1. `map()` Transformation

The `map()` transformation applies a function to each element of the RDD and returns a new RDD with the results. There's a one-to-one correspondence between input and output elements.

Key characteristics of `map()`:

- **One-to-one transformation:** For every input element, there is exactly one output element.
- **Maintains the number of records:** The resulting RDD will have the same number of elements as the original RDD.
- **Used for:**
  - Changing the type of elements (e.g., string to integer).
  - Performing calculations on each element.
  - Extracting specific parts of each element.

Syntax: `rdd.map(func)`

Example 1: Converting strings to their lengths

```
# Apply a lambda function to get the length of each string
rdd_lengths = rdd.map(lambda x: len(x))

print("\nRDD after map (lengths):")
print(rdd_lengths.collect())
```

Output:

```
RDD after map (lengths):
[11, 14, 15]
```

- **Explanation:** `hello world` (11 characters), `spark is great` (14 characters), `data processing` (15 characters). Each input string produced one integer output.

Example 2: Converting strings to uppercase

```
# Convert each string to uppercase
rdd_upper = rdd.map(lambda x: x.upper())

print("\nRDD after map (uppercase):")
print(rdd_upper.collect())
```

Output:

```
RDD after map (uppercase):
['HELLO WORLD', 'SPARK IS GREAT', 'DATA PROCESSING']
```

- **Explanation:** Each input string was transformed into its uppercase version, maintaining the one-to-one relationship.

2. `flatMap()` Transformation

The `flatMap()` transformation applies a function to each element of the RDD, and then flattens the result.

Key characteristics of `flatMap()`:

- **One-to-many or one-to-zero transformation:** An input element can produce zero, one, or multiple output elements.
- **Changes the number of records:** The resulting RDD can have more or fewer elements than the original RDD.
- **Used for:**
  - Splitting a single line of text into individual words.
  - Exploding a list of items within each record into separate records.
  - Filtering elements by returning an empty iterable for unwanted ones.

Syntax: `rdd.flatMap(func)`

Example 1: Splitting sentences into words

```
# Split each sentence into words
rdd_words = rdd.flatMap(lambda x: x.split(" "))

print("\nRDD after flatMap (words):")
print(rdd_words.collect())
```

Output:

```
RDD after flatMap (words):
['hello', 'world', 'spark', 'is', 'great', 'data', 'processing']
```

- **Explanation:**
  - `"hello world"` was split into `['hello', 'world']`.
  - `"spark is great"` was split into `['spark', 'is', 'great']`.
  - `"data processing"` was split into `['data', 'processing']`.
  - `flatMap` then combined all these individual words into a single flat RDD. Notice that the number of elements changed from 3 (sentences) to 7 (words).

Example 2: `flatMap` with a custom function (and filtering)

Let's say we have a list of numbers and we want to create a new RDD that contains each number and its square, but only for numbers greater than 5.

```
numbers_rdd = spark.sparkContext.parallelize([2, 6, 3, 8, 5, 10])

def process_number(num):
    if num > 5:
        return [num, num * num] # Return a list of two elements
    else:
        return [] # Return an empty list to filter out the element

rdd_processed_numbers = numbers_rdd.flatMap(process_number)

print("\nRDD after flatMap (processed numbers):")
print(rdd_processed_numbers.collect())
```

Output:

```
RDD after flatMap (processed numbers):
[6, 36, 8, 64, 10, 100]
```

- **Explanation:**
  - For `2` (not > 5), `process_number` returns `[]`, so `2` is effectively filtered out.
  - For `6` (> 5), `process_number` returns `[6, 36]`.
  - For `3` (not > 5), `process_number` returns `[]`.
  - For `8` (> 5), `process_number` returns `[8, 64]`.
  - For `5` (not > 5), `process_number` returns `[]`.
  - For `10` (> 5), `process_number` returns `[10, 100]`.
  - `flatMap` combines all these individual elements from the returned lists into a single flat RDD.

When to use `map` vs. `flatMap`

- **Use `map()` when:**
  - You want to transform each element independently into a single, corresponding output element.
  - The number of elements in the output RDD should be the same as the input RDD.
  - Examples: converting data types, performing simple arithmetic, formatting strings.
- **Use `flatMap()` when:**
  - You want to transform each element into zero or more output elements.
  - You need to "flatten" a collection of collections into a single collection.
  - The number of elements in the output RDD might be different from the input RDD.
  - Examples: tokenizing text (splitting sentences into words), expanding nested data structures.

In summary, `map` is for one-to-one transformations, while `flatMap` is for one-to-many (or one-to-zero) transformations that also flatten the resulting structure.

9. Broadcast join in pyspark, why would you use? Whats the use? What kind of optimization does it give?

Broadcast join is a optimization technique in PySpark that significantly improves the performance of join operations, especially when one of the DataFrames is considerably smaller than the other.

What is Broadcast Join?

- In a typical (shuffle-based) join, when two large DataFrames are joined, Spark shuffles data across the network.
- This means that rows with the same join key from both DataFrames are sent to the same executor node so they can be matched.
- This shuffling process involves significant network I/O and disk I/O (writing to and reading from disk), which can be very expensive and time-consuming for large datasets.

A **broadcast join** works differently:

1. **Identify the smaller DataFrame:** Spark identifies the smaller of the two DataFrames involved in the join.
2. **Broadcast the smaller DataFrame:** The entire content of this smaller DataFrame is collected by the Spark driver, serialized, and then broadcasted to *all* executor nodes in the cluster.
3. **Local Join:** Each executor node receives a complete copy of the smaller DataFrame in its memory. When an executor processes a partition of the *larger* DataFrame, it can perform the join operation locally with the broadcasted smaller DataFrame without needing to shuffle any data from the larger DataFrame across the network.

Why would you use Broadcast Join? What's the Use?

You would use a broadcast join primarily for **performance optimization** when:

1. **One DataFrame is significantly smaller than the other:** This is the most common and ideal scenario. Think of joining a large "fact" table (e.g., sales transactions) with a small "dimension" table (e.g., product categories, customer lookup, currency exchange rates).
2. **The smaller DataFrame can fit entirely into the memory of each executor:** This is a strict requirement. If the smaller DataFrame is too large to fit into memory, broadcasting it will lead to OutOfMemory errors and likely worse performance than a shuffle join.
3. **You want to minimize data shuffling:** Shuffling is the most expensive operation in Spark due to network and disk I/O. Broadcast joins drastically reduce or eliminate shuffling for the larger DataFrame.

Typical Use Cases:

- **ETL processes:** Joining large transaction logs with small reference data.
- **Data enrichment:** Adding descriptive attributes from small lookup tables to a large dataset.
- **Filtering:** Using a small list of IDs or criteria to filter a large table.
- **Star Schema / Snowflake Schema:** Joining fact tables with dimension tables in data warehousing.

What kind of optimization does it give?

The primary optimization provided by a broadcast join is the **reduction or elimination of data shuffling** for the larger DataFrame. This leads to several performance benefits:

1. **Reduced Network I/O:**
2. **Reduced Disk I/O:**
3. **Faster Join Execution:**
4. **Better Utilization of Resources:**

10. What is the syntax for a CTE at a high level, how to use the CTE, when will we use CTE, Why will we use it? How is it different from subquery

Syntax for a CTE (High Level)

The basic syntax for a CTE involves the `WITH` clause:

```
WITH <cte_name_1> AS (
    -- Your first CTE query definition
    SELECT column1, column2
    FROM your_table
    WHERE some_condition
),
<cte_name_2> AS (
    -- Your second CTE query definition (can reference cte_name_1)
    SELECT colA, colB
    FROM another_table
    JOIN <cte_name_1> ON ...
),
-- ... you can have multiple CTEs chained together
```

```
<cte_name_n> AS (
    -- Your Nth CTE query definition
    SELECT ...
)
-- The final query that uses one or more of the defined CTEs
SELECT final_column_x, final_column_y
FROM <cte_name_1>
JOIN <cte_name_2> ON ...
WHERE another_condition;
```

Key parts:

- `WITH` **clause:** This keyword introduces one or more CTE definitions.
- `<cte_name>`**:** This is the temporary name you give to your result set. It's good practice to choose descriptive names.
- `AS ( ... )`**:** This immediately follows the CTE name and contains the `SELECT` statement that defines the CTE's result set.
- **Comma (`,`):** If you define multiple CTEs, you separate them with commas.
- **Final Query:** After all CTEs are defined, there's a single main SQL statement (SELECT, INSERT, UPDATE, DELETE, MERGE) that uses the CTE(s).

How to Use a CTE?

Once defined, you use a CTE just like you would a regular table or view in the subsequent parts of your SQL query.

Example:

Let's say you have an `Employees` table with `EmployeeID`, `Name`, `DepartmentID`, and `Salary`.

```
-- Step 1: Define a CTE named 'HighPaidEmployees'
WITH HighPaidEmployees AS (
    SELECT EmployeeID, Name, Salary
    FROM Employees
    WHERE Salary > 70000
)
-- Step 2: Use the CTE in the final SELECT statement
SELECT Name, Salary
```

```
FROM HighPaidEmployees
ORDER BY Salary DESC;
```

In this example, `HighPaidEmployees` is a temporary result set containing employees earning over 70000. The final `SELECT` statement then queries this temporary result set as if it were a real table.

When Will We Use CTEs?

CTEs are incredibly useful in several scenarios, primarily for improving query structure and sometimes for performance.

1. **Readability and Maintainability:**
   - **Breaking Down Complex Logic:** When you have a very complex query with multiple steps or nested subqueries, CTEs allow you to break down the logic into smaller, named, logical blocks. This makes the query much easier to read, understand, and debug.
   - **Self-Documenting Code:** The names you give to CTEs can describe the purpose of each sub-result set, acting as self-documentation.
2. **Referencing the Same Result Set Multiple Times:**
   - If you need to use the result of a sub-query multiple times within the same main query, defining it once as a CTE avoids repetitive code. While some optimizers might handle this with subqueries, a CTE explicitly states your intent.
3. **Recursive Queries:**
   - CTEs are the **only way** to write recursive queries in standard SQL. This is essential for traversing hierarchical or tree-like data structures (e.g., organizational charts, bill of materials, network paths). A recursive CTE typically has an "anchor member" (base case) and a "recursive member" (recursive step), combined with `UNION ALL`.
4. **Before an `INSERT`, `UPDATE`, or `DELETE` statement:**
   - You can define a CTE to prepare the data to be inserted, updated, or deleted, and then use the CTE directly in the DML statement. This can be much cleaner than nesting complex subqueries within the DML.
5. **Simulating Views for One-Time Use:**
   - If you need a temporary, complex result set that acts like a view but you don't want to store it permanently in the database schema, a CTE is perfect.

How is it Different from a Subquery?

While both CTEs and subqueries allow you to embed one query within another, they differ significantly in their scope, readability, and capabilities.

Feature	CTE (Common Table Expression)	Subquery (Derived Table / Nested Query)
Definition	Defined using the `WITH` clause at the beginning of the query.	Can be defined almost anywhere within a query's `FROM`, `WHERE`, `SELECT`, `HAVING` clause.
Naming	Always named (e.g., `WITH MyCTE AS (...)`).	Can be named (`FROM (SELECT ...) AS MySubqueryAlias`) or unnamed (`WHERE column IN (SELECT ...)`).
Scope / Reusability	Can be referenced multiple times within the *same* main query.	Generally, only referenced once at the point of definition. If needed again, it must be rewritten (or nested further).
Readability	Significantly improves readability for complex queries by breaking logic into named, sequential steps.	Can quickly lead to deeply nested, hard-to-read, and understand queries, especially for multiple layers.
Recursion	**Supports recursion** (recursive CTEs).	**Does NOT support recursion.**
DML Statements	Can be used as the target of `INSERT`, `UPDATE`, `DELETE`, `MERGE` statements (database dependent, but common).	Can be used as the *source* for DML, but generally not directly as the target.
Optimization	Can sometimes provide better optimization hints to the query optimizer, especially when referenced multiple times. The optimizer *might* materialize it once.	Often treated as opaque blocks. Optimizers might have to re-evaluate them multiple times if nested and not well-optimized.
Materialization	Behavior varies by database. Some databases might materialize the CTE results once; others might re-evaluate them for each reference (similar to a view).	Generally, a subquery is evaluated for each context it's used in, though optimizers try to be smart.

Example of Subquery vs. CTE:

Subquery:

```
SELECT
    e.Name,
    e.Salary,
    dept.DepartmentName
FROM
    Employees e
JOIN
    (SELECT DepartmentID, DepartmentName FROM Departments WHERE Location =
'Mumbai') AS dept -- Subquery
ON
    e.DepartmentID = dept.DepartmentID;
```

Equivalent CTE:

```
WITH MumbaiDepartments AS (
    SELECT DepartmentID, DepartmentName
    FROM Departments
    WHERE Location = 'Mumbai'
)
SELECT
    e.Name,
    e.Salary,
    md.DepartmentName
FROM
    Employees e
JOIN
    MumbaiDepartments md -- CTE usage
ON
    e.DepartmentID = md.DepartmentID;
```

In the above simple example, the difference might seem minor. But imagine if `MumbaiDepartments` was a much more complex query, and you needed to join it with several other tables or perform further aggregations based on it. The CTE approach would immediately be much clearer.

11. What is .format()

The `.format()` method in Python is a versatile and powerful way to **format strings**.

Syntax and Examples

1. Positional Arguments:

The simplest way is to use empty curly braces `{}` as placeholders. The values are inserted in the order they appear as arguments to `.format()`.

```
name = "Alice"
age = 30
message = "My name is {} and I am {} years old.".format(name, age)
print(message)

# Output: My name is Alice and I am 30 years old.
```

12. What are the 2 arguments to map function

Function and iterable

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
print(list(squared_numbers))
# Output: [1, 4, 9, 16, 25]
```

13. What are the parameters of a filter function?

Function and iterable

```
def is_even(num):
    return num % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(is_even, numbers)
print(list(even_numbers))
# Output: [2, 4, 6, 8, 10]
```

14. When will you use a map, when will you use a flatmap

You use map when you want one output element for each input element.

You use flatMap when you want zero or more output elements for each input element, and you want to flatten the results into a single collection.

15. What does take command do in pyspark

In PySpark, the `take()` command is an **action** that allows you to retrieve a specified number of elements (rows) from an RDD or DataFrame and bring them to the **driver program** as a standard Python list.

What `take()` does:

1. **Retrieves N elements:** It fetches the first `n` elements (records/rows) from the RDD or DataFrame.
2. **Returns a Python list:** The result is a standard Python `list`. For DataFrames, each element in the list will be a `Row` object.
3. **Triggers computation:** As an action, `take()` forces the execution of all the preceding lazy transformations on the RDD or DataFrame to compute the required elements.
4. **Optimized for small `n`:** Spark is optimized to efficiently retrieve a small number of elements. It does this by scanning only the necessary partitions (often starting with the first one) until it gathers enough elements. This avoids scanning the entire dataset, which would happen with `collect()`.

Syntax:

- **For RDDs:** `rdd.take(num: int)`
- **For DataFrames:** `dataframe.take(num: int)`

Examples:

1. On an RDD:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("TakeExample").getOrCreate()
```

```
sc = spark.sparkContext

data = [10, 20, 30, 40, 50, 60, 70]
rdd = sc.parallelize(data)

# Take the first 3 elements
first_three = rdd.take(3)
print(f"First 3 RDD elements: {first_three}")

# Output: First 3 RDD elements: [10, 20, 30]
```

2. On a DataFrame:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType, Row

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

data = [
    ("Alice", 30, "NY"),
    ("Bob", 24, "LA"),
    ("Charlie", 35, "Chicago"),
    ("David", 28, "SF"),
    ("Eve", 22, "Houston")
]
schema = StructType([
    StructField("Name", StringType(), True),
    StructField("Age", IntegerType(), True),
    StructField("City", StringType(), True)
])
df = spark.createDataFrame(data, schema)

# Take the first 2 rows
first_two_rows = df.take(2)
print(f"First 2 DataFrame rows: {first_two_rows}")
# Output: First 2 DataFrame rows: [Row(Name='Alice', Age=30, City='NY'),
Row(Name='Bob', Age=24, City='LA')]

# You can access elements within the Row objects
for row in first_two_rows:
```

```
    print(f"Name: {row.Name}, Age: {row.Age}")
# Output:
# Name: Alice, Age: 30
# Name: Bob, Age: 24
```

When to use `take()`:

- **Sampling/Previewing Data:**
- **Debugging:**
- **Small Datasets:**

16. What is the difference between show and take?

- `show()` returns None. You call it for its side effect of printing.
- `take()` returns a list of Row objects (or whatever the elements of your RDD are). This means you can store that list in a variable and manipulate it with standard Python code.
- If you just want to look at your data, use `show()`.
- If you need to work with a small piece of your data in your Python script or interactively, use `take()` to get those actual data objects."

17. How do you list all files in a folder in hadoop

```
hdfs dfs -ls /path/to/your/folder
```

18.What is a delta lake?

Delta Lake is an open-source storage layer that brings reliability and performance to your data lake

It does this by adding ACID transactions, schema enforcement, and data versionin on top of your existing data files

19. What are the 3 different ways of parquet file encoding

1. **Delta Encoding**
2. **Dictionary Encoding (RLE_DICTIONARY)**
3. **Run-Length Encoding (RLE)**

1. Delta encoding

Delta encoding is a way to **store data more efficiently** by saving the **difference** between values instead of the values themselves.

Imagine this column of numbers:

```
[1000, 1001, 1002, 1005, 1008]
```

Instead of storing all of them as-is, **Delta Encoding** stores:

```
[1000, +1, +1, +3, +3]
```

- First value is stored as-is: `1000`
- After that, it stores the **difference (delta)** from the previous value:
  - 1001 - 1000 = `+1`
  - 1002 - 1001 = `+1`
  - 1005 - 1002 = `+3`
  - 1008 - 1005 = `+3`
- This is more **compact** and **compresses better**, especially when values increase gradually.

2. Dictionary Encoding (RLE_DICTIONARY)

- **How it works:** This is highly effective for columns with a **low cardinality** (i.e., a small number of unique/distinct values) or many **repeated values**.
  1. A "dictionary" of all unique values in the column is created and stored separately.
  2. Each actual value in the column is then replaced by a small integer "index" that points to its position in the dictionary.
  3. The dictionary itself is usually Plain encoded. The indices are then encoded using Run-Length Encoding / Bit-Packing.
- **When it's used:**

    - **Categorical data:** (e.g., `gender` ('Male', 'Female'), `status` ('Active', 'Inactive', 'Pending'), `country` codes).
    - Columns with many repeated strings or numbers.
- **Benefit:**
    - **Significant storage reduction:** Storing small integers (indices) is much more efficient than storing repeated large strings or complex values.
    - **Faster queries:** Query engines can often operate on the smaller integer indices directly, and only look up the actual values in the dictionary when needed for output. This reduces I/O.
- **Drawback:** Not effective for high-cardinality columns (e.g., unique IDs, timestamps with milliseconds) because the dictionary itself would become very large, negating the benefits.

3. Run-Length Encoding (RLE) /

- **Run-Length Encoding (RLE):** Efficiently stores sequences of repeating values. Instead of storing `A, A, A, A, B, B`, it stores `(A, 4), (B, 2)`.
- **Benefit:**
    - **High compression for repeating sequences:** Excellent for data with many consecutive identical values.
    - **Efficient storage of small integers:** Bit-packing reduces the footprint of small integer values.
- **Drawback:** Less effective for truly random data without many consecutive repetitions or where values require a large number of bits to represent.

20. What is partitioning and bucketing

Partitioning

- Partitioning is the process of dividing a large dataset into smaller, more manageable parts based on the values of one or more columns (called partition keys).
- Each partition is stored as a separate folder in the filesystem.
- If you have a sales table with millions of records, you can partition it by country or year.
    - /sales/country=India/
    - /sales/country=USA/
    - /sales/country=UK/

- Each folder contains the sales data only for that specific country.

Bucketing

- Bucketing further divides the data inside each partition (or entire table, if not partitioned) into fixed number of buckets, based on the hash of a column.

21. Inline dataset in dataflow?

An inline dataset in Azure Data Factory (ADF) or Synapse Data Flows refers to dataset definitions that are embedded directly inside a data flow source or sink, rather than being defined separately as reusable dataset objects

22. What does collect do?

- In PySpark, the collect() function retrieves the entire dataset (all rows) from a distributed DataFrame (or RDD) and brings it to the driver program as a list.
- Use collect() only when:
  - The dataset is small.

23. What is DAG in pyspark? How does it help

What is DAG?

- In PySpark, a DAG (Directed Acyclic Graph) is a logical representation of the sequence of computations (transformations) performed on your data.
- It tracks how data should flow and be processed, from the source to the final output.
- Each node in the DAG represents a RDD (Resilient Distributed Dataset) or DataFrame, and each edge represents a transformation (like map, filter, join, etc.).

Example

When you write a PySpark job like:

```
df = spark.read.csv("data.csv")
df2 = df.filter(df["age"] > 30)
df3 = df2.groupBy("country").count()
```

Nothing is actually executed yet — PySpark builds a DAG internally that remembers:

Read from CSV → Filter rows → Group and count

Only when you call an action like .show() or .collect(), the DAG is submitted to the Spark DAG Scheduler for execution.

How does DAG help?

- **Optimized Execution:** DAG enables Spark to analyze and optimize the full chain of transformations before running them.
- **Lazy Evaluation:** Operations are not executed until an action is called, which lets Spark plan the most efficient execution.
- **Parallelism:** The DAG is split into stages and tasks, which Spark runs in parallel across cluster nodes.

24. External and internal table or managed table in Hive?

1. Internal Table (Managed Table)

- An internal table (also called a managed table) is where Hive manages both the metadata and the data.
- When you drop the table, both the table definition and the data in HDFS are deleted.

2.External Table

- An external table is when Hive manages only the metadata, but the actual data is stored outside Hive's control (like in a shared HDFS folder).
- When you drop the table, only metadata is deleted, but data remains untouched

25. What If I drop an external table or managed table? What will happen?

1. DROP MANAGED (Internal) Table

- Deletes metadata from Hive metastore.
- Deletes actual data from HDFS (from warehouse directory).

2. DROP EXTERNAL Table

- Deletes metadata from Hive metastore.
- Keeps the actual data in HDFS untouched.

26. What is HDFS?

- HDFS stands for Hadoop Distributed File System.
- It is the primary storage system used by Hadoop to store large-scale data across a distributed cluster of machines.

Features

- **Distributed Storage:** Stores data across many machines
- **Block-based:** Files are split into fixed-size blocks (default: 128 MB)
- **Fault Tolerance:** Each block is replicated (usually 3 copies) across different machines
- **High Throughput:** Designed for batch processing rather than low-latency real-time access
- **Write Once, Read Many:** Optimized for once-written, frequently-read access pattern

27. In pyspark why do we create a DAG?

- In PySpark, the DAG (Directed Acyclic Graph) is automatically created when you define a series of transformations on data
- The DAG is created to optimize, schedule, and execute your Spark job efficiently and fault-tolerantly.

Lazy Evaluation: Defers execution until needed
Optimization: Builds efficient plan before execution
Fault Tolerance: Uses lineage to recompute lost data

Parallelism: Splits work into stages and tasks
Resource Management: Schedules jobs efficiently on the cluster

28. Difference between action and transformation

Transformation

- A transformation is an operation that defines a new RDD/DataFrame, but doesn't execute immediately. It builds a DAG (Directed Acyclic Graph) of planned operations.
- Transformations are lazy — Spark waits until an action is called to actually process the data
- Examples
  - `filter()` Select rows that meet a condition
  - `map()` Apply a function to each row
  - `select()` Choose specific columns
  - `groupBy()` Group rows based on a key
  - `withColumn()` Add or modify a column

Action

- An action is an operation that triggers execution of all previous transformations and returns a result (or writes output).
- Examples
  - `show()` Displays the top rows
  - `collect()` Brings all data to driver as list
  - `count()` Returns number of rows
  - `write()` Saves data to storage
  - `take(n)` Returns first n rows as list

29. What is a overalapping window? When do we use it? Whats the usecase

- In Azure Data Factory (ADF), an overlapping window refers to a data processing window (usually in triggered pipelines or tumbling window triggers) where time intervals overlap

between consecutive executions.

- This means each pipeline run may process data that intersects with previous or next runs, instead of strictly non-overlapping time blocks.

Example

- Let’s say you set a tumbling window trigger with:
  - Window size: 1 hour
  - Overlapping buffer: 15 minutes
- Each pipeline run processes data from:
  - 9:00 AM to 10:00 AM
  - 9:45 AM to 10:45 AM
  - 10:30 AM to 11:30 AM
- So the window slides by 45 minutes, and each run overlaps with the previous.
- Late-arriving data If data meant for 10:00 AM arrives at 10:20 AM, the next overlapping window still picks it up.

30. What are the 3 different tabs, where do you create a linked service

1. Author (or Author & Monitor)

This is where you build and edit your data pipelines, datasets, dataflows, triggers, notebooks, and other resources.

You create and manage your Linked Services here as well.

2. Monitor

Used to track pipeline runs, trigger runs, debug runs, and monitor overall health of your data factory.

You can see logs, successes, failures, and performance stats.

3. Manage

Here you manage global factory settings like Linked Services, Integration Runtimes, Git configuration, Access control, and Triggers.

You can also create and configure Linked Services here

Where to Create a Linked Service?

In Azure Data Factory Studio:
You can create a Linked Service in either the Manage tab (recommended) or the Author tab under Connections > Linked Services.

31. What is the output a flatmap would give?

- The flatMap() function in PySpark is used to transform each input element into 0, 1, or many output elements.
- Then flatten all those outputs into one continuous collection — no nested lists.

32. What is cardinality of data

- Cardinality refers to the number of unique values in a column of a dataset or table.
- It helps describe how distinct the data is in a given column.
- **High Cardinality** Column has many unique values Email addresses, Social Security Numbers, UUIDs
- **Low Cardinality** Column has few unique values Gender, Boolean flags (Yes/No), Country
- **Medium Cardinality** Moderate number of unique values Zip codes, Product categories

33. Cardinality in terms of bucketing and partitioning. What is the cardininality we choose for bucketing and partitioning?

- In the context of Hive and Spark, cardinality refers to the number of unique values in a column.
- It plays a crucial role in deciding whether a column should be used for bucketing or partitioning.
- Partitioning Columns with low cardinality Creates one folder per value. Too many unique values = too many small folders/files.

- Bucketing Columns with high or medium cardinality Hashes values into fixed number of buckets (files) — avoids metadata explosion.