Pyspark Notes:

1.What is an RDD in Apache Spark? Explain its characteristics.
RDD (Resilient Distributed Dataset) is the fundamental data structure in Apache Spark. It is an immutable, distributed collection of objects that can be processed in parallel.
Characteristics of RDDs:

1. **Immutable:** Once created, RDDs cannot be modified but can be transformed into new RDDs.
2. **Distributed:** Data is split across multiple nodes in a cluster.
3. **Fault-tolerant:** Automatically rebuilds data in case of partition failure using lineage information.
4. **Lazy Evaluation:** Operations are not executed immediately but only when an action (e.g., .collect()) is called.
5. **In-Memory Processing:** Supports in-memory computation for faster processing.
6. **Partitioned:** Data is divided into partitions to enable parallelism.
7. **Transformation and Actions:** Two types of operations—transformations (e.g., .map(), .filter()) create new RDDs, while actions (e.g., .collect(), .count()) compute results.

2.What is lineage transformations in spark ?
Lineage Transformation in Spark refers to the logical sequence of operations applied on an RDD to derive a new RDD. It tracks the series of transformations (like .map(), .filter(), .reduceByKey()) that were applied to build an RDD from its parent RDDs.

Key Points:

1. **Tracks Dependencies:** Spark maintains a Directed Acyclic Graph (DAG) of transformations, enabling it to trace the lineage of an RDD.
2. **Fault Tolerance:** If a partition is lost, Spark can recompute it using the lineage information without reprocessing the entire dataset.
3. **No Immediate Execution:** Transformations are evaluated lazily and only executed when an action (like .count() or .collect()) triggers computation.

3.How RDDs are fault-tolerant ?
In Spark, **fault tolerance** is achieved using RDD **lineage**. If a part of an RDD (a partition) is lost due to node failure, Spark uses the lineage information (the sequence of transformations) to recompute only the lost partition, not the entire dataset .

```
# Step 1: Create an RDD
data = sc.parallelize(range(1, 11), 3)                    # Data: [1, 2, ..., 10] split into 3 partitions

# Step 2: Apply transformations
mapped_rdd = data.map(lambda x: (x, x ** 2))              # Creates pairs: [(1, 1), (2, 4), ...]
filtered_rdd = mapped_rdd.filter(lambda x: x[1] % 2 == 0) # Keeps only even squares
reduced_rdd = filtered_rdd.reduceByKey(lambda a, b: a + b) # Combines by keys (if any)

# Step 3: Trigger an action
result = reduced_rdd.collect()
print(result)

# Example output: [(2, 4), (4, 16), ...]
```

Fault Tolerance in Action:

1. **Scenario:** A node handling one partition of mapped_rdd fails after the map transformation but before filter or reduceByKey.
2. **Recovery:** Spark uses lineage to:
   - **Recreate the lost partition:** Re-execute the map transformation on the corresponding partition of data.
   - **Apply downstream transformations:** Apply filter and reduceByKey only to the recovered data.
3. **Efficiency:** Spark doesn't recompute the entire dataset—just the lost partition, saving time and resources.

How Lineage Works:
Lineage for reduced_rdd:

1. Start with data.
2. Apply map → filter → reduceByKey.
3. Re-execute only the necessary steps for the lost partition to recover it.

This shows how Spark's fault tolerance ensures reliability in complex workflows.

4. **RDD transformations**

1.Create a RDD

```
data = [
   (1, "Alice", 30, "New York", 70000),
   (2, "Bob", 35, "Los Angeles", 80000),
   (3, "Charlie", 40, "Chicago", 75000),
   (4, "David", 28, "New York", 50000),
   (5, "Eve", 32, "San Francisco", 60000)
]

# Create an RDD with 2 partitions
rdd = sc.parallelize(data, 2)
```

2. Partion

```
# Check the number of partitions
```

```
print(rdd.getNumPartitions())  # Output: 2

# Repartition the data
repartitioned_rdd = rdd.repartition(3)
print(repartitioned_rdd.getNumPartitions())  # Output: 3
```

3: Map
Apply a transformation to compute the **annual salary** for each person.

```
# Compute annual salary (monthly salary * 12)
mapped_rdd = rdd.map(lambda x: (x[1], x[4] * 12))  # (Name, Annual Salary)
print(mapped_rdd.collect())
# Output: [('Alice', 840000), ('Bob', 960000), ('Charlie', 900000), ('David', 600000), ('Eve', 720000)]
```

4: Filter
Filter out employees whose **salary is less than 70,000**.

```
filtered_rdd = rdd.filter(lambda x: x[4] >= 70000)
print(filtered_rdd.collect())
# Output: [(1, 'Alice', 30, 'New York', 70000), (2, 'Bob', 35, 'Los Angeles', 80000), (3, 'Charlie', 40, 'Chicago', 75000)]
```

5: Reduce
Find the **maximum salary** among all employees.

```
max_salary = rdd.map(lambda x: x[4]).reduce(lambda a, b: max(a, b))
print(max_salary)  # Output: 80000
```

reduceByKey in PySpark

reduceByKey is a **transformation** in PySpark used to aggregate data for each key in a key-value RDD.
It applies a specified binary function to combine the values of each key.

How it Works

1. **Within Partition:** Combines values with the same key locally (in the same partition).
2. **Across Partitions:** Combines the results from different partitions for the same key.

This two-step process makes reduceByKey more efficient than groupByKey because it reduces data shuffling between nodes.

1: Summing Values for Each Key

```
# RDD with key-value pairs
rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3), ("b", 4), ("c", 5)])

# Apply reduceByKey to sum values for each key
result = rdd.reduceByKey(lambda x, y: x + y).collect()
print(result)
# Output: [('a', 4), ('b', 6), ('c', 5)]
```

2.Average Salary for each department

```
# Sample RDD with (id, name, department, salary)
rdd = sc.parallelize([
   (1, "Alice", "HR", 5000),
   (2, "Bob", "IT", 7000),
   (3, "Charlie", "HR", 6000),
   (4, "David", "IT", 8000),
   (5, "Eve", "Finance", 7500)
])

# Step 1: Map the values to (department, (salary, 1)) where 1 is the count
rdd_mapped = rdd.map(lambda x: (x[2], (x[3], 1)))  # (department, (salary, count))

[('HR', (5000, 1)), ('IT', (7000, 1)), ('HR', (6000, 1)), ('IT', (8000, 1)), ('Finance', (7500, 1))]

# Step 2: Use reduceByKey to aggregate the salary and count for each department
aggregated_rdd = rdd_mapped.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))  # (sum of salary, total count)

[('HR', (11000, 2)), ('IT', (15000, 2)), ('Finance', (7500, 1))]

# Step 3: Calculate the average salary for each department
avg_salary_rdd = aggregated_rdd.mapValues(lambda x: x[0] / x[1])  # Calculate average salary

[('HR', 5500.0), ('IT', 7500.0), ('Finance', 7500.0)]

# Collect the results
result = avg_salary_rdd.collect()

# Print the result
for dept, avg_salary in result:
   print(f"Department: {dept}, Average Salary: {avg_salary}")

Department: HR, Average Salary: 5500.0
Department: IT, Average Salary: 7500.0
Department: Finance, Average Salary: 7500.0
```

Create a rdd with key value pair and then apply reduce by key .

- **reduce**: An **action** that aggregates the entire RDD into a single result.
- **reduceByKey**: A **transformation** that aggregates values by key and returns a new RDD.

6: Aggregate
Find the **total salary** and the **average age** of all employees.

```
result = rdd.aggregate(
                        (0, 0, 0),                                                    # Initial accumulator: (sum of salaries, sum of ages, count)
                        lambda acc, x: (acc[0] + x[4], acc[1] + x[2], acc[2] + 1),    # SeqOp lambda
                        acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1], acc1[2] + acc2[2])   # CombOp
            )

total_salary = result[0]                # 335000
 average_age = result[1] / result[2]    # 165 / 5 = 33.0
```

aggregateByKey : : Aggregates values **by key** in an RDD of key-value pairs, producing a result for each key.

```
rdd = sc.parallelize([
   ("HR", 5000),
   ("IT", 6000),
   ("HR", 7000),
   ("IT", 8000),
   ("Finance", 7500)
])

result = rdd.aggregateByKey(
   (0, 0)  # Initial value: (sum of salaries, count)
)(
   lambda acc, salary: (acc[0] + salary, acc[1] + 1),  # seqOp: Sum salaries and increment count
   lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])  # combOp: Combine partial results
)

averages = result.mapValues(lambda x: x[0] / x[1]).collect()
print(averages)
```

aggregateByKey

Definition:
aggregateByKey is a transformation in PySpark used specifically for **pair RDDs** (key-value RDDs).
It is a powerful operation that allows custom aggregation for values associated with the same key, both **within partitions** and **across partitions**.
aggregateByKey operates in two stages:

1. **Within-Partition Aggregation**:
   - It uses a **sequential function** to combine values of the same key within each partition.
2. **Across-Partition Aggregation**:
   - It uses a **combiner function** to merge results for the same key across different partitions.

rdd.aggregateByKey(zeroValue, seqFunc, combFunc)

- **zeroValue**: The initial value for aggregation (e.g., (0, 0) for sum and count).
- **seqFunc**: Function for aggregating values within a partition.
- **combFunc**: Function for aggregating results across partitions.

When to Use aggregateByKey?

- When you need **complex aggregations** involving multiple metrics for each key.
- When aggregation logic varies between **within partitions** and **across partitions**.

7.GroupBy
Output -> [
("key" , [( ) , ( )])
]
Tuples containing key and list of tuples

```
grouped_rdd = rdd.groupBy(lambda x: x[2])  # Group by department (x[2])

# Collect the result and print it
result = grouped_rdd.collect()
print(result)

[   ( 'HR', [(1, 'Alice', 'HR', 5000), (3, 'Charlie', 'HR', 6000)]) ,
     ('IT', [(2, 'Bob', 'IT', 7000), (4, 'David', 'IT', 8000)]),
      ('Finance', [(5, 'Eve', 'Finance', 7500)])]
```

groupByKey
If your RDD is already in the form of key-value pairs (e.g., (key, value)), you can use groupByKey instead of groupBy

```
rdd = sc.parallelize([("HR", 5000), ("IT", 7000), ("HR", 6000), ("IT", 8000)])
grouped_rdd = rdd.groupByKey()
result = grouped_rdd.collect()
print(result)

[('HR', [5000, 6000]), ('IT', [7000, 8000])]
```

Map vs FlatMap

Feature	map	flatMap
Transformation	One-to-one mapping (1 input → 1 output).	One-to-many mapping (1 input → 0, 1, or many outputs).
Output Structure	Preserves the structure (nested lists stay nested).	Flattens the output structure (nested lists are unwrapped).
Use Case	Simple transformations (e.g., doubling numbers).	Complex transformations (e.g., splitting strings into words).

Map :

```
rdd = sc.parallelize(["hello world", "spark flatmap"])
mapped = rdd.map(lambda x: x.split(" "))
print(mapped.collect())
# Output: [['hello', 'world'], ['spark', 'flatmap']]

*
rdd = sc.parallelize([(1, 2), (3, 4)])
mapped = rdd.map(lambda x: tuple([i**2 for i in x]))
mapped.collect()
# Output: [[1, 4], [9, 16]]
```

FlatMap :

```
flat_mapped = rdd.flatMap(lambda x: x.split(" "))
print(flat_mapped.collect())
# Output: ['hello', 'world', 'spark', 'flatmap']

*
rdd = sc.parallelize([(1, 2), (3, 4)])
mapped = rdd.flatMap(lambda x: tuple([i**2 for i in x]))
mapped.collect()
# Output: [[1, 4,9, 16]]
```

map

- **Applicable to**: Any RDD.
- **Transformation**: Applies the provided function to each element of the RDD, regardless of its structure.
- **Output**: Transforms every element of the RDD (both keys and values, if it's a key-value pair RDD).

mapValues

- **Applicable to**: Key-Value Pair (Pair RDDs) only.
- **Transformation**: Applies the provided function only to the **values** of the key-value pairs, keeping the keys unchanged.
- **Output**: Modifies only the values while retaining the original keys.

Map :

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("c", 3)])

# Apply map to increment the value by 1 and concatenate "key-" to the key
result = rdd.map(lambda x: ("key-" + x[0], x[1] + 1)).collect()

print(result)
# Output: [('key-a', 2), ('key-b', 3), ('key-c', 4)]
```

mapValues :

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("c", 3)])

# Apply mapValues to increment the value by 1
result = rdd.mapValues(lambda x: x + 1).collect()

print(result)
# Output: [('a', 2), ('b', 3), ('c', 4)]
```

Use Case Comparison:

If you need to:

1. **Modify both keys and values** → Use map.
2. **Modify only the values** in a key-value RDD → Use mapValues.

RDD Actions in PySpark

RDD actions trigger computations and return results to the driver program or save data to an external storage system. Unlike transformations, actions are **eager** and result in the execution of the DAG (Directed Acyclic Graph) of operations built by transformations.
Here are the key **RDD actions** in PySpark:

1. collect()

- **Description**: Retrieves all elements of the RDD to the driver as a list.
- **Use Case**: Useful for small RDDs; avoid for large datasets due to memory limitations.
- **Example**:
  rdd = sc.parallelize([1, 2, 3, 4])
  result = rdd.collect()
  print(result) # Output: [1, 2, 3, 4]

2. count()

- **Description**: Returns the number of elements in the RDD.

- **Use Case**: Quick way to get the size of an RDD.
- **Example**:
  rdd = sc.parallelize([1, 2, 3, 4])
  count = rdd.count()
  print(count) # Output: 4

3. first()

- **Description**: Returns the first element of the RDD.
- **Use Case**: Useful to inspect the starting element.
- **Example**:

  rdd = sc.parallelize([1, 2, 3, 4])
  first_element = rdd.first()
  print(first_element) # Output: 1

4. take(n)

- **Description**: Returns the first n elements of the RDD.
- **Use Case**: Useful for previewing data without collecting the entire RDD.
- **Example**:
  rdd = sc.parallelize([1, 2, 3, 4])
  result = rdd.take(2)
  print(result) # Output: [1, 2]

5. takeOrdered(n, key)

- **Description**: Returns the first n elements of the RDD, ordered by a specified key.
- **Use Case**: Retrieve the smallest or largest elements.
- **Example**:

  rdd = sc.parallelize([3, 1, 2, 4])
  result = rdd.takeOrdered(2)
  print(result) # Output: [1, 2]

6. top(n,key)

- **Description**: Returns the top n elements of the RDD in descending order.
- **Use Case**: Retrieve the largest elements.
- **Example**:

  rdd = sc.parallelize([3, 1, 2, 4])
  result = rdd.top(2)
  print(result) # Output: [4, 3]

7. reduce(func)

- **Description**: Aggregates elements of the RDD using a specified associative function.
- **Use Case**: Summing or combining values.
- **Example**:

  rdd = sc.parallelize([1, 2, 3, 4])
  result = rdd.reduce(lambda a, b: a + b)
  print(result) # Output: 10

reduce is an action because it:

- Triggers execution of the RDD lineage.
- Produces a single aggregated result and sends it to the driver.
- Does not return a new RDD for further transformations.

8. fold(zeroValue, func)

- **Description**: Aggregates elements using a zero value and an associative function. Works like reduce but with a default value.
- **Use Case**: Combine values with an initial accumulator.
- **Example**:

  rdd = sc.parallelize([1, 2, 3, 4])
  result = rdd.fold(0, lambda a, b: a + b)
  print(result) # Output: 10

9. aggregate(zeroValue, seqOp, combOp)

- **Description**: Performs aggregation with separate operations for within-partition (seqOp) and cross-partition (combOp).
- **Use Case**: Advanced aggregations like average or custom metrics.
- **Example**:

  rdd = sc.parallelize([1, 2, 3, 4])
  result = rdd.aggregate(
  (0, 0),
  lambda acc, x: (acc[0] + x, acc[1] + 1),
  lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])
)
  average = result[0] / result[1]
  print(average) # Output: 2.5

10. countByKey()

- **Description**: Counts the number of occurrences of each key in a paired RDD.
- **Use Case**: Quickly count items grouped by keys.
- **Example**:

```
rdd = sc.parallelize([('a', 1), ('b', 1), ('a', 1)])
counts = rdd.countByKey()
print(counts)  # Output: {'a': 2, 'b': 1}
```

11. foreach(func)

- **Description**: Applies a function to each element of the RDD.
- **Use Case**: Execute actions like saving or printing data.
- **Example**:

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.foreach(lambda x: print(x))  # Output: Prints 1, 2, 3, 4
```

12. saveAsTextFile(path)

- **Description**: Saves the RDD as a text file to the specified path.
- **Use Case**: Persist data to storage.
- **Example**:

```
rdd = sc.parallelize(["Spark", "Hadoop", "Python"])
rdd.saveAsTextFile("/path/to/save")
```

13. saveAsSequenceFile(path)

- **Description**: Saves the RDD as a Hadoop sequence file (key-value pairs).
- **Use Case**: Persist data in sequence format.

14. saveAsObjectFile(path)

- **Description**: Saves the RDD as serialized Java objects.
- **Use Case**: Efficient storage of RDDs for future use.

15. isEmpty()

- **Description**: Checks if the RDD is empty.
- **Example**:

```
rdd = sc.parallelize([])
print(rdd.isEmpty())  # Output: True
```

Read a txt file

```
rdd = sc.textFile("path/to/your/file.txt")

split_rdd = rdd.map(lambda line: line.split(",")) print(split_rdd.collect())
# Output: [['1', ' Alice', ' HR', ' 50000'], ['2', ' Bob', ' IT', ' 60000'], ...]
```

- **Custom Parsing**: For more complex delimiters (e.g., |, ;), replace line.split(",") with line.split(";").
- **Structured Data**: If your file is large and structured (e.g., CSV), consider using **DataFrames** for better performance:

```
df = spark.read.csv("path/to/data.txt", header=False, inferSchema=True)
df.show()
```

- **Optimization**: Always use the appropriate number of partitions for distributed processing, especially for large files. You can set the number of partitions while loading the file:

```
rdd = sc.textFile("path/to/data.txt", minPartitions=4)
```

. Analyzing Product Sales Data

Scenario: Calculate the total sales per product and identify the top 3 products by revenue.
Input Data (sales.txt):

plaintext
Copy code
1, Laptop, Electronics, 2, 700
2, Mouse, Electronics, 5, 25
3, Chair, Furniture, 3, 150
4, Desk, Furniture, 1, 300
5, Monitor, Electronics, 4, 200
Code:

python
Copy code
```
# Step 1: Load Data
rdd = sc.textFile("sales.txt")
# Step 2: Parse Data
parsed_rdd = rdd.map(lambda line: line.split(",")).map(lambda x: (x[1], int(x[3]) * int(x[4])))
# Step 3: Calculate Total Revenue Per Product
revenue_rdd = parsed_rdd.reduceByKey(lambda a, b: a + b)
# Step 4: Find Top 3 Products by Revenue
top_products = revenue_rdd.takeOrdered(3, key=lambda x: -x[1])
# Output
print(top_products)
# Example Output: [('Laptop', 1400), ('Monitor', 800), ('Chair', 450)]
```

2. Employee Data Analysis

Scenario: Compute the average salary and the highest salary for each department.
Input Data (employees.txt):

plaintext
Copy code
1, Alice, HR, 70000
2, Bob, IT, 80000
3, Charlie, Finance, 75000
4, David, HR, 60000
5, Eve, IT, 90000
Code:

```
# Step 1: Load Data
rdd = sc.textFile("employees.txt")
# Step 2: Parse Data
parsed_rdd = rdd.map(lambda line: line.split(",")).map(lambda x: (x[2], (int(x[3]), 1)))
# Step 3: Calculate Total Salary and Count for Each Department
dept_salary_count = parsed_rdd.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
# Step 4: Calculate Average Salary
avg_salary_rdd = dept_salary_count.mapValues(lambda x: x[0] / x[1])
# Step 5: Find Maximum Salary in Each Department
max_salary_rdd = parsed_rdd.map(lambda x: (x[0], x[1][0])).reduceByKey(lambda a, b: max(a, b))
# Output
print("Average Salary:", avg_salary_rdd.collect())
print("Max Salary:", max_salary_rdd.collect())
# Example Output:
# Average Salary: [('HR', 65000.0), ('IT', 85000.0), ('Finance', 75000.0)]
# Max Salary: [('HR', 70000), ('IT', 90000), ('Finance', 75000)]
```

3. Customer Purchase Patterns

Scenario: Identify frequent buyers who purchase more than 3 times and calculate their total spending.
Input Data (purchases.txt):

plaintext
Copy code
1, Alice, 300
2, Bob, 150
1, Alice, 500
3, Charlie, 700
2, Bob, 400
1, Alice, 200
Code:

python
Copy code
```
# Step 1: Load Data
rdd = sc.textFile("purchases.txt")
# Step 2: Parse Data
parsed_rdd = rdd.map(lambda line: line.split(",")).map(lambda x: (x[1], int(x[2])))
# Step 3: Aggregate Purchase Data
purchase_rdd = parsed_rdd.aggregateByKey((0, 0),
                                         lambda acc, value: (acc[0] + value, acc[1] + 1),
                                         lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))
# Step 4: Filter Frequent Buyers
```

```
frequent_buyers_rdd = purchase_rdd.filter(lambda x: x[1][1] > 3)
# Output
print(frequent_buyers_rdd.collect())
# Example Output: [('Alice', (1000, 3))]
```

4. Log Analysis

Scenario: Count the number of HTTP response codes (e.g., 200, 404) in a web server log file.
Input Data (web_logs.txt):

```
192.168.0.1 - - [27/Oct/2024] "GET /index.html HTTP/1.1" 200
192.168.0.2 - - [27/Oct/2024] "POST /form.html HTTP/1.1" 404
192.168.0.3 - - [27/Oct/2024] "GET /about.html HTTP/1.1" 200
192.168.0.4 - - [27/Oct/2024] "GET /contact.html HTTP/1.1" 404
192.168.0.1 - - [27/Oct/2024] "POST /index.html HTTP/1.1" 500
```

Code:

```
# Step 1: Load Data
rdd = sc.textFile("web_logs.txt")
# Step 2: Extract Response Codes
response_rdd = rdd.map(lambda line: line.split(" ")[-1])
# Step 3: Count Response Codes
response_count_rdd = response_rdd.map(lambda code: (code, 1)).reduceByKey(lambda a, b: a + b)
# Output
print(response_count_rdd.collect())
# Example Output: [('200', 2), ('404', 2), ('500', 1)]
```

Find the Top 3 Most Frequent Words in a Text File

In this example, we use flatMap, map, and reduceByKey to find the top 3 most frequent words in a large text file.

```
# Step 1: Load Data
rdd = sc.textFile("text_file.txt")
# Step 2: Split the text into words and map them to (word, 1)
words_rdd = rdd.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1))
# Step 3: Use reduceByKey to count word frequencies
word_counts_rdd = words_rdd.reduceByKey(lambda x, y: x + y)
# Step 4: Take top 3 most frequent words
top_3_words = word_counts_rdd.takeOrdered(3, key=lambda x: -x[1])
print(top_3_words)
```

Limitations of RDD over dataframes/datasets

1. Lack of Optimization

- **RDDs**: Do not benefit from Spark's Catalyst optimizer and Tungsten execution engine. Operations on RDDs are not optimized automatically, which can result in slower performance.
- **DataFrames/Datasets**: Use Catalyst for query optimization and Tungsten for efficient execution, resulting in better performance.

2. No Schema

- **RDDs**: RDDs are a collection of objects without a schema. This makes it harder to manage and query structured data.
- **DataFrames/Datasets**: Have a schema, making them more suitable for structured data processing and enabling the use of SQL queries.

3. Ease of Use

- **RDDs**: Require verbose and low-level coding, as transformations and actions are performed using functional programming constructs.
- **DataFrames/Datasets**: Offer higher-level APIs and are easier to use, especially for SQL-style operations.

4. Performance

- **RDDs**: Are slower due to the lack of optimization and serialization overhead.
- **DataFrames/Datasets**: Are faster because they use optimized physical execution plans and binary serialization.

5. Memory Usage

- **RDDs**: Use Java serialization, which can result in higher memory consumption and slower serialization/deserialization.
- **DataFrames/Datasets**: Use Tungsten's off-heap memory management, reducing memory usage and improving performance.

6. Data Type Safety

- **RDDs**: Are not type-safe. You can accidentally process data of incorrect types without knowing it until runtime.
- **Datasets**: Provide type safety at compile time, reducing runtime errors.

7. APIs and SQL Integration

- **RDDs**: Do not integrate directly with SQL.
- **DataFrames/Datasets**: Support SQL queries natively, allowing seamless integration with Spark SQL.

8. Error Handling

- **RDDs**: Debugging and error handling can be challenging because of the lack of a structured schema and higher verbosity.
- **DataFrames/Datasets**: Easier to debug due to schema enforcement and descriptive error messages.

Conclusion:

Use **RDDs** when:

- You need low-level transformations or actions that DataFrames/Datasets don't support.
- Working with unstructured or semi-structured data.
- **Unstructured raw files : Server logs , IOT data .**

Prefer **DataFrames/Datasets** when:

- You need performance and ease of use.
- Working with structured data and need SQL query capabilities.
- **Structured data , Schema based for analytics**

What is a DataFrame in PySpark?

A **DataFrame** in PySpark is a distributed collection of data organized into named columns, similar to a table in a relational database or a spreadsheet. It is one of the primary abstractions in Spark, introduced for handling structured and semi-structured data efficiently.

Key Features of DataFrame:

1. **Schema**: DataFrames have a defined schema, meaning each column has a name and a type.
2. **Optimizations**: DataFrames leverage Catalyst Optimizer and Tungsten Execution Engine for query optimization and improved performance.
3. **Ease of Use**: DataFrames support high-level APIs for operations like filtering, grouping, and aggregation, and can be queried using SQL.
4. **Interoperability**: Can be created from various data sources like JSON, CSV, Parquet, ORC, or even from an RDD.

Difference Between DataFrame and RDD

Feature	RDD	DataFrame
Abstraction Level	Low-level (raw distributed data)	High-level (structured data with schema)
Schema	No schema; works on key-value pairs or objects	Schema with named columns
Ease of Use	Requires manual coding for transformations	SQL-like operations; simpler API
Performance	No optimization (functional programming only)	Catalyst Optimizer for query execution
Data Sources	Requires custom parsers	Supports multiple formats (CSV, JSON, Parquet, etc.)
Error Handling	Errors may occur during execution	Errors are caught at compile time (schema validation)
Operations	Supports functional transformations (map, filter)	Supports SQL-like operations and aggregations
Serialization	Java serialization (less efficient)	Optimized serialization (Tungsten)
Use Case	Unstructured or custom data processing	Structured and semi-structured data processing

Example Comparison:

RDD Example:

Find the average salary by department.

python
Copy code

```
rdd = sc.textFile("employees.txt")
rdd = rdd.map(lambda line: line.split(",")).map(lambda x: (x[2], int(x[3])))
result = rdd.reduceByKey(lambda a, b: a + b).mapValues(lambda x: x / len(rdd)).collect()
print(result)
```

DataFrame Example:

Find the average salary by department.

python
Copy code

```
df = spark.read.csv("employees.csv", header=True, inferSchema=True)
df.groupBy("department").avg("salary").show()
```

Key Differences:

- In RDD, we need to manually parse and calculate the average.
- In DataFrame, operations are simpler and benefit from optimizations.

2. How can you create a DataFrame in PySpark?

- From a collection (list, dict)
- By reading a CSV, JSON, or Parquet file
- From an RDD
- From a Pandas DataFrame

From a Collection (List of Tuples or Dictionaries)

Example: List of Tuples

```
data = [("Alice", 25), ("Bob", 30), ("Charlie", 35)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)
df.show()
```

Example: List of Dictionaries

```
data = [{"Name": "Alice", "Age": 25}, {"Name": "Bob", "Age": 30}]
df = spark.createDataFrame(data)
df.show()
```

From an RDD

Example: Using Implicit Schema

```
rdd = spark.sparkContext.parallelize([("Alice", 25), ("Bob", 30)])
df = rdd.toDF(["Name", "Age"])
df.show()
```

Example: Using Explicit Schema

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
rdd = spark.sparkContext.parallelize([("Alice", 25), ("Bob", 30)])
schema = StructType([
   StructField("Name", StringType(), True),
   StructField("Age", IntegerType(), True)
])
df = spark.createDataFrame(rdd, schema)
df.show()
```

By Reading Files

Example: CSV File

```
df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
df.show()
```

Example: JSON File

```
df = spark.read.json("path/to/file.json")
df.show()
```

Example: Parquet File

```
df = spark.read.parquet("path/to/file.parquet")
df.show()
```

From Pandas DataFrame

```
import pandas as pd
pdf = pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [25, 30]})
df = spark.createDataFrame(pdf)
df.show()
```

By Running SQL Queries

```
data = [("Alice", 25), ("Bob", 30)]
df = spark.createDataFrame(data, ["Name", "Age"])
df.createOrReplaceTempView("people")
result_df = spark.sql("SELECT * FROM people WHERE Age > 25")
result_df.show()
```

Handling Bad Records While Loading Data

Example: Using spark.read with a CSV File

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("HandleBadRecords").getOrCreate()
# Load the data and set mode to 'PERMISSIVE' or 'DROPMALFORMED' to capture bad records
df = spark.read.format("csv") \
   .option("header", "true") \
   .option("mode", "PERMISSIVE") \  # Allows malformed records to be captured
   .option("columnNameOfCorruptRecord", "_corrupt_record") \  # Add a column for bad records
   .load("data.csv")

# Filter out bad records
bad_records = df.filter(df["_corrupt_record"].isNotNull())

# Show bad records
bad_records.show(truncate=False)
```

Handling Bad Records During Transformations

```
data = [
   "1,Alice,25",
   "2,Bob,thirty",  # Bad record: Age is not a number
   "3,Charlie,40"
]
```

```
rdd = spark.sparkContext.parallelize(data)

# Parse the records and handle bad ones
def parse_record(record):
    try:
        fields = record.split(",")
        return (int(fields[0]), fields[1], int(fields[2]))  # Convert age to int
    except Exception as e:
        return ("Bad Record", record)

parsed_rdd = rdd.map(parse_record)

# Separate good and bad records
good_records = parsed_rdd.filter(lambda x: x[0] != "Bad Record")
bad_records = parsed_rdd.filter(lambda x: x[0] == "Bad Record")

# Show bad records
print("Bad Records:")
for record in bad_records.collect():
    print(record)

# Output:
# Bad Records:
# ('Bad Record', '2,Bob,thirty')
```

Print vs show vs collect ?

Print(df) ->
DataFrame[Name: string, Age: bigint]

df.collect() ->
[Row(Name='Alice', Age=30), Row(Name='Bob', Age=25), Row(Name='Charlie', Age=35)]

df.show() ->
```
+-------+---+
| Name|Age|
 +-------+---+
| Alice| 30|
| Bob| 25|
|Charlie| 35|
+-------+---+
```

Differences Between union() and unionByName()

Feature	union	unionByName
Column Order	Must match	Matches by name
Missing Columns	Not allowed	Can be filled with null (using allowMissingColumns=True)
Use Case	When schemas are identical	When schemas differ in order or completeness

unionByName is highly useful in real-world scenarios where datasets may have evolved differently but still share common columns.

SQL vs Pyspark Execution of Code :

SQL vs. PySpark Execution

SQL Query:

```
SELECT StoreID, SUM(SalesAmount) AS TotalSales
FROM sales
WHERE SalesAmount > 100
GROUP BY StoreID
HAVING SUM(SalesAmount) > 1000
ORDER BY TotalSales DESC
LIMIT 5;
```

1. **FROM**: Read the sales table.
2. **WHERE**: Filter rows where SalesAmount > 100.
3. **GROUP BY**: Group data by StoreID.
4. **HAVING**: Keep only groups with SUM(SalesAmount) > 1000.
5. **SELECT**: Calculate and select StoreID and TotalSales.
6. **ORDER BY**: Sort by TotalSales in descending order.
7. **LIMIT**: Fetch top 5 rows.

PySpark Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum
spark = SparkSession.builder.appName("SQLvsPySpark").getOrCreate()
```

```
df = spark.read.csv("sales.csv", header=True, inferSchema=True)
result = (
   df.filter(col("SalesAmount") > 100)  # Filter rows
   .groupBy("StoreID")                  # Group by StoreID
   .agg(sum("SalesAmount").alias("TotalSales"))  # Aggregate
   .filter(col("TotalSales") > 1000)   # HAVING condition
   .orderBy(col("TotalSales").desc())  # Sort by TotalSales
   .limit(5)                           # Limit results
)
result.show()
```

1. filter: Filters rows (WHERE equivalent).
2. groupBy + agg: Groups and calculates total sales (GROUP BY + SUM).
3. filter: Applies a condition to the aggregated result (HAVING equivalent).
4. orderBy: Sorts the data.
5. limit: Limits the result.

Necessary Imports :

1.

```
from pyspark.sql import SparkSession  # To create a SparkSession (entry point for DataFrame API)
from pyspark import SparkContext, SparkConf  # For low-level RDD operations
```

2.

```
from pyspark.sql import DataFrame  # For working with DataFrames
from pyspark.sql.functions import *  # Common SQL functions (e.g., col, lit, when, sum, avg, etc.)
from pyspark.sql.types import *  # For defining custom schema (StructType, StructField)
```

3.

```
from pyspark.rdd import RDD  # For working with RDD objects
```

4.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col, row_number, rank, dense_rank, lead, lag, sum
```

Writing window function:

```
windowSpec = Window.partitionBy("Employee").orderBy("ID").rowsBetween(Window.unboundedPreceding, Window.currentRow)

df_with_running_total = df.withColumn("CumulativeSales", sum("Sales").over(windowSpec))
```

Common Window Functions:

Here are some of the common window functions in PySpark:

- row_number(): Assigns a unique sequential number to rows.
- rank(): Assigns a rank to rows with the same value; skips rank in case of ties.
- dense_rank(): Like rank(), but doesn't leave gaps in ranking.
- ntile(n): Divides rows into n buckets (used for quantile-based calculations).
- lag(): Accesses the previous row's value.
- lead(): Accesses the next row's value.
- sum(): Calculates the cumulative sum within the window.
- avg(): Calculates the cumulative average within the window.
- first(), last(): Fetches the first and last value in a window.
- percent_rank(): Calculates the relative rank of a row within its window.
- rowsBetween() & rangeBetween(): Defines a custom frame within the window.

- **reduceByKey**:
  - Works only on RDDs.
  - Efficient for straightforward, key-value aggregation operations.
  - Offers lower-level control but less flexibility.
- **groupBy().agg**:
  - Designed for DataFrames, offering SQL-like abstraction.
  - Supports multiple, flexible aggregation operations in a single step.
  - Slightly higher overhead due to the full shuffle of grouped data.

For structured data or SQL-like operations, groupBy().agg() is the better choice. For simpler, lower-level key-value aggregations with performance optimization, use reduceByKey().

JSON files :

Reading

```
df = spark.read.format('json')\
      .option('multiline','true')\
```

```
                .load('file_name.json')
```

Nested Json

1. Exploding Array Column

```
[
{"name": "Alice", "friends": ["Bob", "Charlie"]},
 {"name": "David", "friends": ["Eve", "Frank"]}
]
```

Here, friends is an **array**.

```
from pyspark.sql.functions import explode, col
df = spark.read.json("nested_array.json")

# Exploding the 'friends' array column
df_exploded = df.withColumn("friend", explode(col("friends")))

df_exploded.show()
```

Output :

```
+-----+----------------+-------+
| name|         friends| friend|
+-----+----------------+-------+
|Alice|[Bob, Charlie]  |   Bob |
|Alice|[Bob, Charlie]  |Charlie|
|David|[Eve, Frank]    |   Eve |
|David|[Eve, Frank]    |  Frank|
+-----+----------------+-------+
```

2. Accessing Fields Inside a Struct (Dictionary):

If a column contains a **struct** (which is essentially a dictionary), you don't use explode(). Instead, you extract the individual fields inside the struct by referencing them with col().

```
[
  {"name": "Alice", "address": {"city": "New York", "zip": "10001"}},
  {"name": "David", "address": {"city": "Los Angeles", "zip": "90001"}}
]

df_with_address = df.select( col("name"), col("address.city").alias("city"), col("address.zip").alias("zip") )
```

```
+-----+-------------+-----+
| name|         city|  zip|
+-----+-------------+-----+
|Alice|    New York |10001|
|David|Los Angeles  |90001|
+-----+-------------+-----+
```

3.Handling Nested Structs

```
[
  {"name": "Alice", "address": {"city": "New York", "contact": {"phone": "12345", "email": "alice@example.com"}}},
  {"name": "David", "address": {"city": "Los Angeles", "contact": {"phone": "67890", "email": "david@example.com"}}}
]

# Accessing fields from a nested struct (address.contact)
df_nested_struct = df.select(
    "name",
    col("address.city").alias("city"),
    col("address.contact.phone").alias("phone"),
    col("address.contact.email").alias("email")
)

df_nested_struct.show()
```

4.Nested struct and Array

Data =

```
# The given JSON data
data = [
    {
        "id": 1,
        "name": "Alice",
        "address": {
            "city": "New York",
            "state": "NY",
            "contacts": [
                {"type": "phone", "value": "123-456-7890"},
                {"type": "email", "value": "alice@example.com"}
            ]
```

```
      },
      "purchases": [
        {"item": "Laptop", "price": 1200.99},
        {"item": "Mouse", "price": 25.5}
      ]
    },
    {
      "id": 2,
      "name": "Bob",
      "address": {
        "city": "San Francisco",
        "state": "CA",
        "contacts": [
          {"type": "phone", "value": "987-654-3210"}
        ]
      },
      "purchases": [
        {"item": "Keyboard", "price": 45.75}
      ]
    }
]

# Convert to DataFrame
df = spark.createDataFrame(data)
df.show(truncate=False)
df.printSchema()
```

```
root
 |-- address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- contacts: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- type: string (nullable = true)
 |    |    |    |-- value: string (nullable = true)
 |    |-- state: string (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- purchases: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- item: string (nullable = true)
 |    |    |-- price: double (nullable = true)
```

.exploding the arrays -> accesing the structs .

```
df_flattened = (
    df
    .withColumn("contact", explode(col("address.contacts")))
    .withColumn("purchase", explode(col("purchases")))
    .select(
        "id",
        "name",
        col("address.city").alias("city"),
        col("address.state").alias("state"),
        col("contact.type").alias("contact_type"),
        col("contact.value").alias("contact_value"),
        col("purchase.item").alias("item"),
        col("purchase.price").alias("price")
    )
)
df_flattened.show(truncate=False)
```

2.

Data =

```
data = [
    {
      "order_id": 101,
      "customer": {
        "name": "Alice",
        "contact": {
          "phone": "123-456-7890",
          "email": "alice@example.com"
        }
      },
      "items": [
        {
          "item_id": 1,
          "details": {"name": "Laptop", "price": 1200.99},
          "discounts": [{"type": "holiday", "amount": 100}, {"type": "coupon", "amount": 50}]
        },
        {
```

```
            "item_id": 2,
            "details": {"name": "Mouse", "price": 25.5},
            "discounts": [{"type": "clearance", "amount": 5}]
          }
        ]
    },
    {
      "order_id": 102,
      "customer": {
        "name": "Bob",
        "contact": {
          "phone": "987-654-3210",
          "email": "bob@example.com"
        }
      },
      "items": [
        {
          "item_id": 3,
          "details": {"name": "Keyboard", "price": 45.75},
          "discounts": []
        }
      ]
    }
]
```

If an item does not have discounts, explode on discounts will result in no rows for that item.
To handle this, you can use posexplode_outer or **explode_outer** to include rows with null for discount_type and discount_amount.

```
df_flattened = (
  df.withColumn("item", explode(col("items")))  # Explode the items array
  .withColumn("discount", explode(col("item.discounts")))  # Explode the discounts array
  .select(
    col("order_id"),
    col("customer.name").alias("customer_name"),
    col("customer.contact.phone").alias("customer_phone"),
    col("customer.contact.email").alias("customer_email"),
    col("item.item_id").alias("item_id"),
    col("item.details.name").alias("item_name"),
    col("item.details.price").alias("item_price"),
    col("discount.type").alias("discount_type"),
    col("discount.amount").alias("discount_amount"),
  )
)

df_flattened.show(truncate=False)
```

order_id	customer_name	customer_phone	customer_email	item_id	item_name	item_price	discount_type	discount_amount
101	Alice	123-456-7890	alice@example.com	1	Laptop	1200.99	holiday	100
101	Alice	123-456-7890	alice@example.com	1	Laptop	1200.99	coupon	50
101	Alice	123-456-7890	alice@example.com	2	Mouse	25.5	clearance	5
102	Bob	987-654-3210	bob@example.com	3	Keyboard	45.75	null	null

Spark Session :

The SparkSession is an entry point which allows you to interact with Spark's APIs for DataFrame and SQL functionalities and acts as a unified interface for interacting with Spark.

Creating a spark session :

```
from pyspark.sql import SparkSession

# Create a Spark Session
spark = SparkSession.builder \
    .appName("MySparkApp") \
    .getOrCreate()

# Print the Spark session details
print(spark.version)

#stop the session
spark.stop()
```

Pyspark Jobs / Stages/ tasks

- **Job**: A **job** is triggered by an **action** (e.g., collect(), show(), save()) and represents the entire computation process from the start to completion. It may consist of multiple stages depending on the transformations.

- **Stage**: A **stage** is a set of operations that can be executed in parallel. Spark divides a job into stages based on **wide transformations** (e.g., groupBy(), join()) which require shuffling. A stage is separated by a shuffle barrier, and **each stage represents a set of operations that can be executed on a partition without moving data between partitions.**

- **Task**: A **task** is the smallest unit of work in Spark and corresponds to a partition of the data. Each stage is divided into tasks, and tasks are executed in parallel across the available executors. The number of tasks in a stage depends on the number of partitions.

Key Points:

1. **Jobs** are triggered by actions.
2. **Stages** are the steps within a job, divided based on shuffling.
3. **Tasks** are the parallel units of work within a stage, one per partition .

Example :

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

# Initialize SparkSession
spark = SparkSession.builder \
   .appName("JobsStagesTasksExample") \
   .getOrCreate()

# Sample data for two DataFrames
data1 = [(1, "Alice", 29), (2, "Bob", 35), (3, "Charlie", 40), (4, "David", 29)]
data2 = [(1, "HR"), (2, "Engineering"), (3, "Marketing"), (4, "Sales")]

# Create DataFrames
df1 = spark.createDataFrame(data1, ["id", "name", "age"])
df2 = spark.createDataFrame(data2, ["id", "department"])

# Step 1: Filter DataFrame to include only rows where age > 30
df1_filtered = df1.filter(col("age") > 30)

# Step 2: Join the filtered DataFrame with df2 on the 'id' column
df_joined = df1_filtered.join(df2, df1_filtered.id == df2.id)

# Step 3: Perform groupBy on 'department' and count the rows in each department
df_grouped = df_joined.groupBy("department").count()

# Trigger action to execute transformations
df_grouped.show()

# Stop SparkSession
spark.stop()
```

Explanation: Jobs, Stages, and Tasks

1. Jobs

- **A Spark job** is triggered by an **action** like show(), count(), or write().
- In this case:
  - The df_grouped.show() action triggers **1 job** that includes all transformations (filter, join, and groupBy).

2. Stages

Spark breaks down the job into **stages** based on **wide transformations** (which require shuffling data).
Here's how it works:

1. **Stage 1: Filter**
   - The filter() operation is a **narrow transformation**, so no shuffle occurs here.
   - This stage includes reading the data and applying the filter.
2. **Stage 2: Join**
   - The join() operation is a **wide transformation** that involves shuffling data between partitions to align keys for the join.
3. **Stage 3: GroupBy**
   - The groupBy() operation is another **wide transformation** that causes a shuffle to aggregate data by the department column.

Total Stages = 3:

1. Read and filter.
2. Shuffle for join.
3. Shuffle for groupBy and aggregation.

3. Tasks

- The **number of tasks** in each stage depends on the **number of partitions**:
  - By default, Spark uses **200 partitions** for shuffle operations (spark.sql.shuffle.partitions = 200).

Here's the breakdown:

1. **Stage 1 (Filter)**: Tasks = Number of input partitions.
   - If df1 has 4 partitions (assume default), this stage will have **4 tasks**.

2. **Stage 2 (Join)**: Tasks = Number of shuffle partitions.
    - Default = **200 tasks**.
3. **Stage 3 (GroupBy)**: Tasks = Number of shuffle partitions.
    - Default = **200 tasks**.

Total Tasks = 4 (Stage 1) + 200 (Stage 2) + 200 (Stage 3) = 404 tasks.