1. What is PySpark, and how does it work?

PySpark is the Python API for Apache Spark, an open-source distributed computing system designed for big data processing and analytics. PySpark allows you to write Spark applications using Python, making it easier for Python developers to work with large datasets across a cluster of machines.

Apache Spark Core:

Spark is written in Scala and runs on the Java Virtual Machine (JVM).

PySpark acts as a bridge between Python code and the Spark engine.

Python Driver Program:

You write Python code using PySpark.

This code creates a SparkContext or SparkSession to connect to a Spark cluster.

Communication via Py4J:

PySpark uses Py4J, a library that enables Python programs to dynamically access Java objects.

Your Python code communicates with the JVM-based Spark engine under the hood.

RDD and DataFrame APIs:

You can work with RDDs (Resilient Distributed Datasets) for low-level operations or use DataFrames for higher-level SQL-like operations.

DataFrames are more optimized and easier to use.

Distributed Processing:

Spark splits your data and code into tasks and distributes them across worker nodes in a cluster.

Workers perform computations in parallel, boosting performance for large datasets.

Lazy Evaluation:

PySpark builds a DAG (Directed Acyclic Graph) of transformations and only executes them when an action (like .collect(), .show(), .write()) is called.

✅ Key Features:

Parallel computing: Handles massive data volumes.

Scalability: Works on a local machine or a massive cluster.

Built-in support for SQL, ML, Streaming, Graphs via libraries like:

pyspark.sql

pyspark.ml

pyspark.streaming

pyspark.graphframes

2. Explain the architecture of PySpark.

1. Driver Program (Python)

Where your PySpark code runs.

Contains the SparkSession or SparkContext.

Converts operations into a DAG (Directed Acyclic Graph).

Sends tasks to the cluster manager.

2. PySpark API (Python Layer)

Provides classes like DataFrame, RDD, SparkSession, etc.

Uses Py4J to translate Python commands into JVM calls.

3. Py4J Gateway

A communication bridge between the Python process and the JVM process.

Converts Python commands into Java bytecode calls and vice versa.

4. Spark Driver (JVM)

The actual master node.

Handles:

Task scheduling

Job coordination

Fault tolerance

5. Cluster Manager

Manages resources across the cluster.

Allocates Executors to the Driver.

Types:

Standalone

YARN

Mesos

Kubernetes

6. Executors (on Worker Nodes)

Run the tasks sent by the driver.

Perform transformations and actions.

Store data in memory or disk (for caching/persistence)

3.What are the differences between RDD, DataFrame, and Dataset?

Feature	RDD (Resilient Distributed Dataset)	DataFrame	Dataset (Scala/Java Only)
Language	Available in **Python**, Scala, Java	Available in **Python**, Scala, Java	**Only in Scala and Java**
Abstraction	Low-level	High-level (like a SQL table)	Typed high-level abstraction
Schema	No schema (just objects)	Has schema (like a table)	Strongly typed schema
Performance	Slower (no optimization)	Optimized via Catalyst & Tungsten	Optimized & strongly typed
Ease of Use	Complex to use (more code)	Simple with rich APIs	More complex than DataFrame
Type Safety	No (Python is dynamic)	No (Python), Yes (Scala)	Yes (Scala/Java only)
Serialization	Java serialization	Opti ↓ d memory format	Encoders (faster than RDD)

4. How does Spark handle fault tolerance?

1. RDD Lineage (Core to Fault Tolerance)

- Spark keeps track of the **lineage** of an RDD — basically, a **record of all transformations** (like map, filter, join) that were used to create it. If part of an RDD is lost (e.g., due to node failure), **Spark can recompute it** from the original data using this lineage.

- All transformations in Spark are **lazy**. When you perform an action (like `collect()` or `count()`), Spark builds a **DAG (Directed Acyclic Graph)** of stages and tasks.If a task fails, Spark **knows exactly how to recompute** it using the DAG plan.

- If an **executor fails**, only the **lost tasks** are rescheduled on other nodes. Spark doesn't restart the entire job — only the failed parts are rerun, saving a lot of time.

- • For long lineage chains or iterative computations (like with ML algorithms), lineage can get expensive.Spark supports **checkpointing**, where it saves an RDD to **reliable storage (like HDFS)**.If a failure occurs, Spark **reads from the checkpoint** instead of recomputing from scratch

5. What are the different types of transformations in PySpark?

Narrow Transformations

- Data required to compute the records in a single partition **reside in a single partition** of the parent RDD/DataFrame.
- No **shuffling** of data between partitions.
- **Fast and efficient.**

Wide Transformations

- Data from **multiple partitions** are required to compute a single partition of the result.
- **Triggers a shuffle**, i.e., redistributing data across the cluster.
- **Slower and more expensive**, but necessary for some operations.

Type	Examples	Shuffle?
Narrow	`map`, `filter`, `union`, `sample`	❌
Wide	`groupBy`, `join`, `reduceByKey`, `repartition`	✅

Transformation	Description
`select()`	Selects columns
`withColumn()`	Adds or modifies a column
`drop()`	Removes columns
`filter()` / `where()`	Filters rows
`groupBy().agg()`	Group and aggregate
`join()`	Joins DataFrames
`orderBy()` / `sort()`	Sorts the DataFrame
`dropDuplicates()`	Removes duplicate rows

6. Explain the significance of the DAG (Directed Acyclic Graph) in Spark.

A **Directed Acyclic Graph (DAG)** in Spark represents a **logical execution plan** of a Spark job. It's a graph where:

- **Nodes** represent **RDDs/DataFrames** created at each transformation step.
- **Edges** represent the **dependencies** between them (i.e., how one RDD is derived from another).
- It is **directed** (has a clear flow of execution) and **acyclic** (no loops or cycles).

1. Tracks Transformations

Spark uses the DAG to **track all transformations** (like map, filter, join) in your program.

2. Enables Lazy Evaluation

Instead of executing each transformation immediately, Spark **builds a DAG** of operations. This:

- Delays execution until an action (collect(), save(), etc.) is triggered.
- Helps optimize the entire workflow.

3. Optimizes Execution

Before running the job, Spark's **DAG Scheduler**:

- Splits the DAG into **stages** (based on narrow and wide transformations).
- Each stage is divided into **tasks** for parallel execution.
- Schedules tasks intelligently to **minimize data shuffling** and maximize locality.

4. Supports Fault Tolerance

If a task or node fails, Spark can **recompute lost data** using the DAG lineage, rather than restarting everything.

Example: Simple DAG Flow

python Copy Edit

```python
rdd = spark.sparkContext.textFile("file.txt")
words = rdd.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("output")
```

DAG View:

arduino Copy Edit

```
textFile → flatMap → map → reduceByKey → saveAsTextFile
```

- Each step becomes a **node**.
- `reduceByKey` triggers a **shuffle**, so it's a stage boundary.
- When `saveAsTextFile()` is called, Spark builds the DAG and executes it.

🔁 DAG vs Traditional MapReduce

Feature	Spark DAG	Traditional MapReduce
Execution Plan	Entire DAG upfront	One stage at a time
Optimization	End-to-end job optimization	Limited optimization
Performance	Faster (in-memory, fewer reads/writes)	Slower (disk I/O between stages)
Flexibility	Fine-grained transformations	Fixed map → shuffle → reduce

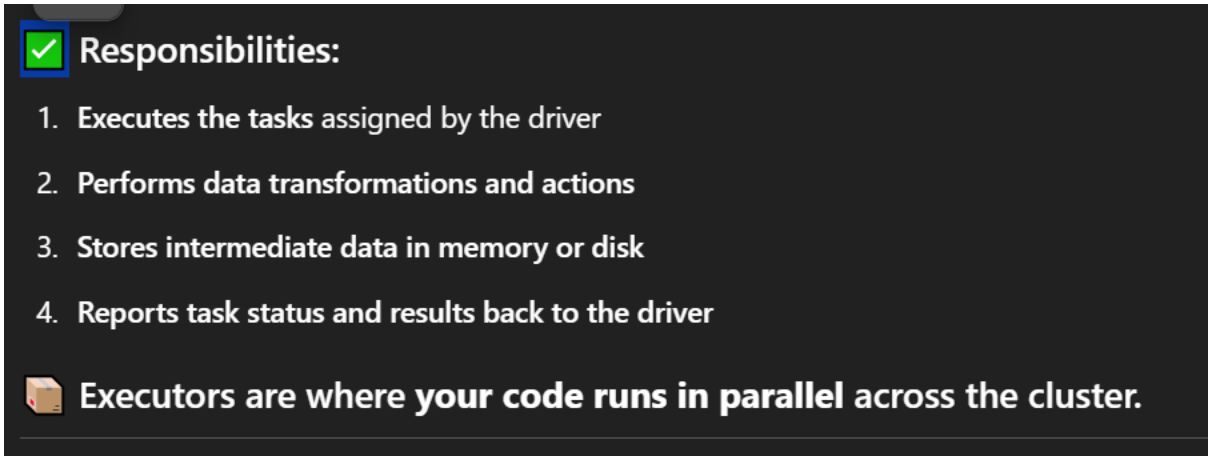
7. is the role of the driver and executor in PySpark?

Component	Role	Where It Runs
Driver	Orchestrates the whole application	On the master node (your machine in local mode)
Executor	Executes tasks assigned by the driver	On worker nodes

The **Driver** is the **master process** that:

✅ Responsibilities:

1. **Creates the SparkSession / SparkContext**
2. **Parses your PySpark code**
3. **Builds a DAG (Directed Acyclic Graph)**
4. **Plans execution (stages & tasks)**
5. **Communicates with the cluster manager** (e.g., YARN, Standalone, Kubernetes)
6. **Distributes tasks to executors**
7. **Collects results from executors**



9. How does PySpark handle schema inference?

1. Enabled by Default for Some Formats

- For formats like **CSV** or **JSON**, you can enable schema inference with inferSchema=True.

When you enable inferSchema=True, Spark does this:

1. **Reads the header** (if header=True)
2. **Scans the first few rows** to sample data types
3. **Infers each column's data type**
4. Builds a StructType schema internally

Schema inference **requires reading a subset of the data** to guess types, which **adds overhead**.For large files, it can slow things down.

Solution: Define schema manually for better performance.

```
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])
```

10.What are the different types of joins in PySpark?

Join Type	Description
inner	Returns only matching rows from both DataFrames
left / left_outer	Returns all rows from the left DataFrame, and matching rows from the right
right / right_outer	Returns all rows from the right DataFrame, and matching rows from the left
full / full_outer	Returns all rows when there is a match in one of the DataFrames
left_semi	Returns only rows from the left DataFrame **that have a match** in the right DataFrame
left_anti	Returns only rows from the left DataFrame **that do not have a match** in the right
cross	Returns the Cartesian product of both DataFrames (every row from left paired with every r from right)

```
df1 = spark.createDataFrame([(1, "A"), (2, "B"), (3, "C")], ["id", "value1"])
df2 = spark.createDataFrame([(1, "X"), (3, "Y"), (4, "Z")], ["id", "value2"])

# Inner join
df1.join(df2, on="id", how="inner").show()

# Left outer join
df1.join(df2, on="id", how="left").show()

# Right outer join
df1.join(df2, on="id", how="right").show()

# Full outer join
df1.join(df2, on="id", how="full").show()

# Left semi join
df1.join(df2, on="id", how="left_semi").show()

# Left anti join
df1.join(df2, on="id", how="left_anti").show()

# Cross join
df1.crossJoin(df2).show()
```

11. How does `cache()` differ from `persist()` in PySpark?

Feature	`cache()`	`persist()`
Storage Level	`MEMORY_AND_DISK` (default only)	You can choose from multiple levels
Flexibility	Less flexible	More flexible
Usage	Simple one-line caching	Useful when you want custom behavior
Under the hood	Shortcut for `persist(MEMORY_AND_DISK)`	Explicit call to `persist(level)`

Storage Level	Description
`MEMORY_ONLY`	Store only in memory. Recompute if not enough space.
`MEMORY_AND_DISK`	Default. Spill to disk if needed.
`MEMORY_ONLY_SER`	Store serialized in memory (more space-efficient).
`DISK_ONLY`	Store only on disk.
`MEMORY_AND_DISK_SER`	Serialize and store in memory, spill to disk if needed.
`OFF_HEAP` (experimental)	Use off-heap memory (needs special config).

12. What is the significance of `broadcast` in Spark?

The **broadcast()** function in Spark allows you to **broadcast small datasets** (i.e., data that can fit into memory) to all the nodes in the cluster. This is particularly useful when one of the datasets in a join operation is small and you want to **avoid shuffling** that data.

When you broadcast a dataset, Spark sends it to each worker node, and each node keeps a **local copy** of that dataset. This eliminates the need for **data shuffling** between nodes during operations like **joins**, leading to a significant performance boost.

```
from pyspark.sql.functions import broadcast

# Small DataFrame (lookup table)
small_df = spark.read.csv("lookup.csv", header=True)

# Large DataFrame
large_df = spark.read.csv("big_data.csv", header=True)

# Broadcast the small DataFrame to avoid shuffling
result = large_df.join(broadcast(small_df), on="key")
```

13. How do you optimize Spark jobs for performance?

1. Use Spark's Built-in Optimizations

Spark comes with several built-in optimization techniques that you can leverage for better performance

- **Catalyst Optimizer**: Automatically optimizes logical query plans (e.g., predicate pushdown, constant folding, etc.) to minimize expensive operations.
- **Tungsten**: A Spark execution engine that optimizes physical query plans and improves memory management by using **off-heap memory** and **code generation**.

2. Avoid Shuffling Data

Shuffling is a very expensive operation because it involves disk and network I/O. To reduce shuffling, follow these tips:

- **Avoid unnecessary** `groupBy` or `join` operations that cause shuffling.
- **Use** `broadcast()` for small datasets during joins to eliminate shuffling.
- **Use** `coalesce()` **instead of** `repartition()` when reducing the number of partitions to avoid unnecessary shuffling.

3. Use `persist()` and `cache()` Wisely

- **Persist** or **cache** DataFrames/RDDs that you reuse multiple times to avoid recomputing them.
- **Choose an appropriate storage level** based on your needs (e.g., `MEMORY_ONLY`, `MEMORY_AND_DISK`).

4. Partition Data Efficiently

Partitioning your data appropriately is critical for minimizing the overhead of distributing data across nodes.

- **Avoid small partitions**: If your data is too small, Spark creates too many partitions, leading to overhead in task scheduling.
- **Use** `repartition()` **or** `coalesce()`: When dealing with large datasets, repartitioning can parallelism. Use `coalesce()` to reduce partitions when you're down to a small number of

14. What are the benefits of partitioning in PySpark?

Partitioning is how Spark **divides your data** into smaller, manageable chunks across the **worker nodes** in a cluster.

Each partition is processed in **parallel**, which allows Spark to **distribute the workload** and **maximize resource usage**.

1. Improved Parallelism & Speed

- Spark processes **each partition in parallel** across its worker nodes.
- More partitions = more concurrent tasks = faster execution (up to a point).

📌 Example: 1 billion rows split into 100 partitions will finish faster than 10 big partitions.

---

2. Reduced Data Shuffling

- **Smart partitioning** (e.g., by join keys or grouping columns) minimizes data movement (a.k.a. shuffle) across the network during joins and aggregations.

📉 Less shuffle = faster jobs & less cluster strain.

---

3. Efficient Joins and Aggregations

- If both datasets are **partitioned by the same key**, Spark can **co-locate data**, eliminating the need to reshuffle.

🚀 This leads to **more efficient joins** and **groupBy** operations.

---

4. Better Memory Utilization

- Proper partition sizes prevent memory overload.
- Spark won't try to load massive data chunks into memory in a single go.

📦 Smaller, well-balanced partitions = fewer out-of-memory errors.

---

5. Control Over Resource Allocation

- You can adjust partition sizes based on:
    - Number of executors
    - Size of data
    - CPU and memory per task

🛠️ This gives you **fine-tuned control** over how your job uses resources.

---

6. Optimized File Writing

- When writing out data (e.g., Parquet), partitioning helps:
    - Organize the data layout (e.g., partition by date, region)
    - Speed up query performance (predicate pushdown)

📁 Partitioned files = faster reads later.

15. What is the difference between `repartition()` and `coalesce()`?

Feature	`repartition()`	`coalesce()`
Main Purpose	**Increase or decrease** partitions	**Decrease only** (without full shuffle)
Shuffle?	✅ Yes – performs a **full shuffle**	🚫 No shuffle (or minimal), unless necessary
Use Case	When you want **even-sized partitions**	When you're **reducing** partitions (e.g., before writing output)
Performance	🐢 Slower (due to shuffle)	⚡ Faster (avoids shuffle)
Custom Partitioning	Allows specifying partitioning column(s)	Does **not** allow specifying partition columns
Data Distribution	More balanced	May result in **uneven partitions**

16. How do you handle skewed data in PySpark?

1. Broadcast Join (for small table joins)

If one table is small enough, broadcast it to all executors to **avoid shuffling** the big table.

2. Salting Keys (for skewed join keys)

If a **few keys are causing the skew**, add a random "salt" value to distribute them across partitions.

Step-by-step:

1. Add a random salt column to the large table.
2. Duplicate skewed keys in the small table with matching salt values.
3. Join on both original key and salt.

✅ 3. Skewed Key Detection and Separate Handling

Split out the **skewed key(s)** and handle them separately:

python Copy Edit

```python
# Separate skewed and non-skewed data
skewed_key = "123"
df_skewed = df.filter(df.customer_id == skewed_key)
df_others = df.filter(df.customer_id != skewed_key)

# Handle each separately
joined_skewed = df_skewed.join(other_df, "customer_id")
joined_others = df_others.join(other_df, "customer_id")

# Combine results
result = joined_skewed.union(joined_others)
```

✅ 4. Repartition Based on a Different Column

If the skew is caused by joining or aggregating on a certain column, repartition using a less skewed one (if possible):

python Copy Edit

```python
df = df.repartition("region")  # instead of a skewed column like customer_id
```

16.How does Spark execute operations in a lazy manner?

1. You apply transformations

- filter, select, withColumn, join, etc.
- These are just **instructions**; Spark builds a **Directed Acyclic Graph (DAG)**.

2. Spark builds a logical plan

- Spark records **what** you want to do, **not how**.

3. You trigger an action

- Examples: show(), collect(), write(), count()
- This is the moment Spark **figures out the execution plan**, **optimizes** it, and **runs the job**.

4. Spark applies optimizations

- **Catalyst Optimizer** rewrites your query plan (e.g., predicate pushdown, reordering filters).
- Spark then creates a **physical plan** and **executes it in stages** across the cluster.

17. How do you optimize join operations in PySpark?

1. Use broadcast() for small tables

If one of your tables is **small enough to fit in memory**, Spark can send it to all executors.

2. **Select only necessary columns before joining(Column Pruning)**

2. Select only necessary columns before joining

Reduce data size **before** the join — don't bring unnecessary baggage:

python Copy Edit

```python
filtered_df = df.select("id", "important_column").filter("status = 'active'")
```

3. Filter early (push-down predicates)

Apply filters **before** the join:

python Copy Edit

```python
df1_filtered = df1.filter("region = 'US'")
df2_filtered = df2.filter("status = 'active'")
joined = df1_filtered.join(df2_filtered, on="customer_id")
```

5. Avoid wide joins when possible

If you can do multiple **narrow joins** (e.g., join small tables first), it's better than one large, wide join with many columns.

6. Handle skewed data explicitly

If certain keys are very frequent (like customer_id = 9999), Spark will struggle.

Fix with:

- **Salting**
- **Broadcasting the small side**
- **Separating skewed keys**

7. Use appropriate join type

- Use `inner` instead of `outer` or `full` joins when possible — they're **less expensive**.

18. What is Adaptive Query Execution (AQE) in Spark?

AQE is a feature in Spark (enabled by default from Spark 3.0+) that allows Spark to **adjust and optimize the query execution plan dynamically at runtime**, based on real data characteristics.

Feature	Description
🔁 Dynamic Join Strategy Switching	Chooses between **sort-merge join** and **broadcast join** at runtime
🧠 Dynamic Partition Pruning	Skips unnecessary partitions in **partitioned tables**
📉 Optimizing Skewed Joins	Detects and splits **skewed partitions** dynamically
✂️ Reducing Shuffle Partitions	Merges small shuffle partitions to reduce overhead

19. Write a PySpark program to find the top 3 most occurring words in a given dataset.

How do you perform word count using PySpark?

```
text_df = spark.read.text("path/to/your/input.txt")  # Replace with actual p

# Step 3: Tokenize, normalize, and explode the words
words_df = text_df.select(
    explode(
        split(lower(col("value")), "\\W+")  # \W+ splits on non-word characters
    ).alias("word")
).filter(col("word") != "")  # Remove empty strings

# Step 4: Group by word and count
word_counts = words_df.groupBy("word").count()

# Step 5: Sort and get top 3
top_words = word_counts.orderBy(col("count").desc()).limit(3)
```

20. Given a DataFrame, write a PySpark code to remove duplicate records.

```
df_unique = df.dropDuplicates(["name", "department"])
```

```
# Remove full row duplicates
df_no_duplicates = df.distinct()
```

21. Write a PySpark job to group by a column and calculate the average value.

```
data = [
    ("Alice", "HR", 3000),
    ("Bob", "IT", 4000),
    ("Charlie", "HR", 3500),
    ("David", "IT", 4500),
    ("Eve", "Finance", 5000)
]
columns = ["name", "department", "salary"]

df = spark.createDataFrame(data, columns)

# Step 3: Group by 'department' and calculate average salary
avg_salary_df = df.groupBy("department").agg(avg("salary").alias("avg_salary"))
```

22. How do you handle missing/null values in a PySpark DataFrame?

```
# Check for null values in a specific column
df.filter(col("column_name").isNull()).show()

# Check for rows with nulls in any column
df.filter(df.isNull()).show()
```

```
df.filter(df["column_name"].isNull()).count()
```

```
df_cleaned = df.na.drop()  # Drops rows with any null values
```

```
df_cleaned = df.na.drop(subset=["column1", "column2"])
```

```
mean_salary = df.select(mean("salary")).collect()[0][0]  # Calculate mean salary
df_filled = df.na.fill({"salary": mean_salary})
```

23. Write a PySpark program to count distinct values in a column.

```
distinct_count = df.select(countDistinct("department").alias("distinct_department_count"))
```

24. Given a DataFrame, write a PySpark code to filter records where salary > 50,000.

```
filtered_df = df.filter(col("salary") > 50000)
```

25. Write a PySpark job to read a JSON file and convert it into a DataFrame.

```
# Step 2: Read JSON file (multiline format)
df = spark.read.option("multiline", "true").json("path/to/people.json")

# Step 3: Flatten the nested structure
flattened_df = df.select(
    "name",
    "age",
    "address.city",
    "address.zipcode",
    explode("skills").alias("skill")  # Explode array into multiple rows
)
```

26. Write a PySpark query to find the second highest salary from an employee table.

```
# Step 3: Define window spec
window_spec = Window.orderBy(col("salary").desc())

# Step 4: Apply dense_rank to rank salaries
ranked_df = df.withColumn("rank", dense_rank().over(window_spec))

# Step 5: Filter for second highest salary
second_highest_salary_df = ranked_df.filter(col("rank") == 2)
```

27. Write a PySpark job to join two DataFrames and select specific columns.

```
# Step 3: Join on emp_id
joined_df = emp_df.join(sal_df, on="emp_id", how="inner")
```

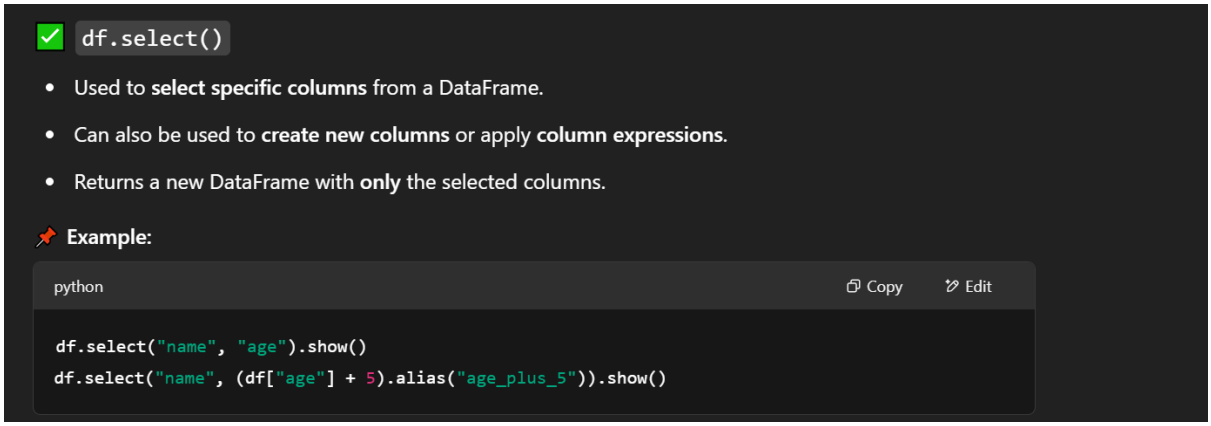
28. How do you create a temporary view in PySpark?

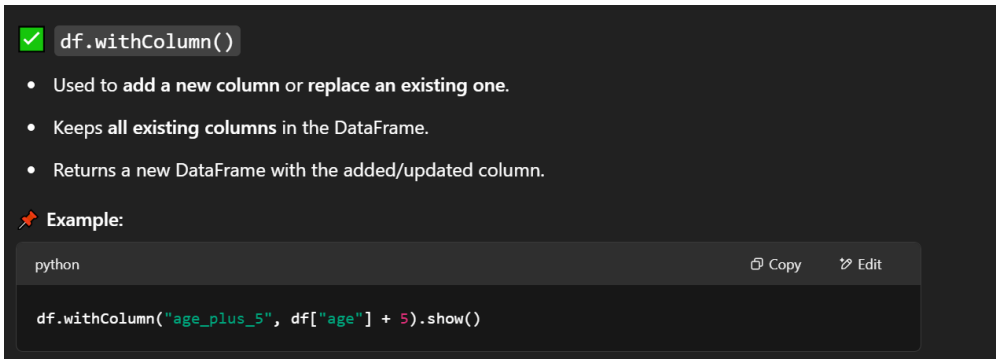
```
df = spark.createDataFrame(data, columns)

# Step 3: Create a temporary view
df.createOrReplaceTempView("people_view")

# Step 4: Use Spark SQL to query the temp view
result = spark.sql("SELECT name FROM people_view WHERE age > 30")
```

29. What is the difference between `df.select()` and `df.withColumn()`?





30. How do you write and execute SQL queries in PySpark?

```
result = spark.sql("SELECT name, age FROM people WHERE age > 30")
```

31. What is the difference between `explode()` and `posexplode()`?

✅ `explode()`

The `explode()` function **explodes** an array or map into multiple rows, with one element from the array or map per row.

- **Return Type**: A single column containing the exploded elements.
- It doesn't provide the **index/position** of the element in the array.

```
+-----+-----------------+-------+
| name|           skills|  skill|
+-----+-----------------+-------+
|Alice|[Python, Spark, SQL]| Python|
|Alice|[Python, Spark, SQL]|  Spark|
|Alice|[Python, Spark, SQL]|    SQL|
|  Bob|    [Scala, Hadoop]|  Scala|
|  Bob|    [Scala, Hadoop]| Hadoop|
```

✅ `posexplode()`

The `posexplode()` function is similar to `explode()`, but it **also returns the position/index** of each element in the array or map along with the value.

- **Return Type**: Two columns:
  1. **Position**: Index of the element in the array.
  2. **Value**: The element itself.

pgsql

```
+-----+-----------------+---+-------+
| name|           skills|pos|  skill|
+-----+-----------------+---+-------+
|Alice|[Python, Spark, SQL]|  0| Python|
|Alice|[Python, Spark, SQL]|  1|  Spark|
|Alice|[Python, Spark, SQL]|  2|    SQL|
|  Bob|    [Scala, Hadoop]|  0|  Scala|
|  Bob|    [Scala, Hadoop]|  1| Hadoop|
+-----+-----------------+---+-------+
```

32. How do you convert a DataFrame into a SQL table?

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

# Step 3: Create a temporary view
df.createOrReplaceTempView("people_view")

# Step 4: Use Spark SQL to query the temp view
result = spark.sql("SELECT name FROM people_view WHERE age > 30")
```

```
df.write.saveAsTable("my_permanent_table")
```

```
df.write.format("parquet").saveAsTable("my_parquet_table")
```

33. How do you use window functions in PySpark?

```
windowSpec = Window.partitionBy("department").orderBy(df["salary"].desc())
```

```
df.withColumn("row_num", row_number().over(windowSpec)).show()
```

34. Write a PySpark SQL query to get the cumulative sum of a column.

```
window_spec = Window.partitionBy("department").orderBy("date") \
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)
```

```
df_cumsum = df.withColumn("cumulative_sales", sum("sales").over(window_spec))
df_cumsum.show()
```

35. What is the difference between `groupBy()` and `rollup()`?

🔷 `groupBy()`

- Performs **standard group aggregation**.
- You get **only** the combinations of group keys that exist in the data.

✅ **Example:**

python Copy Edit

```python
df.groupBy("department", "year").sum("sales")
```

🔷 `rollup()`

- Performs **multi-level aggregation** (a hierarchical grouping).
- Includes **subtotals and grand totals**.

✅ **Example:**

python Copy Edit

```python
df.rollup("department", "year").sum("sales")
```

This returns:

- Sum of sales for each `(department, year)` pair
- Sum of sales **per department** across all years
- Sum of sales for **all departments and years** (grand total)

Feature	`groupBy()`	`rollup()`
Scope	Single-level grouping	Multi-level (hierarchical) grouping
Subtotals	❌ No	✅ Yes (subtotals and grand totals included)
Null rows	❌ No rows with `NULL`	✅ Adds `NULL` to represent grouped rollups
Use case	Standard aggregations	Reports, pivot tables, summary dashboards

`rollup("department", "year")`:

department	year	sum(sales)
Sales	2024	100
Sales	2025	150
Sales	null	250
HR	2024	200
HR	null	200
null	null	450

🔷 `cube()` in PySpark

- `cube()` performs **multi-dimensional aggregations**.
- It returns **all possible combinations** (Cartesian product) of the grouping columns, **including subtotals and grand total**.
- Think of it as:
  `groupBy()` ⊂ `rollup()` ⊂ `cube()`

🔍 Difference from `rollup()`:

Feature	`rollup("dept", "year")`	`cube("dept", "year")`
Combos	Hierarchical	All combinations
Missing Combos	Skipped	Included (e.g., year totals, dept totals)
Grand Total	✅ Yes	✅ Yes
Null Columns	For upper levels only	For all combinations

Output:

department	year	sum(sales)
HR	2024	200
HR	null	200
Sales	2024	100
Sales	2025	150
Sales	null	250
null	2024	300
null	2025	150
null	null	450

36. How do you perform pivot operations in PySpark?

```
pivot_df = df.groupBy("department").pivot("year").sum("sales")
pivot_df.show()
```

```
+----------+----+----+
|department|2024|2025|
+----------+----+----+
|        HR|200 |100 |
|     Sales|100 |150 |
|        IT|null|300 |
+----------+----+----+
```

PySpark doesn't have a built-in melt() function like Pandas, but you can **simulate unpivoting** using selectExpr + stack() if needed.

37. What is Spark Streaming, and how does it work?

Spark Streaming is a component of **Apache Spark** that enables **real-time processing** of data streams. It allows you to process live data from sources like Kafka, Flume, Kinesis, or sockets and perform transformations and actions just like batch jobs.

Spark Streaming works by:

1. **Dividing** the incoming data stream into small **batches** (micro-batching).
2. Each batch is processed using the **Spark engine** (like a mini Spark job).
3. The results are returned in near-real-time

Component	Role
Data Source	Kafka, Kinesis, TCP socket, etc.
Receiver	Collects data and stores it in Spark's memory
Batch Generator	Splits stream into small time-based batches (e.g., every 5 sec)
Driver Program	Runs the main Spark application and coordinates work
Executor	Performs transformations and actions on batches

```
# Read streaming data from socket
df = spark.readStream.format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
```

38. What is the difference between structured and unstructured streaming?

🔷 Structured Streaming

Structured Streaming is the modern approach introduced in **Spark 2.x** that integrates seamlessly with **DataFrames and Datasets**.

Key Characteristics:

- **DataFrame/Dataset API**: Uses the same high-level API as **batch processing** (DataFrames/Datasets) for stream processing, making it easy to use for developers familiar with Spark.
- **Stateful Operations**: Allows operations like aggregation, join, and windowing that maintain state over time.
- **End-to-End Exactly Once**: Supports exactly-once fault tolerance, which guarantees the data will be processed once and only once.
- **Micro-Batching**: Data is processed in small batches, but internally Spark manages the underlying complexity, allowing you to work with it as a continuous process.
- **Fault Tolerance**: Automatically handles fault tolerance via **checkpointing** and **write-ahead logs**.
- **Unified Batch and Streaming**: The same code is used for batch processing and streaming processing, making it more maintainable.

🔷 Unstructured Streaming (DStream)

Unstructured Streaming, also known as **DStream-based Streaming**, is the older streaming model based on **Resilient Distributed Datasets (RDDs)**.

Key Characteristics:

- **RDD API**: Works with the **RDD-based** API, where each stream is represented as a **DStream** (a sequence of RDDs).
- **Simpler**: The DStream API is more intuitive for beginners, as it follows the RDD transformation pattern (similar to batch processing).
- **Limited State Handling**: Handling complex stateful operations such as joins and aggregations across windows is more difficult than in Structured Streaming.
- **Fault Tolerance**: Offers fault tolerance using **checkpointing**, but does not have the built-in **exactly-once semantics** of Structured Streaming.
- **Not Optimized**: DStream-based processing is generally less optimized for performance compared to Structured Streaming due to Spark's use of RDDs, which do not provide as many optimizations as the DataFrame/Dataset API.

39.How do you handle late-arriving data in Spark Streaming?

Watermarking is a technique in **Structured Streaming** used to track the **progress** of event time and allow late-arriving data to be processed even after the expected time window has passed. When data is processed by **event time** (e.g., timestamps in the data), late data that arrives after the window has been processed is typically dropped. Watermarking allows Spark to **delay processing** by a specified duration to accommodate late data.

- **Watermark Definition**: You define a watermark to indicate how much "lateness" you are willing to accept for each event time.
- **How It Works**: Spark will retain state for windows of data that are within the watermark threshold, and it will process late data that falls within the allowed window.

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

# Sample DataFrame with timestamped data
df = spark.readStream.format("csv") \
    .option("header", "true") \
    .schema("timestamp TIMESTAMP, value STRING") \
    .load("path/to/input")

# Define watermark to allow late data up to 10 minutes after the window
df_with_watermark = df.withWatermark("timestamp", "10 minutes")

# Window the data by 1 minute intervals and calculate the count of events
df_windowed = df_with_watermark.groupBy(window(col("timestamp"), "1 minute")).count()

# Write the output to the console
query = df_windowed.writeStream.outputMode("append").format("console").start()

query.awaitTermination()
```

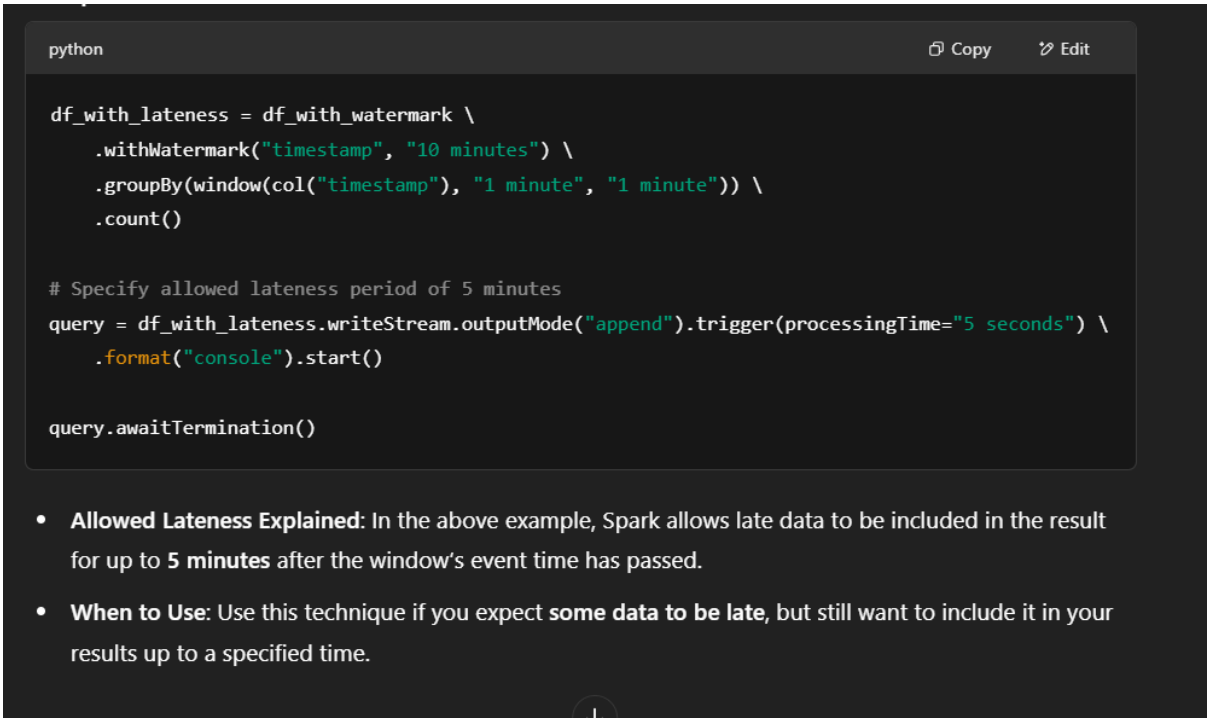
Watermark Explained: In the above example, withWatermark("timestamp", "10 minutes") tells Spark to allow late data to be processed for up to **10 minutes** after the window time.

When to Use: Use watermarking when you are working with **event-time** data (like logs or sensor data) and expect that some events may arrive out of order but you still want to process them correctly.

2 Allowed Lateness

In **Structured Streaming**, **allowed lateness** is the period during which late data can still be included in the result of a windowed operation. It works in conjunction with watermarks.

- **How It Works**: Allowed lateness ensures that the processing of late-arriving data is done even after the window has been closed.
- **Effect**: If you specify an allowed lateness period, Spark will allow late data to be added to a previously closed window until the allowed lateness period has passed.



40. Describe the role of the Catalyst optimizer in PySpark. How does it enhance query execution?

Catalyst is a **query optimization framework** used in **Spark SQL and Structured Streaming**. It takes user-written queries (in SQL or DataFrame API) and **optimizes them through multiple stages**, resulting in **high-performance execution plans**.

1. Logical Plan Optimization

- It starts with a **logical plan** (based on what the user writes using DataFrame or SQL).
- Catalyst applies **rule-based optimizations** like:
    - Constant folding
    - Boolean expression simplification
    - Predicate pushdown
    - Null propagation

2. Physical Plan Generation

- Converts the optimized logical plan into one or more **physical plans**.
- Evaluates **different strategies** (e.g., hash join, sort merge join) and selects the most efficient one using **cost-based optimization** (if enabled).

3. Code Generation (Tungsten Project)

- Uses **whole-stage code generation (WSCG)** to generate optimized **Java bytecode** for execution.

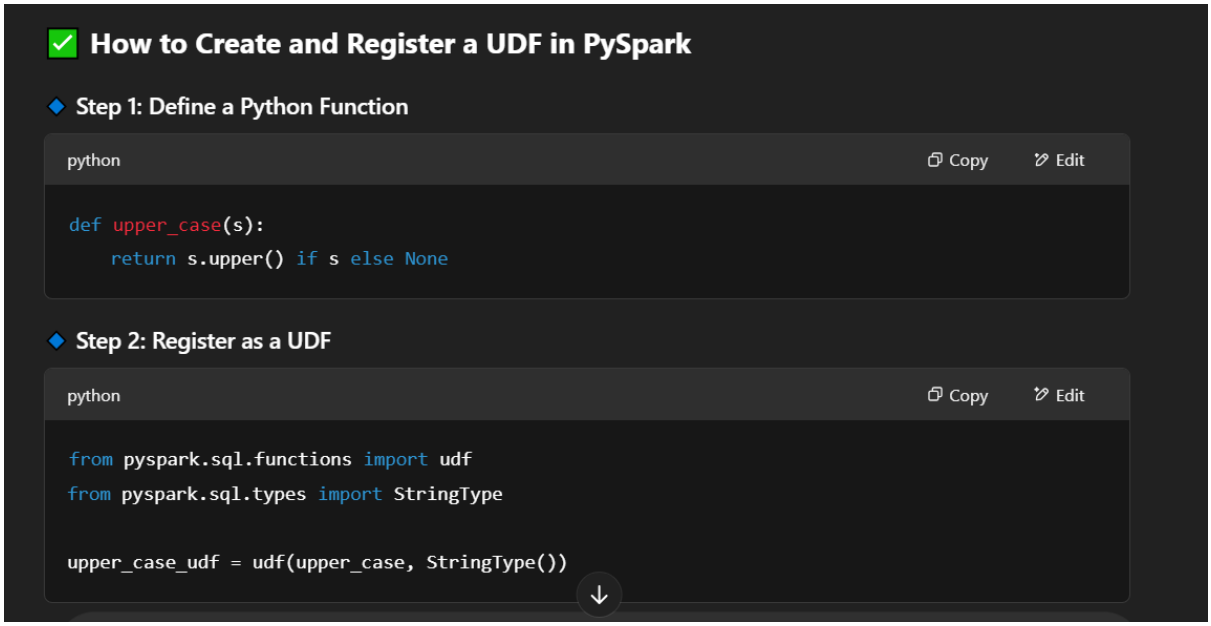
- Minimizes virtual function calls and object creation, improving CPU efficiency.

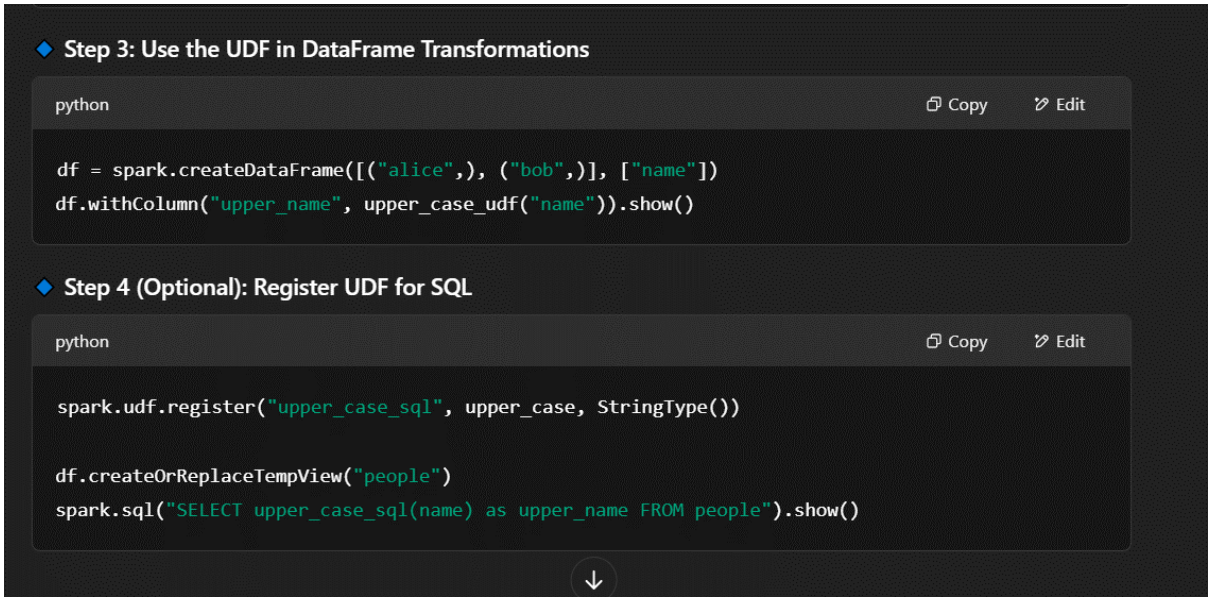
4. Extensibility

- Catalyst is highly modular and supports adding **custom rules, expressions, and strategies**, which helps advanced users fine-tune query behavior.

41. How do you create and register a user-defined function (UDF) in PySpark? What are the performance considerations?

Creating and registering a **User-Defined Function (UDF)** in PySpark allows you to apply custom Python logic to Spark DataFrames. While UDFs offer flexibility, they come with performance trade-offs, especially compared to Spark's native functions.





Consideration	Explanation
❗ Serialization Overhead	Data must be serialized between JVM (Spark engine) and Python (UDF), which is slow.
❗ No Catalyst Optimizations	UDFs are treated as a black box — Catalyst cannot optimize or push down filters.
⚡ Use Native Functions First	Always prefer Spark's built-in functions (`functions` module) — they are faster and optimized.
⚡ Use Pandas UDFs (Vectorized)	With Apache Arrow, they can run faster by working on batches of data.
⚠️ Testing and Debugging	Harder to debug than native transformations — should be thoroughly tested.

42. Discuss the concept of shuffling in PySpark. How does it affect performance, and how can it be minimized?

Shuffling is the process of **redistributing data** across different partitions or executors, often over the network. It happens when operations require **data from multiple partitions to be grouped or moved**, such as:

🔷 **Common operations that trigger shuffling:**

- groupBy()
- reduceByKey()
- join()
- distinct()
- repartition()
- cogroup()
- sort()

⚠️ Why is Shuffling Expensive?

Impact	Description
📦 Disk I/O	Intermediate data may be written to disk.
🌐 Network I/O	Data is sent across nodes over the network.
🧠 Memory usage	Requires memory for sorting and buffering.
⏱️ Performance overhead	Slows down jobs and increases GC pressure.

📉 How to Minimize Shuffling

✅ 1. Use `reduceByKey()` Instead of `groupByKey()`

python

```python
rdd.reduceByKey(lambda x, y: x + y)  # better
rdd.groupByKey().mapValues(sum)      # causes more shuffling
```

✅ 2. Broadcast Smaller DataFrames

Avoid shuffle-heavy joins:

python

```python
from pyspark.sql.functions import broadcast

df_large.join(broadcast(df_small), "key")  # avoids shuffle
```

✅ 3. Use Partition-Aware Joins

If both DataFrames are **partitioned by the join key**, Spark can avoid shuffling.

✅ 4. Repartition Wisely

Avoid excessive repartitioning.

python

```python
df.repartition(10, "key")   # triggers shuffle
df.coalesce(2)              # avoids shuffle if reducing partitions
```

✅ 5. Avoid Wide Transformations If Possible

Use **narrow transformations** like `map`, `filter`, `flatMap`, which **don't require shuffling**.

43. How do you read data from and write data to various file formats (e.g., CSV, Parquet, JSON) in PySpark?

✅ CSV

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

✅ Parquet

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

✅ JSON

```python
df_json = spark.read.option("multiline", "true").json("path/to/file.json")
```

```python
df.write.option("header", "true").csv("path/to/output/folder")
```

✅ Parquet

```python
df.write.parquet("path/to/output/folder")
```

✅ JSON

```python
df.write.json("path/to/output/folder")
```

🧠 Common Options for `.read` and `.write`

Option	Purpose
`header`	Treat first row as column names (CSV)
`inferSchema`	Automatically detect data types (CSV/JSON)
`multiline`	Read multiline JSON
`mode("overwrite")`	Overwrite existing data
`mode("append")`	Append to existing directory

44. Calculate the top 5 products with the highest sales in each region.

```
# Define window partitioned by region and ordered by sales descending
window_spec = Window.partitionBy("region").orderBy(col("sales").desc())

# Add row number to each record within the region
ranked_df = df.withColumn("rank", row_number().over(window_spec))

# Filter top 5 products per region
top5_df = ranked_df.filter(col("rank") <= 5).drop("rank")
```

45. Join two DataFrames based on a composite key and filter rows where the sales amount exceeds a threshold.

```
# Join on composite key (region, product)
df_joined = df_sales.join(df_info, on=["region", "product"], how="inner")

# Filter where sales > 1000
df_filtered = df_joined.filter(col("sales") > 1000)
```

46. Calculate the running total of sales for each customer, partitioned by customer ID and ordered by date.

```
# Define the window: partition by customer, order by date
window_spec = Window.partitionBy("customer_id").orderBy("date") \
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)

# Add a running total column
df_with_running_total = df.withColumn(
    "running_total",
    sum("sales").over(window_spec)
)
```

47. Implement a custom UDF to perform text sentiment analysis.

```
# Define the custom UDF for sentiment analysis
def get_sentiment(text):
    blob = TextBlob(text)
    # Return the polarity as sentiment score
    if blob.sentiment.polarity > 0:
        return 'positive'
    elif blob.sentiment.polarity < 0:
        return 'negative'
    else:
        return 'neutral'

# Register the UDF with Spark
sentiment_udf = udf(get_sentiment, StringType())

# Apply the UDF to the DataFrame
df_with_sentiment = df.withColumn("sentiment", sentiment_udf("review_text"))
```

48. Identify outliers in a dataset using standard deviation or IQR methods.

1. Standard Deviation Method

```
from pyspark.sql import functions as F

# Sample DataFrame
data = [(1, 10), (2, 12), (3, 14), (4, 100), (5, 15), (6, 16), (7, 18), (8, 50), (9, 20), (10, 22
columns = ["id", "value"]
df = spark.createDataFrame(data, columns)

# Calculate mean and standard deviation
mean_val = df.select(F.avg("value")).collect()[0][0]
stddev_val = df.select(F.stddev("value")).collect()[0][0]

# Calculate outliers: values more than 3 standard deviations away from the mean
outliers_df = df.filter(F.abs(df["value"] - mean_val) > 3 * stddev_val)

outliers_df.show()
```

2. IQR Method

python Copy Edit

```python
# Calculate Q1 (25th percentile) and Q3 (75th percentile)
Q1 = df.approxQuantile("value", [0.25], 0.01)[0]
Q3 = df.approxQuantile("value", [0.75], 0.01)[0]
IQR = Q3 - Q1

# Define bounds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out the outliers
outliers_iqr_df = df.filter((df["value"] < lower_bound) | (df["value"] > upper_bound))

outliers_iqr_df.show()
```

49. Perform data partitioning and bucketing for optimized query performance.

1. Data Partitioning in PySpark

Partitioning splits the data into different partitions based on a specified column. This can improve performance by enabling Spark to perform parallel processing on data that resides in different partitions. Partitioning is particularly useful when you often filter data by a specific column.

```python
# Partition the data by department_id
df_partitioned = df.repartition(3, "department_id")
```

2. Data Bucketing in PySpark

Bucketing is a technique used to divide data into a fixed number of buckets based on the hash of a column. Unlike partitioning, which is based on the values of the column, bucketing creates a fixed number of **equal-sized** buckets, regardless of the data distribution.

```python
# Bucketing by 'department_id' with 3 buckets
df.write.bucketBy(3, "department_id").saveAsTable("bucketed_data")

# Reading the bucketed data
bucketed_df = spark.table("bucketed_data")
bucketed_df.show()
```

Feature	Partitioning	Bucketing
Data Distribution	Divides data based on column values	Divides data into a fixed number of buckets based on a hash function
Use Case	Optimized for filter queries	Optimized for join queries on the same column
Flexibility	Flexible, can partition by any column	Fixed number of buckets, less flexibility
Storage	Data is physically split into multiple files	Data is stored in predefined buckets
Performance	Speeds up query performance when filtering by the partitioned column	Speeds up joins between bucketed tables

50. Use PySpark to merge multiple CSV files into a single DataFrame and remove duplicates.

```
# Step 2: Read multiple CSV files into one DataFrame
# You can use a wildcard pattern to read all matching files in a folder
csv_path = "/path/to/csv/folder/*.csv"
df = spark.read.option("header", "true").csv(csv_path)

# Step 3: Remove duplicates (by all columns)
df_deduped = df.dropDuplicates()
```

51. Convert a PySpark DataFrame to a Pandas DataFrame for advanced analytics.

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


# Step 3: Convert PySpark DataFrame to Pandas DataFrame
pandas_df = df.toPandas()
```

52. Write a PySpark script to stream data from Kafka and process it in real-time.

```

# Step 1: Start SparkSession with Kafka support
spark = SparkSession.builder \
    .appName("KafkaStreamingExample") \
    .getOrCreate()

spark.sparkContext.setLogLevel("WARN")  # Optional: suppress INFO logs

# Step 2: Read streaming data from Kafka topic
kafka_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \  # Replace with your Kafka broker
    .option("subscribe", "test-topic") \                    # Replace with your Kafka topic
    .option("startingOffsets", "latest") \
    .load()

# Step 3: Extract and process value field from Kafka (assumed to be UTF-8 string)
value_df = kafka_df.selectExpr("CAST(value AS STRING) as message")

# Optional: Simple transformation (e.g., filter messages containing "error")
filtered_df = value_df.filter(expr("message LIKE '%error%'"))

# Step 4: Output result to console
query = filtered_df.writeStream \
    .outputMode("append") \
    .format("console") \
    .option("truncate", "false") \
    .start()

query.awaitTermination()

```

53. How would you convert a PySpark DataFrame column with a JSON string to multiple columns

```python
# Create Spark session
spark = SparkSession.builder.appName("Parse JSON").getOrCreate()

# Sample data
data = [
    ('{"name": "Alice", "age": 30, "city": "NY"}',),
    ('{"name": "Bob", "age": 25, "city": "LA"}',)
]

# Create DataFrame with JSON string
df = spark.createDataFrame(data, ['json_str'])

# Define schema for the JSON
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("city", StringType(), True)
])

# Convert JSON string to multiple columns
df_parsed = df.withColumn("parsed", from_json(col("json_str"), schema)).select("parsed.*")

# Show the final DataFrame
df_parsed.show()
```

54. How can you remove duplicate rows in a DataFrame based on specific columns?

```python
df.dropDuplicates(["col1", "col2"])
```

55. how you would filter a DataFrame to include only rows where a column contains a specific substring.

```python
df.filter(df["column_name"].contains("substring"))
```

56. How do you add a constant column to a PySpark DataFrame?

```python
from pyspark.sql.functions import lit
df = df.withColumn("new_column", lit("constant_value"))
```

57. Question: How would you convert a DataFrame column from one data type to another?

```python
df = df.withColumn("column_name",
df["column_name"].cast("new_data_type"))
```

58. Explain how to use a PySpark SQL query on a DataFrame.

```
df.createOrReplaceTempView("temp_view")
spark.sql("SELECT * FROM temp_view WHERE column > 10")
```

59. How can you rename multiple columns in a PySpark DataFrame?

```
new_column_names = ["new_name1", "new_name2"]
df = df.toDF(*new_column_names)
```

60. Describe how you would pivot a DataFrame in PySpark

```
df.groupBy("pivot_column").pivot("category_column").agg({
"value_column": "sum"})
```

61. How can you calculate the distinct count of values in a column

```
df.select("column").distinct().count()
```

62. Explain the purpose of the groupBy() and agg() function

```
df.groupBy("group_column").agg({"value_column": "sum"})
```

63. How do you handle null values in a DataFrame column?

```
df.fillna({"column_name": "default_value"})
```

64. How would you read a CSV file into a PySpark DataFrame with a header and custom delimiter?

```
df = spark.read.option("header",
"true").option("delimiter", ",").csv("path/to/file.csv")
```

65 . How can you sort a DataFrame by multiple columns in ascending and descending order?

```
df.orderBy(df["col1"].asc(), df["col2"].desc())
```

66. Describe how you would join two DataFrames on multiple keys.

```
df1.join(df2, (df1["col1"] == df2["col1"]) & (df1["col2"]
== df2["col2"]), "inner")
```

67. How do you calculate the rank of rows within each partition of a DataFrame?

```
from pyspark.sql.window import Window
from pyspark.sql.functions import rank

window_spec =
Window.partitionBy("partition_column").orderBy("order_col
umn")
df = df.withColumn("rank", rank().over(window_spec))
```

68. How do you perform an inner join and filter out rows with nulls in one of the columns after the join?

```
df = df1.join(df2, "join_column",
"inner").filter(df2["column"].isNotNull())
```

69. How can you change the number of partitions in a DataFrame to improve performance?

- **Answer:** Use `repartition()` to increase partitions or `coalesce()` to reduce them:

```
df = df.repartition(10)   # Increases partitions
df = df.coalesce(5)       # Reduces partitions
```

70. How would you aggregate data using both sum and average functions in a single query?

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

df.groupBy("group_column").agg(sum("value_column"),
avg("value_column"))
```

71. Explain how to concatenate two string columns with a separator.\

- **Answer:** Use `concat_ws()` to join with a separator:

```
from pyspark.sql.functions import concat_ws

df = df.withColumn("full_name", concat_ws(" ", df["first_name"], df["last_name"]))
```