To write a code we need to start with these codes

```
Pyspark Program:


#Importing the Libraries
from pyspark import SparkContext
from pyspark.sql import SparkSession
#Creating a SparkContext
sc = SparkContext.getOrCreate()
#create a sparksession
spark = SparkSession.builder.appName('PySpark DataFrame From RDD').getOrCreate()
```

1st 4 lines wii be same for call code
Next:

```
#create a rdd
rdd = sc.parallelize([('C',85,76,87,91), ('B',85,76,87,91), ("A", 85,78,96,92), ("A", 92,76,89,96)], 4)
print(type(rdd))
```

Next:

```
#Converting the RDD into PySpark DataFrame
#schema rdd,rdd,dataframe
sub = ['Division','English','Mathematics','Physics','Chemistry']
marks_df= spark.createDataFrame(rdd, schema=sub)
```

Spark basics

Spark context

`pyspark.SparkContext` is an entry point to the PySpark functionality that is used to communicate with the cluster and to create an RDD, accumulator, and broadcast variables. The Spark driver program creates and uses SparkContext to connect to the cluster manager to submit PySpark jobs, and know what resource manager (YARN, Mesos, or Standalone) to communicate to. It is the heart of the PySpark application.

SparkContext

```
[20] # Create SparkSession from builder
     from pyspark.sql import SparkSession
     spark = SparkSession.builder.master("local[1]") \
                         .appName('SparkByExamples.com') \
                         .getOrCreate()
     print(spark.sparkContext)
     print("Spark App Name : "+ spark.sparkContext.appName)
```

```
<SparkContext master=local[*] appName=Life Stage Classification>
Spark App Name : Life Stage Classification
```

```
# SparkContext stop() method
spark.sparkContext.stop()
```

You can stop the SparkContext by calling the `stop()` method.If you want to create another, you need to shutdown it first by using stop() method and create a new SparkContext.

Create PySpark RDD:

Once you have a SparkContext object, you can create a PySpark RDD in several ways. Below I have used the `range()` function.

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("SimpleRDDApp") \
    .master("local[*]") \
    .getOrCreate()

# Get the SparkContext
sc = spark.sparkContext
rdd = sc.range(1, 5)
print(rdd.collect())
```

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

We can create rdd in three ways

- From a Text File
- From an existing Python collection
- By transforming another RDD

RDD Operations:

Transformation	What It Does
`map`	Applies a function to each element and returns a new RDD. Example: multiply each number by 2.
`filter`	Filters elements based on a condition. Example: keep only even numbers.
`flatMap`	Like `map`, but flattens the result. Good for splitting text into words.
`groupByKey`	Groups values with the same key (only for `(key, value)` RDDs).
`reduceByKey`	Combines values with the same key using a function like sum.
`join`	Joins two RDDs by key. Like SQL JOIN.
`union`	Merges two RDDs together.
`distinct`	Removes duplicate elements.

RDD (Resilient Distributed Dataset) is a core building block of PySpark. It is a fault-tolerant, immutable, distributed collection of objects. Immutable means that once you create an RDD, you cannot change it. The data within RDDs is segmented into logical partitions, allowing for distributed computation across multiple nodes within the cluster.

Step 1: Initialize `SparkSession`

Before you can use RDDs, you need to create a `SparkSession` (which internally sets up `SparkContext`).

```
# Imports
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder
      .master("local[1]")
      .appName("SparkByExamples.com")
      .getOrCreate()
```

# Get SparkContext - add this in the last
sc = spark.sparkContext

📌 What does each method mean?

- `master("local[x]")`: Runs Spark locally with `x` CPU cores.
- `appName("...")`: Sets your application name.
- `getOrCreate()`: Reuses an existing session or creates a new one.

Using sparkContext.parallelize()

By using `parallelize()` function of SparkContext (sparkContext.parallelize()) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This method of creating an RDD is used when you already have data in memory that is either loaded from a file or from a database. and all data must be present in the driver program prior to creating RDD.

4. RDD Creation

You can create RDDs in **two main ways**:

1. From a **Python collection** using `.parallelize()`
2. By **loading files** (like `.txt`, `.csv`, etc.) using `.textFile()` or `.wholeTextFiles()`

What is parallelize() in PySpark?

parallelize() is a method used to **create an RDD from an existing Python collection (like a list or array)** that is already loaded in memory (in your driver program).

💡 **When to use it?**

Use parallelize() when:

- You already have data in Python (like a list or result from a DB/file).
- You want to distribute that data across multiple nodes for parallel processing.

✅ **Syntax**

rdd = spark.sparkContext.parallelize(your_data, num_partitions)

- your_data: Your list or array.
- num_partitions *(optional)*: How many chunks to split the data into.

🧪 Example:

```
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
    .master("local[1]") \
    .appName("ParallelizeExample") \
    .getOrCreate()
# Create RDD from parallelize
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd = spark.sparkContext.parallelize(data)
print(rdd.collect())
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

✅ Ways to Create an RDD:

1. **From Python collection (in-memory) - same example use for parallelize**
2. **From external storage (disk)**

Using sparkContext.textFile()

Use the textFile() method to read a .txt file into RDD.

```
# Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

Using sparkContext.wholeTextFiles()

wholeTextFiles() function returns a PairRDD with the key being the file path and the value being file content.

```
# Read entire file into a RDD as single record.
rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
```

Besides using text files, we can also create RDD from CSV file, JSON, and more formats.

3. Empty RDDs

Create an Empty RDD

```
[33] empty_rdd = sc.emptyRDD()
     print(empty_rdd.collect())
```

```
[]
```

Create an Empty RDD with 10 partitions

```
rdd_with_partitions = sc.parallelize([], 10)
print(rdd_with_partitions.collect())
```

```
[]
```

RDD Partitions

When we use `parallelize()`, `textFile()` or `wholeTextFiles()` methods of SparkContext to initiate RDD, it automatically splits the data into partitions based on resource availability.

getNumPartitions() – This is an RDD function that returns a number of partitions your dataset split int

Check number of partitions:

```
# Get partition count
print("Initial partition count:"+str(rdd.getNumPartitions()))

# Outputs: Initial partition count:2
```

Set parallelize manually – We can also set a number of partitions manually, all we need is to pass a number of partitions as the second parameter to these functions for example;

```
# Set partitions manually
sparkContext.parallelize([1,2,3,4,56,7,8,9,12,3], 10)
```

Repartition and Coalesce

Sometimes, we may need to repartition the RDD, PySpark provides two ways to repartition; first using `repartition()` method, which shuffles data from all nodes also called full shuffle and second coalesce() method which shuffles data from minimum nodes, for examples if you have data in 4 partitions and doing `coalesce(2)` moves data from just 2 nodes.

Repartition (full shuffle, expensive):

```
rdd_new = rdd.repartition(4)
print(rdd_new.collect())
```

```
[1, 2, 3]
```

Coalesce (less shuffling, faster for reducing partitions):

```
rdd_small = rdd.coalesce(2)
print(rdd_small.collect())
```

```
[1, 2, 3]
```

Note: repartition() or coalesce() methods also return a new RDD.

PySpark RDD Operations

RDD operations are the core transformations and actions performed on RDDs

Two Types:

Transformations – Create a new RDD (lazy, nothing runs until an action is called).

Actions – Trigger computation and return results (or save data).

Each line from the file becomes one RDD element.

🪵 Step 1: Load a text file

```
rdd = spark.sparkContext.textFile("/content/sample_text.txt")
```

- Each line from the file becomes one RDD element.

✂️ Step 2: flatMap() – Split lines into words

```
rdd2 = rdd.flatMap(lambda line: line.split(" "))
print("\nWords after flatMap:")
print(rdd2.collect())
```

```
Words after flatMap:
['PySpark', 'is', 'an', 'interface', 'for', 'Apache', 'Spark', 'in', 'Python.', 'It', 'allows', 'you', 'to', 'write', 'Spark', 'applicatic
```

- **Purpose**: Splits each line into individual words.
- **Why flatMap?**: It flattens the result. For example:
    - Line: `"Hello world"` → `["Hello", "world"]`
    - Normal `map()` would return a list of lists.

`flatMap()` returns a flat list: `["Hello", "world"]`

➕ Step 3: map() – Add (word, 1) to each word

```
rdd3 = rdd2.map(lambda word: (word, 1))
print("\nAfter map (word, 1):")
print(rdd3.collect())
```

```
After map (word, 1):
[('PySpark', 1), ('is', 1), ('an', 1), ('interface', 1), ('for', 1), ('Apache', 1), ('Spark', 1), ('in', 1), ('Python.', 1), ('It', 1), ('allows
```

- Converts each word into a key-value pair:
  `'word' → ('word', 1)`

🧮 Step 4: reduceByKey() – Count words

```
rdd4 = rdd3.reduceByKey(lambda a, b: a + b)
print("\nAfter reduceByKey (word counts):")
print(rdd4.collect())
```

```
After reduceByKey (word counts):
[('PySpark', 2), ('an', 1), ('interface', 1), ('for', 1), ('It', 1), ('you', 1), ('to', 2), ('write', 1), ('applications', 1), ('using
```

Purpose: Adds up values for each word (key).

Example:

[("PySpark", 1), ("PySpark", 1)] → ("PySpark", 2)

Efficient: Combines values on the same partition before shuffling across the network.

Step 5: map() to flip key-value

```
rdd5 = rdd4.map(lambda x: (x[1], x[0])).sortByKey()
print("\nSorted by count:")
print(rdd5.collect())
```

```
Sorted by count:
[(1, 'an'), (1, 'interface'), (1, 'for'), (1, 'It'), (1, 'you'), (1, 'write'), (1, 'applications'), (1, 'using'), (1, 'Python'), (
```

- **Purpose**: Swaps from `(word, count)` to `(count, word)`
- **Why?**: So we can sort by count (which is now the key).

🔢 Step 6: sortByKey() – Sort by frequency

```
# Sort by count (ascending order)
rdd5 = rdd4.map(lambda x: (x[1], x[0])).sortByKey()

# Print sorted results
print("\nSorted word counts (ascending):")
for count, word in rdd5.collect():
    print(f"{word}: {count}")
```

```
Sorted word counts (ascending):
an: 1
interface: 1
for: 1
It: 1
you: 1
write: 1
applications: 1
using: 1
Python: 1
helps: 1
and: 1
engineers: 1
work: 1
with: 1
core: 1
of: 1
Apache: 1
in: 1
Python.: 1
allows: 1
APIs.: 1
scientists: 1
large-scale: 1
processing.: 1
RDD: 1
a: 1
```

```
# Sort by count (descending order)
rdd5 = rdd4.map(lambda x: (x[1], x[0])).sortByKey(ascending=False)

# Print sorted results
print("\nSorted word counts (desc):")
for count, word in rdd5.collect():
    print(f"{word}: {count}")
```

```
Sorted word counts (desc):
PySpark: 2
to: 2
is: 2
Spark: 2
data: 2
an: 1
interface: 1
for: 1
It: 1
you: 1
write: 1
applications: 1
using: 1
Python: 1
helps: 1
and: 1
engineers: 1
work: 1
with: 1
core: 1
of: 1
Apache: 1
in: 1
Python.: 1
allows: 1
APIs.: 1
scientists: 1
```

- First switch from `(word, count)` → `(count, word)`
- Then sort by count (key)
- Final output will be: lowest → highest frequency

🖨️ Display Results

print(rdd5.collect())

`(action)` - count(),first(),max(),reduce(),take(),collect(),saveAsTextFile()

PYSPARK DATAFRAME

A DataFrame in PySpark is:

- A distributed collection of data organized into rows and columns.
- Similar to a table in a database or a Pandas DataFrame.
- Stored across multiple machines in a Spark cluster, which allows Spark to process big data in parallel.

💡 Difference from Pandas:

- Pandas DataFrame: Runs on one machine, data must fit in memory.
- PySpark DataFrame: Runs on many machines in parallel, handles huge datasets.

Why PySpark DataFrame is Faster than Pandas

- PySpark: Uses cluster computing and parallel processing → good for large datasets.
- Pandas: Single-machine in-memory processing → better for small to medium datasets.

✅ If you have GBs or TBs of data, use PySpark.

✅ If you have MBs of data, Pandas is enough.

Creating a DataFrame

Ways to create Dataframe

- Create DataFrame from RDD
  `- Using to DF()`
  `- Using createDataFrame()`

- Create DataFrame from Files
  `- From CSV`
  `- From Text File`
  `- From JSON File`

- Create DataFrame From Python List

Create DataFrame from Python List

```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]")\
        .appName("DataFrame Example")\
        .getOrCreate()
data = [
      ('James', '', 'Smith', '1991-04-01', 'M', 3000),
      ('Michael', 'Rose', '', '2000-05-19', 'M', 4000),
      ('Robert', '', 'Williams', '1978-09-05', 'M', 4000),
      ('Maria', 'Anne', 'Jones', '1967-12-01', 'F', 4000),
      ('Jen', 'Mary', 'Brown', '1980-02-17', 'F', -1)
]
#col names
columns = ["FirstName", "MiddleName","LastName", "DOB", "Gender", "Salary"]
df = spark.createDataFrame(data = data, schema = columns)
df.show()
'''
--- Note ---
local[1] → Spark will run with 1 core (single-threaded).
local[2] → Spark will run with 2 cores.
local[*] → Spark will run with all cores available.

'''
```

```
+---------+----------+--------+----------+------+------+
|FirstName|MiddleName|LastName|       DOB|Gender|Salary|
+---------+----------+--------+----------+------+------+
|    James|          |   Smith|1991-04-01|     M|  3000|
|  Michael|      Rose|        |2000-05-19|     M|  4000|
|   Robert|          |Williams|1978-09-05|     M|  4000|
|    Maria|      Anne|   Jones|1967-12-01|     F|  4000|
|      Jen|      Mary|   Brown|1980-02-17|     F|    -1|
+---------+----------+--------+----------+------+------+
```

Check schema (column names & data types):

```
[60] df.printSchema()
```

```
root
 |-- FirstName: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- DOB: string (nullable = true)
 |-- Gender: string (nullable = true)
 |-- Salary: long (nullable = true)
```

Create DataFrame from RDD

Using toDF()

If you already have an RDD, you can convert it into a DataFrame using toDF().

Note: In PySpark, parallelize() is a method of SparkContext that is used to create an RDD (Resilient Distributed Dataset) from a Python collection (like a list).

It takes your local data and distributes it across the Spark cluster (or CPU cores if running locally), allowing parallel processing.

```
# Create RDD
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]
spark = SparkSession.builder.appName('DataFrame by RDD').getOrCreate()
rdd = spark.sparkContext.parallelize(data)
# Convert RDD to DataFrame without column names
dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
dfFromRDD1.show()
```

```
root
 |-- _1: string (nullable = true)
 |-- _2: string (nullable = true)

+------+------+
|    _1|    _2|
+------+------+
|  Java| 20000|
|Python|100000|
| Scala|  3000|
+------+------+
```

Adding Column Names

```
columns = ["Languages", "Users_count"]
dfFromRDD1 = rdd.toDF(columns)
dfFromRDD1.printSchema()
dfFromRDD1.show()
```

```
root
 |-- Languages: string (nullable = true)
 |-- Users_count: string (nullable = true)

+---------+-----------+
|Languages|Users_count|
+---------+-----------+
|     Java|      20000|
|   Python|     100000|
|    Scala|       3000|
+---------+-----------+
```

Using createDataFrame()

```
[ ] columns = ["Languages", "users_count"]
    df2 = spark.createDataFrame(rdd).toDF(*columns)
    df2.show()
```

```
+---------+-----------+
|Languages|users_count|
+---------+-----------+
|     Java|      20000|
|   Python|     100000|
|    Scala|       3000|
+---------+-----------+
```

Create DataFrame from Files

From CSV

header=True → uses the first row as column names.

inferSchema=True → automatically detects column data types.

```
df_csv = spark.read.csv("/content/Marks_data.csv", header = True, inferSchema=True)
df_csv.show()
```

```
+----+--------+--------+---+
|Name|M1 Score|M2 Score|age|
+----+--------+--------+---+
|Alex|      62|      80| 20|
|Brad|      45|      56| 19|
|Joey|      85|      98| 21|
|NULL|      54|      79| 20|
|abhi|    NULL|    NULL| 20|
+----+--------+--------+---+
```

From Text File

This creates a single column DataFrame with column name value.

```
[ ] df_txt = spark.read.text("/content/test.txt")
    df_txt.show()
```

```
+--------+
|   value|
+--------+
| Ananya |
|    Arun|
|Saveetha|
|  Vimala|
+--------+
```

From JSON File

Spark automatically maps JSON keys to columns.

```
[ ] df_json = spark.read.json("/content/data.json.txt")
    df_json.show()
```

```
+---+-------+
|age|   name|
+---+-------+
| 25|   Teja|
| 30|  Shiva|
| 28|Charlie|
+---+-------+
```

Key Features of PySpark DataFrames

- Distributed → works on multiple machines.
- Optimized → uses Spark's engine for speed.
- Schema-aware → knows column names and data types.
- Supports SQL → you can run SQL queries using df.createOrReplaceTempView() and spark.sql().

DataFrame Operations

1. RENAMING

Rename a Single Column

The simplest way is to use withColumnRenamed().

```
[ ] df = df.withColumnRenamed("dob","DateOfBirth")
    df.printSchema()
```

```
root
 |-- FirstName: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- DateOfBirth: string (nullable = true)
 |-- Gender: string (nullable = true)
 |-- Salary: long (nullable = true)
```

(from dob to dateofbirth)

Rename Multiple Columns

You can chain multiple withColumnRenamed() calls:

```
df2 = df.withColumnRenamed("dob", "DateOfBirth")\
        .withColumnRenamed("Salary", "Salary_amount")
df2.printSchema()
```

```
root
 |-- FirstName: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- DateOfBirth: string (nullable = true)
 |-- Gender: string (nullable = true)
 |-- Salary_amount: long (nullable = true)
```

OR dynamically loop through a dictionary:

```
[ ] rename_map = {"FirstName": "First_Name", "LastName":"Last_Name"}
    for old_col, new_col in rename_map.items():
      df = df.withColumnRenamed(old_col, new_col)
    df.printSchema()
```

```
root
 |-- First_Name: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- Last_Name: string (nullable = true)
 |-- DateOfBirth: string (nullable = true)
 |-- Gender: string (nullable = true)
 |-- Salary: long (nullable = true)
```

Rename Nested Columns using withColumn()

```
[ ] from pyspark.sql.functions import col
    df_new = df.withColumn("fname", col("First_Name"))\
              .withColumn("mname", col("MiddleName"))\
              .withColumn("lname", col("Last_Name"))\
              .drop("First_Name", "MiddleName", "Last_Name")
    df_new.printSchema()
```

```
root
 |-- DateOfBirth: string (nullable = true)
 |-- Gender: string (nullable = true)
 |-- Salary: long (nullable = true)
 |-- fname: string (nullable = true)
 |-- mname: string (nullable = true)
 |-- lname: string (nullable = true)
```

Rename All Columns using toDF()

If your DataFrame is flat (not nested), the easiest way is toDF():

```
new_columns = ["First_Name", "Middle_Name", "Last_Name", "DateOfBirth", "gender", "salary"]
df_renamed = df.toDF(*new_columns)
df_renamed.printSchema()
```

```
root
 |-- First_Name: string (nullable = true)
 |-- Middle_Name: string (nullable = true)
 |-- Last_Name: string (nullable = true)
 |-- DateOfBirth: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: long (nullable = true)
```

Selecting

- select() is a transformation function in PySpark DataFrame.
- It is used to select specific columns (single, multiple, by index, or nested).
- It always returns a new DataFrame (because DataFrames in Spark are immutable).

Select Single or Multiple Columns

```
#single column
df.select("FirstName").show()
#multiple colum
df.select("FirstName", "LastName").show()
#using dataframe object
df.select(df.FirstName, df.LastName).show()
#using bracket notation
df.select(df["FirstName"], df["LastName"]).show()
#using col() function
from pyspark.sql.functions import col
df.select(col("DOB"),col("Salary")).show()
```

Select All Columns from a List

If you have a list of column names, you can unpack it:

```
columns = ["FirstName", "MiddleName", "LastName", "DOB", "Gender", "Salary"]
#using list unpacking
df.select(*columns).show()
#using list comprehension
df.select([col for col in df.columns]).show()
#select all col
df.select("*").show()
```

```
+---------+----------+--------+----------+------+------+
|FirstName|MiddleName|LastName|       DOB|Gender|Salary|
+---------+----------+--------+----------+------+------+
|    James|          |   Smith|1991-04-01|     M|  3000|
|  Michael|      Rose|        |2000-05-19|     M|  4000|
|   Robert|          |Williams|1978-09-05|     M|  4000|
|    Maria|      Anne|   Jones|1967-12-01|     F|  4000|
|      Jen|      Mary|   Brown|1980-02-17|     F|    -1|
+---------+----------+--------+----------+------+------+

+---------+----------+--------+----------+------+------+
|FirstName|MiddleName|LastName|       DOB|Gender|Salary|
+---------+----------+--------+----------+------+------+
|    James|          |   Smith|1991-04-01|     M|  3000|
|  Michael|      Rose|        |2000-05-19|     M|  4000|
|   Robert|          |Williams|1978-09-05|     M|  4000|
|    Maria|      Anne|   Jones|1967-12-01|     F|  4000|
|      Jen|      Mary|   Brown|1980-02-17|     F|    -1|
+---------+----------+--------+----------+------+------+

+---------+----------+--------+----------+------+------+
|FirstName|MiddleName|LastName|       DOB|Gender|Salary|
+---------+----------+--------+----------+------+------+
|    James|          |   Smith|1991-04-01|     M|  3000|
|  Michael|      Rose|        |2000-05-19|     M|  4000|
|   Robert|          |Williams|1978-09-05|     M|  4000|
|    Maria|      Anne|   Jones|1967-12-01|     F|  4000|
|      Jen|      Mary|   Brown|1980-02-17|     F|    -1|
+---------+----------+--------+----------+------+------+
```

Select Columns by Index

```
# Select first 3 columns
df.select(df.columns[:3]).show(3)
# Select columns 2 to 4
df.select(df.columns[2:4]).show(3)
```

```
+---------+----------+--------+
|FirstName|MiddleName|LastName|
+---------+----------+--------+
|    James|          |   Smith|
|  Michael|      Rose|        |
|   Robert|          |Williams|
+---------+----------+--------+
only showing top 3 rows

+--------+----------+
|LastName|       DOB|
+--------+----------+
|   Smith|1991-04-01|
|        |2000-05-19|
|Williams|1978-09-05|
+--------+----------+
only showing top 3 rows
```

Filtering

- filter() is used to filter rows in a DataFrame based on a condition.
- It is similar to the SQL WHERE clause.
- Returns a **new DataFrame**, doesn't change the original.
- You can use `.filter()` or `.where()` — both do the same thing.

Syntax:

df.filter(condition)

df.where(condition) #same as filter

condition can be:

- A column expression
- A SQL expression (string)
- A function like startswith(), like(), isin(), etc.

```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('DataFrame Created For Filtering.com').getOrCreate()
# Data as nested lists
data = [
    [["James","","Smith"], ["Java","Scala","C++"], "OH", "M"],
    [["Anna","Rose",""], ["Spark","Java","C++"], "NY", "F"],
    [["Julia","","Williams"], ["CSharp","VB"], "OH", "F"],
    [["Maria","Anne","Jones"], ["CSharp","VB"], "NY", "M"],
    [["Jen","Mary","Brown"], ["CSharp","VB"], "NY", "M"],
    [["Mike","Mary","Williams"], ["Python","VB"], "OH", "M"]
]
# Column names
columns = ["name", "languages", "state", "gender"]
# Create DataFrame with schema
df = spark.createDataFrame(data, columns)
df.show(truncate=False)
df.printSchema()
```

```
+----------------------+------------------+-----+------+
|name                  |languages         |state|gender|
+----------------------+------------------+-----+------+
|[James, , Smith]      |[Java, Scala, C++]|OH   |M     |
|[Anna, Rose, ]        |[Spark, Java, C++]|NY   |F     |
|[Julia, , Williams]   |[CSharp, VB]      |OH   |F     |
|[Maria, Anne, Jones]  |[CSharp, VB]      |NY   |M     |
|[Jen, Mary, Brown]    |[CSharp, VB]      |NY   |M     |
|[Mike, Mary, Williams]|[Python, VB]      |OH   |M     |
+----------------------+------------------+-----+------+

root
 |-- name: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- languages: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- state: string (nullable = true)
```

1. Filter with Column Expressions

Filter using equal condition

```
#filter example
df.filter(df.state == "OH").show()
#where example
df.where(df.state == "NY").show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+
```

Filter using not equal

```
#Using != operator
df.filter(df.state != "OH").show()
#Using ~ (Negation) operator
df.where(~(df.state == "NY")).show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+
```

```
#Using != operator
df.filter(~(df.state == "OH")).show()
#Using ~ (Negation) operator
df.where(df.state != "NY").show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+
```

Note : If you use this both (! ~) both operators can be used for filter and where

2. Filter using SQL Expression

Filter using col()

```
from pyspark.sql.functions import col
df.filter(col("state") == "NY").show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+
```

Note: In PySpark, <> is just another way of writing "not equal to", just like !=.

```
df.filter("gender == 'M'").show()
df.filter("state != 'NY'").show()
df.filter("gender <> 'M'").show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|     [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Julia, , Williams]|      [CSharp, VB]|   OH|     F|
+--------------------+------------------+-----+------+
```

3. Filter using multiple conditions - AND Condition, OR Condition

```
# And condition
df.filter((df.state == "OH")&(df.gender == "M")).show()
# Or condition
df.filter((df.state =="NY")|(df.gender == "F")).show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+
```

4. Filter with functions - startswith(), **like()** – SQL style pattern match,**isin()** – IN condition

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

spark = SparkSession.builder.master("local[*]").appName("Filter Example").getOrCreate()

data = [
    ("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("Raman", "Finance", 3300),
    ("Scott", "Finance", 3900),
    ("Jen", "Finance", 3000),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
]

columns = ["Name", "Department", "Salary"]
df = spark.createDataFrame(data, columns)

# Filter: Salary > 3000
df.filter(col("Salary") > 3000).show()
```

```
+-------+----------+------+
|   Name|Department|Salary|
+-------+----------+------+
|Michael|     Sales|  4600|
| Robert|     Sales|  4100|
|  Raman|   Finance|  3300|
|  Scott|   Finance|  3900|
+-------+----------+------+
```

1. starts with() :

```
df.filter(col("Name").startswith("J")).show()
```

```
+-----+----------+------+
| Name|Department|Salary|
+-----+----------+------+
|James|     Sales|  3000|
|  Jen|   Finance|  3000|
| Jeff| Marketing|  3000|
+-----+----------+------+
```

2.like() :

```
[19] df.filter(col("Name").like("J%")).show()   # Names starting with J
```

```
+-----+----------+------+
| Name|Department|Salary|
+-----+----------+------+
|James|     Sales|  3000|
|  Jen|   Finance|  3000|
| Jeff| Marketing|  3000|
+-----+----------+------+
```

3.isin() :

```
df.filter(col("Department").isin("Sales", "Finance")).show()
```

```
+-------+----------+------+
|   Name|Department|Salary|
+-------+----------+------+
|  James|     Sales|  3000|
|Michael|     Sales|  4600|
| Robert|     Sales|  4100|
|  Maria|   Finance|  3000|
|  Raman|   Finance|  3300|
|  Scott|   Finance|  3900|
|    Jen|   Finance|  3000|
+-------+----------+------+
```

Filter using list values

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('DataFrame Created For Filtering.com').getOrCreate()

# Data as nested lists
data = [
    [["James","","Smith"], ["Java","Scala","C++"], "OH", "M"],
    [["Anna","Rose",""], ["Spark","Java","C++"], "NY", "F"],
    [["Julia","","Williams"], ["CSharp","VB"], "OH", "F"],
    [["Maria","Anne","Jones"], ["CSharp","VB"], "NY", "M"],
    [["Jen","Mary","Brown"], ["CSharp","VB"], "NY", "M"],
    [["Mike","Mary","Williams"], ["Python","VB"], "OH", "M"]
]

# Column names
columns = ["name", "languages", "state", "gender"]

# Create DataFrame without schema
df = spark.createDataFrame(data, columns)
df.show(truncate=False)
df.printSchema()
```

```
+----------------------+------------------+-----+------+
|name                  |languages         |state|gender|
+----------------------+------------------+-----+------+
|[James, , Smith]      |[Java, Scala, C++]|OH   |M     |
|[Anna, Rose, ]        |[Spark, Java, C++]|NY   |F     |
|[Julia, , Williams]   |[CSharp, VB]      |OH   |F     |
|[Maria, Anne, Jones]  |[CSharp, VB]      |NY   |M     |
|[Jen, Mary, Brown]    |[CSharp, VB]      |NY   |M     |
|[Mike, Mary, Williams]|[Python, VB]      |OH   |M     |
+----------------------+------------------+-----+------+
```

```
li = ["OH", "CA", "NY"]
#isin
df.filter(df.state.isin(li)).show()

#not isin
df.filter(~df.state.isin(li)).show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

+----+---------+-----+------+
|name|languages|state|gender|
+----+---------+-----+------+
+----+---------+-----+------+
```

Filter using string functions

Using startwith, endwith, contain function

```
# startswith
df.where(df.state.startswith("N")).show()
# endswith
df.where(df.state.endswith("H")).show()
# contains
df.where(df.state.contains("Y")).show()
```

```
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+
```

Filter using array column

```
from pyspark.sql.functions import array_contains
# array_contains
df.filter(array_contains(df.languages,"Java")).show()
# not array_contains
df.filter(~array_contains(df.languages,"CSharp")).show()
```

```
+----------------+------------------+-----+------+
|            name|         languages|state|gender|
+----------------+------------------+-----+------+
|[James, , Smith]|[Java, Scala, C++]|   OH|     M|
|  [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
+----------------+------------------+-----+------+

+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+
```

PySpark Distinct to Drop Duplicate Rows

Dropping

```
from os import truncate
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.appName("Dropping Example").getOrCreate()
simpleData = [
    ("James", "", "Smith", "36636", "NewYork", 3100),
    ("Michael", "Rose", "", "40288", "California", 4300),
    ("Robert", "", "Williams", "42114", "Florida", 1400),
    ("Maria", "Anne", "Jones", "39192", "Florida", 5500),
    ("Jen", "Mary", "Brown", "34561", "NewYork", 3000)
]
columns  = ["firstname", "middlename", "lastname", "id","location", "salary"]
df = spark.createDataFrame(data = simpleData, schema = columns)
df.printSchema()
df.show(truncate= False)
```

```
root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- location: string (nullable = true)
 |-- salary: long (nullable = true)

+---------+----------+--------+-----+----------+------+
|firstname|middlename|lastname|id   |location  |salary|
+---------+----------+--------+-----+----------+------+
|James    |          |Smith   |36636|NewYork   |3100  |
|Michael  |Rose      |        |40288|California|4300  |
|Robert   |          |Williams|42114|Florida   |1400  |
|Maria    |Anne      |Jones   |39192|Florida   |5500  |
|Jen      |Mary      |Brown   |34561|NewYork   |3000  |
+---------+----------+--------+-----+----------+------+
```

Drop a Single Column

In PySpark, if you want to remove (drop) one column from a DataFrame, you can use the `.drop()` method.
Syntax: **`df.drop(column_name)`**

But how you refer to the column can be done **in 3 different ways**:

1. Drop by String Name

`df.drop("firstname")`

- `"firstname"` is passed as a **string**.
- This is the most common way.
- Works well when column names are simple and known.

✅ **Use when**: You just want to drop a column by name, plain and simple.

2. Drop by Column Object

```
from pyspark.sql.functions import col
df.drop(col("firstname"))
```

- `col("firstname")` returns a **column object**.
- You are explicitly telling PySpark: “Here’s a column object, drop it.”
- Useful when combining with conditions or other transformations.

✅ **Use when**: You already use `col()` for other expressions and want consistency.

3. Drop by DataFrame.column Reference

```
df.drop(df.firstname)
```

- `df.firstname` directly accesses the column object from the DataFrame.
- This is more "Pythonic" but less flexible than using `col()`.
- It’s like saying: “Hey, drop this column right from my DataFrame.”

✅ **Use when**: You prefer dot notation or are already working with `df.column_name`.

EXAMPLE :

```
df.drop("firstname").printSchema()
df.drop(col("firstname")).printSchema()
df.drop(df.firstname).printSchema()
```

```
root
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- location: string (nullable = true)
 |-- salary: long (nullable = true)

root
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- location: string (nullable = true)
 |-- salary: long (nullable = true)

root
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- location: string (nullable = true)
 |-- salary: long (nullable = true)
```

Drop Multiple Columns

We can drop column in two ways

- Pass multiple arguments - Drops both `firstname` and `lastname` columns.
- Pass a tuple or list with * - The `*` unpacks the list and passes each element as a separate argument to `drop()`.This is cleaner when you already have a list of column names.

```
# method 1: Pass multiple arguments
df.drop("firstname","middlename" ,"lastname").printSchema()
# method 2: Pass a tuple or list with *
cols = ("id", "location")
df.drop(*cols).printSchema()
```

```
root
 |-- id: string (nullable = true)
 |-- location: string (nullable = true)
 |-- salary: long (nullable = true)

root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- salary: long (nullable = true)
```

Drop Rows

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()

data = [["1", None, "company 1"],
        ["2", "ojaswi", "company 2"],
        [None, "bobby", "company 3"],
        ["1", "sravan", "company 1"],
        ["2", "ojaswi", None],
        [None, "rohith", "company 2"],
        ["5", "gnanesh", "company 1"],
        ["2", None, "company 2"],
        ["3", "bobby", "company 3"],
        ["4", "rohith", None]]

columns = ['Employee_ID', 'Employee_NAME', 'Company_Name']
df = spark.createDataFrame(data, columns)
df.show()
```

```
+-----------+-------------+------------+
|Employee_ID|Employee_NAME|Company_Name|
+-----------+-------------+------------+
|          1|         NULL|   company 1|
|          2|       ojaswi|   company 2|
|       NULL|        bobby|   company 3|
|          1|       sravan|   company 1|
|          2|       ojaswi|        NULL|
|       NULL|       rohith|   company 2|
|          5|      gnanesh|   company 1|
|          2|         NULL|   company 2|
|          3|        bobby|   company 3|
|          4|       rohith|        NULL|
+-----------+-------------+------------+
```

Drop Rows with Null or Missing Values

1) Using dropna() :

The dropna() method is used to remove rows that contain any null (missing) values in the DataFrame. It’s handy for quick cleanup when you want to keep only fully complete records

```
df_clean = df.dropna()
df_clean.show()
```

```
+-----------+-------------+------------+
|Employee_ID|Employee_NAME|Company_Name|
+-----------+-------------+------------+
|          2|       ojaswi|   company 2|
|          1|       sravan|   company 1|
|          5|      gnanesh|   company 1|
|          3|        bobby|   company 3|
+-----------+-------------+------------+
```

2) Using isNotNull() :

The `isNotNull()` method is used to **filter out null values** in a DataFrame.It's commonly used with the `.filter()` or `.where()` clause to **keep only rows where a specific column is *not null***.

```
df.where(df.Employee_ID.isNotNull()).show()
```

```
+-----------+-------------+------------+
|Employee_ID|Employee_NAME|Company_Name|
+-----------+-------------+------------+
|          1|         NULL|   company 1|
|          2|       ojaswi|   company 2|
|          1|       sravan|   company 1|
|          2|       ojaswi|        NULL|
|          5|      gnanesh|   company 1|
|          2|         NULL|   company 2|
|          3|        bobby|   company 3|
|          4|       rohith|        NULL|
+-----------+-------------+------------+
```

this is using where()

```
df.filter(df["Employee_NAME"].isNotNull()).show()
```

```
+-----------+-------------+------------+
|Employee_ID|Employee_NAME|Company_Name|
+-----------+-------------+------------+
|          2|       ojaswi|   company 2|
|       NULL|        bobby|   company 3|
|          1|       sravan|   company 1|
|          2|       ojaswi|        NULL|
|       NULL|       rohith|   company 2|
|          5|      gnanesh|   company 1|
|          3|        bobby|   company 3|
|          4|       rohith|        NULL|
+-----------+-------------+------------+
```

this is for filter ()

Drop duplicate rows

Duplicate rows mean rows are the same among the dataframe, we are going to remove those rows by using dropDuplicates() function.

1) dropDuplicates()

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "sravan", "company 1"],
        ["2", "ojaswi", "company 2"],
        ["3", "bobby", "company 3"],
        ["1", "sravan", "company 1"],
        ["2", "ojaswi", "company 2"],
        ["6", "rohith", "company 2"],
        ["5", "gnanesh", "company 1"],
        ["2", "ojaswi", "company 2"],
        ["3", "bobby", "company 3"],
        ["4", "rohith", "company 2"]]

columns = ['ID', 'Employee_NAME', 'Company_Name']
df = spark.createDataFrame(data, columns)
df.dropDuplicates().show()
```

```
+---+-------------+------------+
| ID|Employee_NAME|Company_Name|
+---+-------------+------------+
|  2|       ojaswi|   company 2|
|  1|       sravan|   company 1|
|  3|        bobby|   company 3|
|  4|       rohith|   company 2|
|  6|       rohith|   company 2|
|  5|      gnanesh|   company 1|
+---+-------------+------------+
```

2) Drop duplicates based on the column name.

```
[ ] df.dropDuplicates(['Employee_Name']).show()
```

```
+---+-------------+------------+
| ID|Employee_NAME|Company_Name|
+---+-------------+------------+
|  3|        bobby|   company 3|
|  5|      gnanesh|   company 1|
|  2|       ojaswi|   company 2|
|  6|       rohith|   company 2|
|  1|       sravan|   company 1|
+---+-------------+------------+
```

Remove duplicate rows by using a distinct function

```
df.distinct().show()
```

```
+---+-------------+------------+
| ID|Employee_NAME|Company_Name|
+---+-------------+------------+
|  2|       ojaswi|   company 2|
|  1|       sravan|   company 1|
|  3|        bobby|   company 3|
|  4|       rohith|   company 2|
|  6|       rohith|   company 2|
|  5|      gnanesh|   company 1|
+---+-------------+------------+
```

Dropping Rows with Condition

1) Using Where condition

```
from pyspark.sql.functions import col
df.where(col("ID").cast("int")>4).show()
```

```
+---+-------------+------------+
| ID|Employee_NAME|Company_Name|
+---+-------------+------------+
|  6|       rohith|   company 2|
|  5|      gnanesh|   company 1|
+---+-------------+------------+
```

Not equal to

```
df.where(col("Company_Name") != 'company 1').show()
```

```
+---+-------------+------------+
| ID|Employee_NAME|Company_Name|
+---+-------------+------------+
|  2|       ojaswi|   company 2|
|  3|        bobby|   company 3|
|  2|       ojaswi|   company 2|
|  6|       rohith|   company 2|
|  2|       ojaswi|   company 2|
|  3|        bobby|   company 3|
|  4|       rohith|   company 2|
+---+-------------+------------+
```

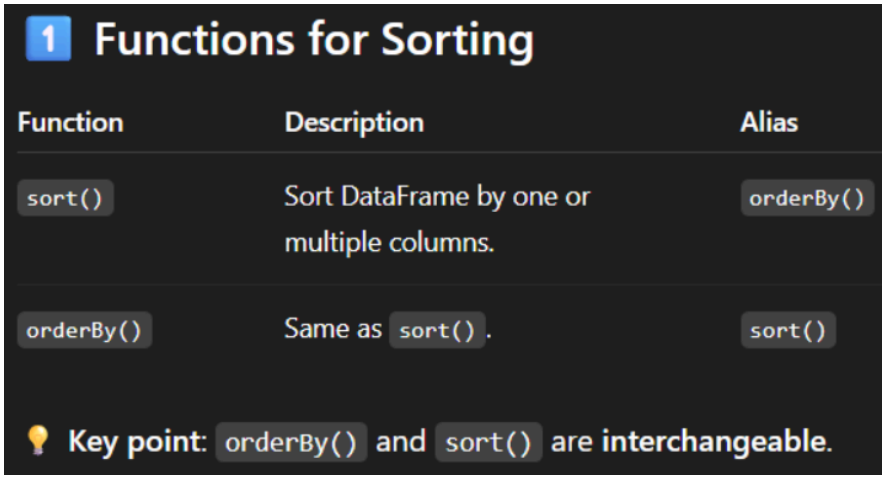
2) Using Filter() Condition

```
df.filter(df.ID != 4).show()
```

```
+---+-------------+------------+
| ID|Employee_NAME|Company_Name|
+---+-------------+------------+
|  1|       sravan|   company 1|
|  2|       ojaswi|   company 2|
|  3|        bobby|   company 3|
|  1|       sravan|   company 1|
|  2|       ojaswi|   company 2|
|  6|       rohith|   company 2|
|  5|      gnanesh|   company 1|
|  2|       ojaswi|   company 2|
|  3|        bobby|   company 3|
+---+-------------+------------+
```

Sorting

Sorting in PySpark allows you to rearrange rows of a DataFrame based on one or multiple columns in ascending or descending order.



```
from pyspark.sql import SparkSession
import pandas as pd
from google.colab import drive
spark = SparkSession.builder.master("local[1]").appName("Sorting Example").getOrCreate()
sc = spark.sparkContext
df = spark.read.csv('/content/college_student_placement_dataset (1).csv', header= True, inferSchema=True)
df.show()
```

```
+----------+---+---------------+-----+--------------------+--------------------+---------------------+--------------------+------------------+
|College_ID| IQ|Prev_Sem_Result| CGPA|Academic_Performance|Internship_Experience|Extra_Curricular_Score|Communication_Skills|Projects_Completed|
+----------+---+---------------+-----+--------------------+--------------------+---------------------+--------------------+------------------+
|   CLG0030|107|           6.61| 6.28|                   8|                  No|                    8|                   8|                 4|
|   CLG0061| 97|           5.52| 5.37|                   8|                  No|                    7|                   8|                 0|
|   CLG0036|109|           5.36| 5.83|                   9|                  No|                    3|                   1|                 1|
|   CLG0055|122|           5.47| 5.75|                   6|                 Yes|                    1|                   6|                 1|
|   CLG0004| 96|           7.91| 7.69|                   7|                  No|                    8|                  10|                 2|
|   CLG0015| 96|           5.26| 5.32|                   7|                  No|                    5|                   8|                 0|
|   CLG0071|123|           6.68| 6.58|                   5|                  No|                    7|                   8|                 2|
|   CLG0096|111|           8.77| 8.76|                   7|                  No|                    3|                   1|                 2|
|   CLG0097| 92|           6.47| 6.33|                   9|                  No|                    7|                   8|                 5|
|   CLG0057|108|           8.82|  8.6|                   4|                  No|                    5|                   9|                 1|
|   CLG0063| 93|           8.73|  8.9|                   2|                 Yes|                    5|                   6|                 0|
|   CLG0077| 93|           6.23| 6.51|                   8|                  No|                    5|                   7|                 4|
|   CLG0064|103|           8.64| 9.01|                   7|                 Yes|                    8|                   6|                 1|
|   CLG0017| 71|           8.74|  8.4|                   6|                  No|                    0|                   5|                 2|
|   CLG0053| 74|           6.99| 7.31|                   7|                  No|                    0|                   1|                 2|
|   CLG0040| 91|           6.05|  5.8|                   3|                  No|                    4|                   2|                 3|
|   CLG0070| 84|           7.61| 7.54|                   6|                  No|                    0|                  10|                 0|
|   CLG0050|104|           9.61|10.01|                  10|                 Yes|                    9|                   2|                 4|
|   CLG0068| 86|            8.2| 8.15|                   7|                  No|                    8|                   9|                 4|
|   CLG0015| 78|           5.86| 5.56|                   7|                 Yes|                    3|                   6|                 2|
+----------+---+---------------+-----+--------------------+--------------------+---------------------+--------------------+------------------+
only showing top 20 rows
```

a) Sort by Single Column (Ascending by default)

```
df.sort("CGPA").show()
```

```
+----------+---+---------------+----+--------------------+--------------------+---------------------+--------------------+------------------+---------+
|College_ID| IQ|Prev_Sem_Result|CGPA|Academic_Performance|Internship_Experience|Extra_Curricular_Score|Communication_Skills|Projects_Completed|Placement|
+----------+---+---------------+----+--------------------+--------------------+---------------------+--------------------+------------------+---------+
|   CLG0044|113|           5.03|4.54|                   6|                 Yes|                    9|                   5|                 4|       No|
|   CLG0017| 99|           5.01|4.56|                   5|                 Yes|                    0|                   9|                 4|       No|
|   CLG0025| 97|           5.06|4.57|                   6|                  No|                    7|                   5|                 2|       No|
|   CLG0005|123|           5.02|4.58|                   3|                 Yes|                    2|                   9|                 5|      Yes|
|   CLG0029| 98|           5.03|4.59|                  10|                  No|                   10|                   4|                 0|       No|
|   CLG0090| 98|           5.06|4.59|                   8|                  No|                    3|                   3|                 0|       No|
|   CLG0067|130|           5.04|4.59|                   3|                 Yes|                    1|                  10|                 4|      Yes|
|   CLG0011|110|           5.04| 4.6|                   1|                  No|                    0|                   4|                 1|       No|
|   CLG0050| 92|           5.09| 4.6|                   8|                  No|                    4|                   3|                 2|       No|
|   CLG0023|101|           5.02|4.61|                   5|                  No|                    7|                   3|                 4|       No|
|   CLG0078|103|           5.01|4.61|                   2|                 Yes|                    0|                  10|                 4|       No|
|   CLG0085|109|           5.03|4.61|                   4|                  No|                   10|                  10|                 5|       No|
|   CLG0009| 93|           5.05|4.61|                   2|                  No|                    7|                   7|                 1|       No|
|   CLG0069|105|            5.0|4.61|                   8|                  No|                    7|                   3|                 1|       No|
|   CLG0010|103|           5.05|4.61|                   3|                 Yes|                    6|                   2|                 4|       No|
|   CLG0026| 81|           5.07|4.62|                  10|                  No|                    7|                   9|                 5|       No|
|   CLG0076| 73|            5.0|4.62|                   9|                  No|                    1|                   6|                 4|       No|
|   CLG0066| 95|           5.08|4.63|                   3|                 Yes|                    7|                   2|                 4|       No|
|   CLG0004|104|           5.05|4.64|                  10|                 Yes|                    2|                   5|                 1|       No|
|   CLG0053| 77|           5.08|4.65|                   4|                 Yes|                    3|                   1|                 3|       No|
+----------+---+---------------+----+--------------------+--------------------+---------------------+--------------------+------------------+---------+
only showing top 20 rows
```

a) Sort by Multiple Column

```
from pyspark.sql.functions import col
df.sort("Prev_Sem_Result","CGPA").show()
```

```
+----------+---+--------------+----+------------------+------------------+-------------------+-------------------+------------------+---------+
|College_ID| IQ|Prev_Sem_Result|CGPA|Academic_Performance|Internship_Experience|Extra_Curricular_Score|Communication_Skills|Projects_Completed|Placement|
+----------+---+--------------+----+------------------+------------------+-------------------+-------------------+------------------+---------+
|   CLG0069|105|           5.0|4.61|                 8|                No|                  7|                  3|                 1|       No|
|   CLG0076| 73|           5.0|4.62|                 9|                No|                  1|                  6|                 4|       No|
|   CLG0004| 88|           5.0|4.69|                 3|                No|                  0|                  2|                 0|       No|
|   CLG0038|108|           5.0| 4.7|                 6|                No|                  8|                  7|                 3|       No|
|   CLG0069| 86|           5.0|4.72|                 7|               Yes|                 10|                  6|                 0|       No|
|   CLG0088| 90|           5.0|4.81|                 7|                No|                  3|                  6|                 4|       No|
|   CLG0088| 73|           5.0|4.86|                 8|                No|                  7|                  4|                 2|       No|
|   CLG0097|117|           5.0|4.87|                 3|               Yes|                  8|                  6|                 1|       No|
|   CLG0046| 92|           5.0|4.95|                 4|               Yes|                  1|                  5|                 5|       No|
|   CLG0036|118|           5.0|5.41|                 2|                No|                  9|                  5|                 1|       No|
|   CLG0017| 99|          5.01|4.56|                 5|               Yes|                  0|                  9|                 4|       No|
|   CLG0078|103|          5.01|4.61|                 2|               Yes|                  0|                 10|                 4|       No|
|   CLG0068|116|          5.01|4.66|                 9|                No|                  4|                  6|                 1|       No|
|   CLG0055| 99|          5.01| 4.8|                 8|                No|                  3|                  1|                 1|       No|
|   CLG0085|114|          5.01|4.83|                 5|                No|                  2|                  5|                 0|       No|
|   CLG0061| 82|          5.01|4.92|                10|                No|                  5|                  8|                 3|       No|
|   CLG0005|116|          5.01|4.94|                 8|                No|                  3|                  8|                 1|       No|
|   CLG0071|101|          5.01|4.94|                 1|                No|                  2|                  9|                 5|       No|
|   CLG0041| 99|          5.01|4.97|                 6|               Yes|                  7|                  7|                 4|       No|
|   CLG0012| 81|          5.01|4.97|                10|                No|                  8|                 10|                 2|       No|
+----------+---+--------------+----+------------------+------------------+-------------------+-------------------+------------------+---------+
only showing top 20 rows
```

c) Ascending & Descending Combination

```
from pyspark.sql.functions import col
df.sort(col("Prev_Sem_Result").asc(), col("CGPA").desc()).show()
```

```
+----------+---+--------------+----+------------------+------------------+-------------------+-------------------+------------------+---------+
|College_ID| IQ|Prev_Sem_Result|CGPA|Academic_Performance|Internship_Experience|Extra_Curricular_Score|Communication_Skills|Projects_Completed|Placement|
+----------+---+--------------+----+------------------+------------------+-------------------+-------------------+------------------+---------+
|   CLG0036|118|           5.0|5.41|                 2|                No|                  9|                  5|                 1|       No|
|   CLG0046| 92|           5.0|4.95|                 4|               Yes|                  1|                  5|                 5|       No|
|   CLG0097|117|           5.0|4.87|                 3|               Yes|                  8|                  6|                 1|       No|
|   CLG0088| 73|           5.0|4.86|                 8|                No|                  7|                  4|                 2|       No|
|   CLG0088| 90|           5.0|4.81|                 7|                No|                  3|                  6|                 4|       No|
|   CLG0069| 86|           5.0|4.72|                 7|               Yes|                 10|                  6|                 0|       No|
|   CLG0038|108|           5.0| 4.7|                 6|                No|                  8|                  7|                 3|       No|
|   CLG0004| 88|           5.0|4.69|                 3|                No|                  0|                  2|                 0|       No|
|   CLG0076| 73|           5.0|4.62|                 9|                No|                  1|                  6|                 4|       No|
|   CLG0069|105|           5.0|4.61|                 8|                No|                  7|                  3|                 1|       No|
|   CLG0045|112|          5.01|5.41|                 7|                No|                 10|                  9|                 1|       No|
|   CLG0033| 92|          5.01|5.35|                 2|                No|                  4|                  4|                 3|       No|
|   CLG0085|113|          5.01|5.28|                 8|               Yes|                  2|                  5|                 4|       No|
|   CLG0034| 88|          5.01|5.04|                 3|                No|                  9|                  7|                 0|       No|
|   CLG0036| 90|          5.01|4.98|                10|                No|                  4|                  2|                 2|       No|
|   CLG0041| 99|          5.01|4.97|                 6|               Yes|                  7|                  7|                 4|       No|
|   CLG0012| 81|          5.01|4.97|                10|                No|                  8|                 10|                 2|       No|
|   CLG0071|101|          5.01|4.94|                 1|                No|                  2|                  9|                 5|       No|
|   CLG0005|116|          5.01|4.94|                 8|                No|                  3|                  8|                 1|       No|
|   CLG0061| 82|          5.01|4.92|                10|                No|                  5|                  8|                 3|       No|
+----------+---+--------------+----+------------------+------------------+-------------------+-------------------+------------------+---------+
only showing top 20 rows
```

🔍 Problem: Sorting with NULLs

When you sort a DataFrame, PySpark needs to decide **where NULL values should go**. By default:

Order	NULL Position (Default)
Ascending	Top
Descending	Bottom

But sometimes, you want to **override this behavior**.

✅ Solution: Use These Functions for Explicit NULL Handling

PySpark provides **4 special functions** in `pyspark.sql.functions` to customize NULL placement when sorting:

1️⃣ `asc_nulls_first(column)`

- **Ascending sort**
- **NULLs come first**
- ✅ This is the **default behavior**

```
from pyspark.sql.functions import asc_nulls_first
df.orderBy(asc_nulls_first("score")).show()
```

Result Example:

```
+------+
|score |
+------+
|  null|
|   1.2|
|   3.4|
|   4.5|
+------+
```

2️⃣ `asc_nulls_last(column)`

- **Ascending sort**
- **NULLs go to the bottom**

```
from pyspark.sql.functions import asc_nulls_last
```

```
df.orderBy(asc_nulls_last("score")).show()
```

Result Example:

```
+------+
|score |
+------+
|   1.2|
|   3.4|
|   4.5|
|  null|
+------+
```

3️⃣ `desc_nulls_first(column)`

- **Descending sort**
- **NULLs come first**

```
from pyspark.sql.functions import desc_nulls_first
df.orderBy(desc_nulls_first("score")).show()
```

Result Example:

```
+------+
|score |
+------+
|  null|
|   4.5|
|   3.4|
|   1.2|
+------+
```

4️⃣ `desc_nulls_last(column)`

- **Descending sort**
- **NULLs go to the bottom**
- ✅ This is the **default behavior for descending**

```
from pyspark.sql.functions import desc_nulls_last
df.orderBy(desc_nulls_last("score")).show()
```

Result Example:

```
+------+
|score |
+------+
|   4.5|
|   3.4|
|   1.2|
|  null|
+------+
```

```
from pyspark.sql import SparkSession
import pandas as pd
from google.colab import drive
spark = SparkSession.builder.master("local[1]").appName("Sorting Example").getOrCreate()
sc = spark.sparkContext
df = spark.read.csv('/content/LoanData (1).csv', header= True, inferSchema=True)
df.show()
```

```
+--------+------+-------+----------+------------+-------------+---------------+----------------+----------+---------------+-------------+-------------+-----------+
| Loan_ID|Gender|Married|Dependents|   Education|Self_Employed|ApplicantIncome|CoapplicantIncome|LoanAmount|Loan_Amount_Term|Credit_History|Property_Area|Loan_Status|
+--------+------+-------+----------+------------+-------------+---------------+----------------+----------+---------------+-------------+-------------+-----------+
|LP001002|  Male|     No|         0|    Graduate|           No|           5849|             0.0|      NULL|            360|            1|        Urban|          Y|
|LP001003|  Male|    Yes|         1|    Graduate|           No|           4583|          1508.0|       128|            360|            1|        Rural|          N|
|LP001005|  Male|    Yes|         0|    Graduate|          Yes|           3000|             0.0|        66|            360|            1|        Urban|          Y|
|LP001006|  Male|    Yes|         0|Not Graduate|           No|           2583|          2358.0|       120|            360|            1|        Urban|          Y|
|LP001008|  Male|     No|         0|    Graduate|           No|           6000|             0.0|       141|            360|            1|        Urban|          Y|
|LP001011|  Male|    Yes|         2|    Graduate|          Yes|           5417|          4196.0|       267|            360|            1|        Urban|          Y|
|LP001013|  Male|    Yes|         0|Not Graduate|           No|           2333|          1516.0|        95|            360|            1|        Urban|          Y|
|LP001014|  Male|    Yes|        3+|    Graduate|           No|           3036|          2504.0|       158|            360|            0|    Semiurban|          N|
|LP001018|  Male|    Yes|         2|    Graduate|           No|           4006|          1526.0|       168|            360|            1|        Urban|          Y|
|LP001020|  Male|    Yes|         1|    Graduate|           No|          12841|         10968.0|       349|            360|            1|    Semiurban|          N|
|LP001024|  Male|    Yes|         2|    Graduate|           No|           3200|           700.0|        70|            360|            1|        Urban|          Y|
|LP001027|  Male|    Yes|         2|    Graduate|         NULL|           2500|          1840.0|       109|            360|            1|        Urban|          Y|
|LP001028|  Male|    Yes|         2|    Graduate|           No|           3073|          8106.0|       200|            360|            1|        Urban|          Y|
|LP001029|  Male|     No|         0|    Graduate|           No|           1853|          2840.0|       114|            360|            1|        Rural|          N|
|LP001030|  Male|    Yes|         2|    Graduate|           No|           1299|          1086.0|        17|            120|            1|        Urban|          Y|
|LP001032|  Male|     No|         0|    Graduate|           No|           4950|             0.0|       125|            360|            1|        Urban|          Y|
|LP001034|  Male|     No|         1|Not Graduate|           No|           3596|             0.0|       100|            240|         NULL|        Urban|          Y|
|LP001036|Female|     No|         0|    Graduate|           No|           3510|             0.0|        76|            360|            0|        Urban|          N|
|LP001038|  Male|    Yes|         0|Not Graduate|           No|           4887|             0.0|       133|            360|            1|        Rural|          N|
|LP001041|  Male|    Yes|         0|    Graduate|         NULL|           2600|          3500.0|       115|           NULL|            1|        Urban|          Y|
+--------+------+-------+----------+------------+-------------+---------------+----------------+----------+---------------+-------------+-------------+-----------+
only showing top 20 rows
```

```
from pyspark.sql.functions import asc_nulls_first, asc_nulls_last, desc_nulls_first, desc_nulls_last
df.orderBy(asc_nulls_first("Credit_History")).show() # Ascending, nulls at top
df.orderBy(desc_nulls_last("Credit_History")).show() # Descending, nulls at bottom
```

```
+--------+------+-------+----------+------------+-------------+---------------+----------------+----------+---------------+-------------+-------------+-----------+
| Loan_ID|Gender|Married|Dependents|   Education|Self_Employed|ApplicantIncome|CoapplicantIncome|LoanAmount|Loan_Amount_Term|Credit_History|Property_Area|Loan_Status|
+--------+------+-------+----------+------------+-------------+---------------+----------------+----------+---------------+-------------+-------------+-----------+
|LP001405|  Male|    Yes|         1|    Graduate|           No|           2214|          1398.0|        85|            360|         NULL|        Urban|          Y|
|LP002178|  Male|    Yes|         0|    Graduate|           No|           3013|          3033.0|        95|            300|         NULL|        Urban|          Y|
|LP001443|Female|     No|         0|    Graduate|           No|           3692|             0.0|        93|            360|         NULL|        Rural|          Y|
|LP001052|  Male|    Yes|         1|    Graduate|         NULL|           3717|          2925.0|       151|            360|         NULL|    Semiurban|          N|
|LP001465|  Male|    Yes|         0|    Graduate|           No|           6080|          2569.0|       182|            360|         NULL|        Rural|          N|
|LP001123|  Male|    Yes|         0|    Graduate|           No|           2400|             0.0|        75|            360|         NULL|        Urban|          Y|
|LP001469|  Male|     No|         0|    Graduate|          Yes|          20166|             0.0|       650|            480|         NULL|        Urban|          Y|
|LP001273|  Male|    Yes|         0|    Graduate|           No|           6000|          2250.0|       265|            360|         NULL|    Semiurban|          N|
|LP001541|  Male|    Yes|         1|    Graduate|           No|           6000|             0.0|       160|            360|         NULL|        Rural|          Y|
|LP001326|  Male|     No|         0|    Graduate|         NULL|           6782|             0.0|      NULL|            360|         NULL|        Urban|          N|
|LP001634|  Male|     No|         0|    Graduate|           No|           1916|          5063.0|        67|            360|         NULL|        Rural|          N|
|LP001864|  Male|    Yes|        3+|Not Graduate|           No|           4931|             0.0|       128|            360|         NULL|    Semiurban|          N|
|LP002137|  Male|    Yes|         0|    Graduate|           No|           6333|          4583.0|       259|            360|         NULL|    Semiurban|          Y|
|LP001865|  Male|    Yes|         1|    Graduate|           No|           6083|          4250.0|       330|            360|         NULL|        Urban|          Y|
|LP001091|  Male|    Yes|         1|    Graduate|         NULL|           4166|          3369.0|       201|            360|         NULL|        Urban|          N|
|LP001908|Female|    Yes|         0|Not Graduate|           No|           4100|             0.0|       124|            360|         NULL|        Rural|          Y|
|LP001280|  Male|    Yes|         2|Not Graduate|           No|           3333|          2000.0|        99|            360|         NULL|    Semiurban|          Y|
|LP001998|  Male|    Yes|         2|Not Graduate|           No|           7667|             0.0|       185|            360|         NULL|        Rural|          Y|
|LP001671|Female|    Yes|         0|    Graduate|           No|           3416|          2816.0|       113|            360|         NULL|    Semiurban|          Y|
|LP002008|  Male|    Yes|         2|    Graduate|          Yes|           5746|             0.0|       144|             84|         NULL|        Rural|          Y|
+--------+------+-------+----------+------------+-------------+---------------+----------------+----------+---------------+-------------+-------------+-----------+
only showing top 20 rows
```

```
+--------+------+-------+----------+------------+-------------+--------------+-----------------+----------+---------------+--------------+-------------+-----------+
| Loan_ID|Gender|Married|Dependents|   Education|Self_Employed|ApplicantIncome|CoapplicantIncome|LoanAmount|Loan_Amount_Term|Credit_History|Property_Area|Loan_Status|
+--------+------+-------+----------+------------+-------------+--------------+-----------------+----------+---------------+--------------+-------------+-----------+
|LP001046|  Male|    Yes|         1|    Graduate|           No|          5955|           5625.0|       315|            360|             1|        Urban|          Y|
|LP001136|  Male|    Yes|         0|Not Graduate|          Yes|          4695|              0.0|        96|           NULL|             1|        Urban|          Y|
|LP001003|  Male|    Yes|         1|    Graduate|           No|          4583|           1508.0|       128|            360|             1|        Rural|          N|
|LP001066|  Male|    Yes|         0|    Graduate|          Yes|          9560|              0.0|       191|            360|             1|    Semiurban|          Y|
|LP001005|  Male|    Yes|         0|    Graduate|          Yes|          3000|              0.0|        66|            360|             1|        Urban|          Y|
|LP001068|  Male|    Yes|         0|    Graduate|           No|          2799|           2253.0|       122|            360|             1|    Semiurban|          Y|
|LP001008|  Male|     No|         0|    Graduate|           No|          6000|              0.0|       141|            360|             1|        Urban|          Y|
|LP001073|  Male|    Yes|         2|Not Graduate|           No|          4226|           1040.0|       110|            360|             1|        Urban|          Y|
|LP001013|  Male|    Yes|         0|Not Graduate|           No|          2333|           1516.0|        95|            360|             1|        Urban|          Y|
|LP001086|  Male|     No|         0|Not Graduate|           No|          1442|              0.0|        35|            360|             1|        Urban|          N|
|LP001020|  Male|    Yes|         1|    Graduate|           No|         12841|          10968.0|       349|            360|             1|    Semiurban|          N|
|LP001087|Female|     No|         2|    Graduate|         NULL|          3750|           2083.0|       120|            360|             1|    Semiurban|          Y|
|LP001027|  Male|    Yes|         2|    Graduate|         NULL|          2500|           1840.0|       109|            360|             1|        Urban|          Y|
|LP001095|  Male|     No|         0|    Graduate|           No|          3167|              0.0|        74|            360|             1|        Urban|          N|
|LP001029|  Male|     No|         0|    Graduate|           No|          1853|           2840.0|       114|            360|             1|        Rural|          N|
|LP001097|  Male|     No|         1|    Graduate|          Yes|          4692|              0.0|       106|            360|             1|        Rural|          N|
|LP001032|  Male|     No|         0|    Graduate|           No|          4950|              0.0|       125|            360|             1|        Urban|          Y|
|LP001098|  Male|    Yes|         0|    Graduate|           No|          3500|           1667.0|       114|            360|             1|    Semiurban|          Y|
|LP001041|  Male|    Yes|         0|    Graduate|         NULL|          2600|           3500.0|       115|           NULL|             1|        Urban|          Y|
|LP001100|  Male|     No|        3+|    Graduate|           No|         12500|           3000.0|       320|            360|             1|        Rural|          N|
+--------+------+-------+----------+------------+-------------+--------------+-----------------+----------+---------------+--------------+-------------+-----------+
only showing top 20 rows
```

Interview/Discussion Points

- sort() and orderBy() are aliases.
- Sorting is stable (order is preserved for equal values in Spark ≥ 3.2).
- For different sorting orders on different columns, use ascending=[True, False].
- Use .asc(), .desc() for explicit ordering.
- Use asc_nulls_first() or desc_nulls_last() for null-handling.
- SQL equivalent uses ORDER BY.
- Sorting is an expensive operation because it requires shuffling data across partitions.

Aggregate Functions

```
from pyspark.sql import SparkSession
import pandas as pd
from google.colab import drive
spark = SparkSession.builder.master("local[1]").appName("Sorting Example").getOrCreate()
sc = spark.sparkContext
df = spark.read.csv('/content/salary.csv', header= True, inferSchema=True)
df.show(30, truncate = False)
```

```
+------+---+----+----------+--------+
|name  |id |age |department|salary  |
+------+---+----+----------+--------+
|user1 |1  |25.0|Jr manager|98000.0 |
|user2 |2  |30.0|sr manager|100000.0|
|user3 |6  |35.0|sr manager|100000.0|
|user4 |4  |32.0|head      |70000.0 |
|user5 |1  |45.0|Jr manager|60000.0 |
|user6 |6  |47.0|head2     |45000.0 |
|user7 |5  |21.0|worker    |25000.0 |
|user8 |1  |22.0|Jr manager|50000.0 |
|user9 |10 |54.0|lead      |45000.0 |
|user10|59 |52.0|lead2     |50000.0 |
|user11|6  |25.0|head2     |50000.0 |
|user12|2  |27.0|sr manager|70000.0 |
|user13|59 |54.0|lead2     |45000.0 |
|user14|2  |25.0|sr manager|70000.0 |
|user15|1  |32.0|Jr manager|50000.0 |
|user16|3  |37.0|worker    |25000.0 |
|user17|74 |63.0|Manager   |68000.0 |
|user18|7  |25.0|head      |45000.0 |
|user19|10 |32.0|lvl2 head |52000.0 |
|user20|10 |32.0|lvl2 head |52000.0 |
|user21|12 |NULL|NULL      |NULL    |
```

- count(): This will return the count of rows for each group.
- mean(): This will return the mean of values for each group.
- min(): This will return the minimum of values for each group.
- sum(): This will return the total values for each group.
- avg(): This will return the average for values for each group.
- groupby() - This function groups the data by one or more columns and then applies an aggregate function to each group.

`first('salary')`	Returns the **first value** in the `salary` column (by internal order)	See the earliest entry
`last('salary')`	Returns the **last value** in the `salary` column (by internal order)	See the latest entry
`kurtosis('salary')`	Measures how **heavy or light the tails** of the distribution are compared to a normal distribution	Tells you about outliers/extremes
`skewness('salary')`	Measures how **asymmetrical** the distribution is	See if data is biased toward high or low values
`stddev('salary')` or `stddev_samp('salary')`	Standard deviation using **sample formula**	Shows how spread out the salaries are
`stddev_pop('salary')`	Standard deviation using **population formula**	Use when working with the entire dataset
`sumDistinct('salary')`	Adds only **distinct salary values**	Avoids counting duplicates
`variance('salary')` or `var_samp('salary')`	Variance using **sample formula**	Square of standard deviation (sample)
`var_pop('salary')`	Variance using **population formula**	Square of population std deviation

Example 1: Multiple aggregations on DEPT column with salary column

```
from pyspark.sql import functions
df.groupby('department').agg(functions.min('salary'),
                      functions.max('salary'),
                      functions.sum('salary'),
                      functions.mean('salary'),
                      functions.count('salary'),
                      functions.avg('salary'),
                      functions.first('salary'),
                      functions.last('salary'),
                      functions.kurtosis('salary'),
                      functions.skewness('salary'),
                      functions.stddev('salary'),
                      functions.stddev_pop('salary'),
                      functions.stddev_samp('salary'),
                      functions.sumDistinct('salary'),
                      functions.variance('salary'),
                      functions.var_samp('salary'),
                      functions.var_pop('salary'),).show()
```

```
/usr/local/lib/python3.11/dist-packages/pyspark/sql/functions.py:988: FutureWarning: Deprecated in 3.2, use sum_distinct instead.
  warnings.warn("Deprecated in 3.2, use sum_distinct instead.", FutureWarning)
+----------+-----------+-----------+-----------+------------------+------------+------------------+------------+------------+------------------+--------------------+--------------------+
|department|min(salary)|max(salary)|sum(salary)|       avg(salary)|count(salary)|       avg(salary)|first(salary)|last(salary)|   kurtosis(salary)|   skewness(salary)|     stddev(salary)|
+----------+-----------+-----------+-----------+------------------+------------+------------------+------------+------------+------------------+--------------------+--------------------+
|      NULL|    12000.0|   140000.0|   164000.0|54666.666666666664|           3|54666.666666666664|     12000.0|    140000.0|              -1.5|0.7071067811865475| 73900.8344562720...|
|Jr manager|    50000.0|    98000.0|   258000.0|           64500.0|           4|           64500.0|     60000.0|     50000.0|-0.7924152946680865|1.0165248108608964|22825.424421026655...|
|      head|    45000.0|    70000.0|   165000.0|           55000.0|           3|           55000.0|     45000.0|     50000.0|              -1.5|0.5951700641394974|13228.756555322952...|
|sr manager|    70000.0|   100000.0|   340000.0|           85000.0|           4|           85000.0|     70000.0|    100000.0|              -2.0|               0.0|17320.508075688773...|
|     head2|    45000.0|    50000.0|    95000.0|           47500.0|           2|           47500.0|     45000.0|     50000.0|              -2.0|               0.0|3535.5339059327375...|
| lvl2 head|    52000.0|    52000.0|   104000.0|           52000.0|           2|           52000.0|     52000.0|     52000.0|              NULL|              NULL|                0.0|
|      lead|    45000.0|    45000.0|    45000.0|           45000.0|           1|           45000.0|     45000.0|     45000.0|              NULL|              NULL|                NULL|
|   Manager|    68000.0|    68000.0|    68000.0|           68000.0|           1|           68000.0|     68000.0|     68000.0|              NULL|              NULL|                NULL|
+----------+-----------+-----------+-----------+------------------+------------+------------------+------------+------------+------------------+--------------------+--------------------+
```

Example 2: Multiple aggregation in grouping dept and name column

```
from pyspark.sql import functions
df.groupby('department', 'age').agg(functions.min('salary'),
                      functions.max('salary'),
                      functions.sum('salary'),
                      functions.mean('salary'),
                      functions.count('salary'),
                      functions.avg('salary')).show()
```

```
+----------+----+-----------+-----------+-----------+-----------+-------------+-----------+
|department| age|min(salary)|max(salary)|sum(salary)|avg(salary)|count(salary)|avg(salary)|
+----------+----+-----------+-----------+-----------+-----------+-------------+-----------+
|      lead|54.0|    45000.0|    45000.0|    45000.0|    45000.0|            1|    45000.0|
|    worker|37.0|    25000.0|    25000.0|    25000.0|    25000.0|            1|    25000.0|
|     head2|25.0|    50000.0|    50000.0|    50000.0|    50000.0|            1|    50000.0|
|    worker|21.0|    25000.0|    25000.0|    25000.0|    25000.0|            1|    25000.0|
|      NULL|NULL|    12000.0|   140000.0|   152000.0|    76000.0|            2|    76000.0|
|   Manager|63.0|    68000.0|    68000.0|    68000.0|    68000.0|            1|    68000.0|
|sr manager|30.0|   100000.0|   100000.0|   100000.0|   100000.0|            1|   100000.0|
|Jr manager|45.0|    60000.0|    60000.0|    60000.0|    60000.0|            1|    60000.0|
|sr manager|27.0|    70000.0|    70000.0|    70000.0|    70000.0|            1|    70000.0|
|     lead2|52.0|    50000.0|    50000.0|    50000.0|    50000.0|            1|    50000.0|
|Jr manager|32.0|    50000.0|    50000.0|    50000.0|    50000.0|            1|    50000.0|
| lvl2 head|32.0|    52000.0|    52000.0|   104000.0|    52000.0|            2|    52000.0|
|Jr manager|22.0|    50000.0|    50000.0|    50000.0|    50000.0|            1|    50000.0|
|sr manager|25.0|    70000.0|    70000.0|    70000.0|    70000.0|            1|    70000.0|
|     lead2|54.0|    45000.0|    45000.0|    45000.0|    45000.0|            1|    45000.0|
|sr manager|35.0|   100000.0|   100000.0|   100000.0|   100000.0|            1|   100000.0|
|Jr manager|25.0|    98000.0|    98000.0|    98000.0|    98000.0|            1|    98000.0|
|     head2|47.0|    45000.0|    45000.0|    45000.0|    45000.0|            1|    45000.0|
|      head|25.0|    45000.0|    45000.0|    45000.0|    45000.0|            1|    45000.0|
|      NULL|24.0|    12000.0|    12000.0|    12000.0|    12000.0|            1|    12000.0|
+----------+----+-----------+-----------+-----------+-----------+-------------+-----------+
only showing top 20 rows
```

Group By

The `groupBy()` function in PySpark is used to **group data based on one or more columns**, and then perform **aggregate operations**

```
from pyspark.sql import SparkSession
from pyspark.sql import functions
spark = SparkSession.builder \
                    .appName('Group By examples') \
                    .getOrCreate()
data = [("James","Sales","NY",90000,34,10000),
    ("Michael","Sales","NV",86000,56,20000),
    ("Robert","Sales","CA",81000,30,23000),
    ("Maria","Finance","CA",90000,24,23000),
    ("Raman","Finance","DE",99000,40,24000),
    ("Scott","Finance","NY",83000,36,19000),
    ("Jen","Finance","NY",79000,53,15000),
    ("Jeff","Marketing","NV",80000,25,18000),
    ("Kumar","Marketing","NJ",91000,50,21000)]
schema = ['employee_name', "department", "state", "salary", "age", "bonus"]
df = spark.createDataFrame(data = data, schema = schema)
df.printSchema()
df.show(truncate = False)
```

```
root
 |-- employee_name: string (nullable = true)
 |-- department: string (nullable = true)
 |-- state: string (nullable = true)
 |-- salary: long (nullable = true)
 |-- age: long (nullable = true)
 |-- bonus: long (nullable = true)

+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NV   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Raman        |Finance   |DE   |99000 |40 |24000|
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |NV   |80000 |25 |18000|
|Kumar        |Marketing |NJ   |91000 |50 |21000|
+-------------+----------+-----+------+---+-----+
```

groupBy() with aggregation

```
df.groupBy("state").sum("salary").show()
```

```
+-----+-----------+
|state|sum(salary)|
+-----+-----------+
|   NV|     166000|
|   CA|     171000|
|   NY|     252000|
|   NJ|      91000|
|   DE|      99000|
+-----+-----------+
```

.agg() — Multiple or custom aggregations : Use .agg() to apply one or more aggregations

and give result column custom names using .alias().

Syntax :

```
df.groupBy("column_name").agg(
   function1(col("target_column")).alias("custom_name1"),
   function2(col("target_column")).alias("custom_name2")
)
```

```
dfgroup = df.groupBy("state").agg(functions.sum("salary").alias("sum_salary"))
dfgroup.show()
```

```
+-----+----------+
|state|sum_salary|
+-----+----------+
|   NV|    166000|
|   CA|    171000|
|   NY|    252000|
|   NJ|     91000|
|   DE|     99000|
+-----+----------+
```

.filter() — Filter rows after aggregation

Filters results after aggregation, based on the aggregated value

```
df_filter = dfgroup.filter(dfgroup.sum_salary >100000)
df_filter.show()
```

```
+-----+----------+
|state|sum_salary|
+-----+----------+
|   NV|    166000|
|   CA|    171000|
|   NY|    252000|
+-----+----------+
```

.sort() — Sorting data

Sort results in ascending or descending order.

```
df_filter.sort("sum_salary").show()
df_filter.sort(functions.desc("sum_salary")).show()
```

```
+-----+----------+
|state|sum_salary|
+-----+----------+
|   NV|    166000|
|   CA|    171000|
|   NY|    252000|
+-----+----------+

+-----+----------+
|state|sum_salary|
+-----+----------+
|   NY|    252000|
|   CA|    171000|
|   NV|    166000|
+-----+----------+
```

Joins

```
d1 = [["1", "sravan", "company 1"],     #first dataframe
        ["2", "ojaswi", "company 1"],
        ["3", "rohith", "company 2"],
        ["4", "sridevi", "company 1"]]
cols = ['id', 'Name', 'Company']
df1 = spark.createDataFrame(d1, cols)
df1.show()
d2 = [("1", "45000", "IT"),             #second dataframe
      ("2", "145000", "Manager"),
      ("6", "45000", "HR"),
      ("5", "34000", "Sales")]
cols2 = ['id', 'salary', 'department']
df2 = spark.createDataFrame(d2, cols2)
df2.show()
```

```
+---+-------+---------+
| id|   Name|  Company|
+---+-------+---------+
|  1| sravan|company 1|
|  2| ojaswi|company 1|
|  3| rohith|company 2|
|  4|sridevi|company 1|
+---+-------+---------+

+---+------+----------+
| id|salary|department|
+---+------+----------+
|  1| 45000|        IT|
|  2|145000|   Manager|
|  6| 45000|        HR|
|  5| 34000|     Sales|
+---+------+----------+
```

Inner join

This will join the two PySpark dataframes on key columns, which are common in both dataframes.

Syntax

`dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"inner")`

```
df1.join(df2, df1.id == df2.id, "inner").show()
```

```
+---+------+---------+---+------+----------+
| id|  Name|  Company| id|salary|department|
+---+------+---------+---+------+----------+
|  1|sravan|company 1|  1| 45000|        IT|
|  2|ojaswi|company 1|  2|145000|   Manager|
+---+------+---------+---+------+----------+
```

Full Outer Join

A FULL OUTER JOIN combines all the data from both tables (DataFrames).If there is a match between the two DataFrames, it joins the rows.If there's no match, it still keeps the row and fills the missing values with `null`., we can perform this join in three ways.They are just different words for the same join , does same work just name is different

Syntax

outer: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"outer")

full: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"full")

fullouter: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"fullouter")

```
df1.join(df2, df1.id == df2.id, "outer").show()
```

```
+----+-------+---------+----+------+----------+
|  id|   Name|  Company|  id|salary|department|
+----+-------+---------+----+------+----------+
|   1| sravan|company 1|   1| 45000|        IT|
|   2| ojaswi|company 1|   2|145000|   Manager|
|   3| rohith|company 2|NULL|  NULL|      NULL|
|   4|sridevi|company 1|NULL|  NULL|      NULL|
|NULL|   NULL|     NULL|   5| 34000|     Sales|
|NULL|   NULL|     NULL|   6| 45000|        HR|
+----+-------+---------+----+------+----------+
```

```
df1.join(df2, df1.id == df2.id, "full").show()
```

```
+----+-------+---------+----+------+----------+
|  id|   Name|  Company|  id|salary|department|
+----+-------+---------+----+------+----------+
|   1| sravan|company 1|   1| 45000|        IT|
|   2| ojaswi|company 1|   2|145000|   Manager|
|   3| rohith|company 2|NULL|  NULL|      NULL|
|   4|sridevi|company 1|NULL|  NULL|      NULL|
|NULL|   NULL|     NULL|   5| 34000|     Sales|
|NULL|   NULL|     NULL|   6| 45000|        HR|
+----+-------+---------+----+------+----------+
```

```
df1.join(df2, df1.id == df2.id, "fullouter").show()
```

```
+----+-------+---------+----+------+----------+
|  id|   Name|  Company|  id|salary|department|
+----+-------+---------+----+------+----------+
|   1| sravan|company 1|   1| 45000|        IT|
|   2| ojaswi|company 1|   2|145000|   Manager|
|   3| rohith|company 2|NULL|  NULL|      NULL|
|   4|sridevi|company 1|NULL|  NULL|      NULL|
|NULL|   NULL|     NULL|   5| 34000|     Sales|
|NULL|   NULL|     NULL|   6| 45000|        HR|
+----+-------+---------+----+------+----------+
```

Left Join

- Returns all rows from the left DataFrame and the matched rows from the right DataFrame.
- If no match is found in the right DataFrame, NULL values will be returned for the right DataFrame columns.

Syntax

df1.join(df2, df1.ID == df2.ID, "left")

df1.join(df2, df1.ID == df2.ID, "leftouter") # same as left

```
df1.join(df2, df1.id == df2.id, "left").show()
```

```
+---+-------+---------+----+------+----------+
| id|   Name|  Company|  id|salary|department|
+---+-------+---------+----+------+----------+
|  1| sravan|company 1|   1| 45000|        IT|
|  2| ojaswi|company 1|   2|145000|   Manager|
|  3| rohith|company 2|NULL|  NULL|      NULL|
|  4|sridevi|company 1|NULL|  NULL|      NULL|
+---+-------+---------+----+------+----------+
```

```
df1.join(df2, df1.id == df2.id, "leftouter").show()
```

```
+---+-------+---------+----+------+----------+
| id|   Name|  Company|  id|salary|department|
+---+-------+---------+----+------+----------+
|  1| sravan|company 1|   1| 45000|        IT|
|  2| ojaswi|company 1|   2|145000|   Manager|
|  3| rohith|company 2|NULL|  NULL|      NULL|
|  4|sridevi|company 1|NULL|  NULL|      NULL|
+---+-------+---------+----+------+----------+
```

Right Join

- Returns all rows from the right DataFrame and the matched rows from the left DataFrame.
- If no match is found in the left DataFrame, NULL values will be returned for the left DataFrame columns.

Syntax

right: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"right")

rightouter: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"rightouter")

```
df1.join(df2, df1.id == df2.id, "right").show()
```

```
+----+------+---------+---+------+----------+
|  id|  Name|  Company| id|salary|department|
+----+------+---------+---+------+----------+
|   1|sravan|company 1|  1| 45000|        IT|
|   2|ojaswi|company 1|  2|145000|   Manager|
|NULL|  NULL|     NULL|  5| 34000|     Sales|
|NULL|  NULL|     NULL|  6| 45000|        HR|
+----+------+---------+---+------+----------+
```

```
df1.join(df2, df1.id == df2.id, "rightouter").show()
```

```
+----+------+---------+---+------+----------+
|  id|  Name|  Company| id|salary|department|
+----+------+---------+---+------+----------+
|   1|sravan|company 1|  1| 45000|        IT|
|   2|ojaswi|company 1|  2|145000|   Manager|
|NULL|  NULL|     NULL|  5| 34000|     Sales|
|NULL|  NULL|     NULL|  6| 45000|        HR|
+----+------+---------+---+------+----------+
```

Leftsemi join

- Returns only the rows from the left DataFrame that have a matching key in the right DataFrame.
- Unlike left join, it does not return any columns from the right DataFrame.

Syntax

dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"leftsemi")

```
df1.join(df2, df1.id == df2.id, "leftsemi").show()
```

```
+---+------+---------+
| id|  Name|  Company|
+---+------+---------+
|  1|sravan|company 1|
|  2|ojaswi|company 1|
+---+------+---------+
```

LeftAnti join

- Returns only the rows from the left DataFrame that do NOT have a match in the right DataFrame.
- It's like filtering out all matches and keeping only the unmatched records.

Syntax

dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,"leftanti")

```
df1.join(df2, df1.id == df2.id, "leftanti").show()
```

```
+---+-------+---------+
| id|   Name|  Company|
+---+-------+---------+
|  3| rohith|company 2|
|  4|sridevi|company 1|
+---+-------+---------+
```

Handling Null Values In Pyspark

Handling missing values (NULL/NA) is one of the most critical steps in data preprocessing. PySpark provides multiple tools to detect, drop, or fill missing values efficiently, especially for big data.

```python
from pyspark.sql import SparkSession
# Create Spark session
spark = SparkSession.builder.appName("NullValuesExample").getOrCreate()
# Data with None for NULL values
data = [
    (1, "John", "IT", 29, 4, 5000, 101),
    (2, "Alice", "HR", None, 2, None, 102),
    (3, None, "Sales", 35, None, 7000, None),
    (4, "Bob", None, None, None, None, None),
    (5, "Eve", "Finance", 40, 10, 10000, 105),
    (6, None, None, None, None, None, None),
]
# Column names
columns = ["Employee_ID", "Name", "Department", "Age", "Experience (yrs)", "Salary ($)", "Manager_ID"]
# Create DataFrame
df = spark.createDataFrame(data, columns)
# Show DataFrame
df.show()
df.printSchema()
```

```
+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+
```

Detecting Missing Values

If nullable=True for a column → It contains missing values (or it can potentially contain nulls).

```python
df.printSchema()
```

```
root
 |-- Employee_ID: long (nullable = true)
 |-- Name: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- Experience (yrs): long (nullable = true)
 |-- Salary ($): long (nullable = true)
 |-- Manager_ID: long (nullable = true)
```

Dropping NULL Value

PySpark provides df.na.drop() to remove rows with missing values.

a) Drop all rows with any NULL value

Removes rows if any column has a NULL value.

```
df.na.drop().show()
```

```
+-----------+----+----------+---+----------------+----------+----------+
|Employee_ID|Name|Department|Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+----+----------+---+----------------+----------+----------+
|          1|John|        IT| 29|               4|      5000|       101|
|          5| Eve|   Finance| 40|              10|     10000|       105|
+-----------+----+----------+---+----------------+----------+----------+
```

b) Using how parameter : Controls when to drop rows

Parameter	Behavior
any	Drops rows if **at least one NULL** is present in any column.
`all`	Drops rows only if **all columns** in the row are NULL.

```
df.na.drop(how="any").show()
df.na.drop(how="all").show()
```

```
+-----------+----+----------+---+----------------+----------+----------+
|Employee_ID|Name|Department|Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+----+----------+---+----------------+----------+----------+
|          1|John|        IT| 29|               4|      5000|       101|
|          5| Eve|   Finance| 40|              10|     10000|       105|
+-----------+----+----------+---+----------------+----------+----------+

+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+
```

c) Using thresh parameter

Drops rows based on a minimum number of non-null values required:

```
df.na.drop(thresh=2).show() #Keeps rows with at least 2 non-null values, drops others.
```

```
+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
+-----------+-----+----------+----+----------------+----------+----------+
```

d) Using subset parameter : Drop null values only for specific columns

```
df.na.drop(how="any", subset=["Experience (yrs)"]).show()
```

```
+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
+-----------+-----+----------+----+----------------+----------+----------+
```

Passing multiple columns to drop the null values

```
df.na.drop(how="any", subset=["Age", "Department"]).show()
```

```
+-----------+----+----------+---+----------------+----------+----------+
|Employee_ID|Name|Department|Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+----+----------+---+----------------+----------+----------+
|          1|John|        IT| 29|               4|      5000|       101|
|          3|NULL|     Sales| 35|            NULL|      7000|      NULL|
|          5| Eve|   Finance| 40|              10|     10000|       105|
+-----------+----+----------+---+----------------+----------+----------+
```

Filling Missing Values (Imputation)

Instead of dropping rows, we can replace NULL values using fill() or fillna().

a) Fill NULL values with a constant

```
df.na.fill('NA', 'Name').show() #Fill String Column
df.na.fill(0, subset =["Age","Experience (yrs)","Salary ($)","Manager_ID"]).show() #Fill Numeric Column
df.na.fill('NA', subset=["Name", "Department"]).show()
```

```
+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3|   NA|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6|   NA|      NULL|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+

+-----------+-----+----------+---+----------------+----------+----------+
|Employee_ID| Name|Department|Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+---+----------------+----------+----------+
|          1| John|        IT| 29|               4|      5000|       101|
|          2|Alice|        HR|  0|               2|         0|       102|
|          3| NULL|     Sales| 35|               0|      7000|         0|
|          4|  Bob|      NULL|  0|               0|         0|         0|
|          5|  Eve|   Finance| 40|              10|     10000|       105|
|          6| NULL|      NULL|  0|               0|         0|         0|
+-----------+-----+----------+---+----------------+----------+----------+

+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3|   NA|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|        NA|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6|   NA|        NA|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+
```

b) Fill NULL values for all string or numeric columns

```
df.na.fill('').show() # Replaces NULL with empty string for all string columns
df.na.fill(0).show()  # Replaces NULL with 0 for all numeric columns
```

```
+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3|     |     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|          |NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6|     |          |NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+

+-----------+-----+----------+---+----------------+----------+----------+
|Employee_ID| Name|Department|Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+---+----------------+----------+----------+
|          1| John|        IT| 29|               4|      5000|       101|
|          2|Alice|        HR|  0|               2|         0|       102|
|          3| NULL|     Sales| 35|               0|      7000|         0|
|          4|  Bob|      NULL|  0|               0|         0|         0|
|          5|  Eve|   Finance| 40|              10|     10000|       105|
|          6| NULL|      NULL|  0|               0|         0|         0|
+-----------+-----+----------+---+----------------+----------+----------+
```

c) Fill NULL values using a dictionary (different values for different columns)

```
df.na.fill({"Name":"Unknown", "Salary ($)": 0}).show()
```

```
+-----------+-------+----------+----+----------------+----------+----------+
|Employee_ID|   Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-------+----------+----+----------------+----------+----------+
|          1|   John|        IT|  29|               4|      5000|       101|
|          2|  Alice|        HR|NULL|               2|         0|       102|
|          3|Unknown|     Sales|  35|            NULL|      7000|      NULL|
|          4|    Bob|      NULL|NULL|            NULL|         0|      NULL|
|          5|    Eve|   Finance|  40|              10|     10000|       105|
|          6|Unknown|      NULL|NULL|            NULL|         0|      NULL|
+-----------+-------+----------+----+----------------+----------+----------+
```

Imputing Missing Values using Statistics (Mean, Median, Mode)

PySpark's Imputer (from pyspark.ml.feature) can replace NULL values with the mean, median, or mode.

Mean

```
from pyspark.ml.feature import Imputer
imputer = Imputer(
    inputCols = ['Age', 'Experience (yrs)', 'Salary ($)'],
    outputCols=["{}_imputed".format(c) for c in ['Age', 'Experience (yrs)', 'Salary ($)']]).setStrategy("mean")
df_imputed = imputer.fit(df).transform(df)
df_imputed.show()
```

```
+-----------+-----+----------+----+----------------+----------+----------+-----------+-----------------------+-----------------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|Age_imputed|Experience (yrs)_imputed|Salary ($)_imputed|
+-----------+-----+----------+----+----------------+----------+----------+-----------+-----------------------+-----------------+
|          1| John|        IT|  29|               4|      5000|       101|         29|                      4|             5000|
|          2|Alice|        HR|NULL|               2|      NULL|       102|         34|                      2|             7333|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|         35|                      5|             7000|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|         34|                      5|             7333|
|          5|  Eve|   Finance|  40|              10|     10000|       105|         40|                     10|            10000|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|         34|                      5|             7333|
+-----------+-----+----------+----+----------------+----------+----------+-----------+-----------------------+-----------------+
```

Median

```
from pyspark.ml.feature import Imputer
imputer = Imputer(
    inputCols = ['Age', 'Experience (yrs)', 'Salary ($)'],
    outputCols=["{}_imputed".format(c) for c in ['Age', 'Experience (yrs)', 'Salary ($)']]).setStrategy("median")
df_imputed = imputer.fit(df).transform(df)
df_imputed.show()
```

```
+-----------+-----+----------+----+---------------+----------+----------+-----------+-----------------------+-----------------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|Age_imputed|Experience (yrs)_imputed|Salary ($)_imputed|
+-----------+-----+----------+----+---------------+----------+----------+-----------+-----------------------+-----------------+
|          1| John|        IT|  29|              4|      5000|       101|         29|                      4|             5000|
|          2|Alice|        HR|NULL|              2|      NULL|       102|         35|                      2|             7000|
|          3| NULL|     Sales|  35|           NULL|      7000|      NULL|         35|                      4|             7000|
|          4|  Bob|      NULL|NULL|           NULL|      NULL|      NULL|         35|                      4|             7000|
|          5|  Eve|   Finance|  40|             10|     10000|       105|         40|                     10|            10000|
|          6| NULL|      NULL|NULL|           NULL|      NULL|      NULL|         35|                      4|             7000|
+-----------+-----+----------+----+---------------+----------+----------+-----------+-----------------------+-----------------+
```

Mode

```
from pyspark.ml.feature import Imputer
imputer = Imputer(
    inputCols = ['Age', 'Experience (yrs)', 'Salary ($)'],
    outputCols=["{}_imputed".format(c) for c in ['Age', 'Experience (yrs)', 'Salary ($)']]).setStrategy("mode")
df_imputed = imputer.fit(df).transform(df)
df_imputed.show()
```

```
+-----------+-----+----------+----+---------------+----------+----------+-----------+-----------------------+-----------------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|Age_imputed|Experience (yrs)_imputed|Salary ($)_imputed|
+-----------+-----+----------+----+---------------+----------+----------+-----------+-----------------------+-----------------+
|          1| John|        IT|  29|              4|      5000|       101|         29|                      4|             5000|
|          2|Alice|        HR|NULL|              2|      NULL|       102|         29|                      2|             5000|
|          3| NULL|     Sales|  35|           NULL|      7000|      NULL|         35|                      2|             7000|
|          4|  Bob|      NULL|NULL|           NULL|      NULL|      NULL|         29|                      2|             5000|
|          5|  Eve|   Finance|  40|             10|     10000|       105|         40|                     10|            10000|
|          6| NULL|      NULL|NULL|           NULL|      NULL|      NULL|         29|                      2|             5000|
+-----------+-----+----------+----+---------------+----------+----------+-----------+-----------------------+-----------------+
```

Filtering NULL Values

Instead of dropping or filling, you can filter rows using isNull() or isNotNull():

```
from pyspark.sql.functions import col
df.filter(col("Age").isNull()).show() # Only rows with NULL salary
df.filter(col("Salary ($)").isNotNull()).show() # Only rows without NULL salary
```

```
+-----------+-----+----------+----+---------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+---------------+----------+----------+
|          2|Alice|        HR|NULL|              2|      NULL|       102|
|          4|  Bob|      NULL|NULL|           NULL|      NULL|      NULL|
|          6| NULL|      NULL|NULL|           NULL|      NULL|      NULL|
+-----------+-----+----------+----+---------------+----------+----------+

+-----------+----+----------+---+---------------+----------+----------+
|Employee_ID|Name|Department|Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+----+----------+---+---------------+----------+----------+
|          1|John|        IT| 29|              4|      5000|       101|
|          3|NULL|     Sales| 35|           NULL|      7000|      NULL|
|          5| Eve|   Finance| 40|             10|     10000|       105|
+-----------+----+----------+---+---------------+----------+----------+
```

Using Coalesce() for NULL Handling

coalesce() returns the first non-null value among columns

```
from pyspark.sql.functions import coalesce
df.withColumn("Final_Salary", coalesce(col("Salary ($)"), col("Experience (yrs)"))).show()
```

```
+-----------+-----+----------+----+----------------+----------+----------+------------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|Final_Salary|
+-----------+-----+----------+----+----------------+----------+----------+------------+
|          1| John|        IT|  29|               4|      5000|       101|        5000|
|          2|Alice|        HR|NULL|               2|      NULL|       102|           2|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|        7000|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|        NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|       10000|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|        NULL|
+-----------+-----+----------+----+----------------+----------+----------+------------+
```

Aggregations Ignore NULLs

When performing aggregations (avg, sum, etc.), Spark ignores NULL values:

```
from pyspark.sql.functions import avg
df.select(avg("Salary ($)")).show()
```

```
+-----------------+
|  avg(Salary ($))|
+-----------------+
|7333.333333333333|
+-----------------+
```

NULL Handling in Joins

When joining two DataFrames, NULLs in join keys do NOT match unless explicitly handled.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("NullValuesExample").getOrCreate()
d1 = [(1, "John", "IT", 29, 4, 5000, 101),
    (2, "Alice", "HR", None, 2, None, 102),
    (3, None, "Sales", 35, None, 7000, None),
    (4, "Bob", None, None, None, None, None),
    (5, "Eve", "Finance", 40, 10, 10000, 105),
    (6, None, None, None, None, None, None),]
d2 = [(None, "Alice", "HR", None, 2, None, 102),
    (3, None, "Sales", 35, None, 7000, None),
    (5, "Eve", "Finance", 40, 10, 10000, 105),
    (None, None, None, None, None, None, None),]
columns = ["Employee_ID", "Name", "Department", "Age", "Experience (yrs)", "Salary ($)", "Manager_ID"]
df1 = spark.createDataFrame(d1, columns)
df2 = spark.createDataFrame(d1, columns)
df1.show()
df1.printSchema()
df2.show()
df2.printSchema()
```

```
+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+
```

```
root
 |-- Employee_ID: long (nullable = true)
 |-- Name: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- Experience (yrs): long (nullable = true)
 |-- Salary ($): long (nullable = true)
 |-- Manager_ID: long (nullable = true)

+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|
|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+

root
 |-- Employee_ID: long (nullable = true)
 |-- Name: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- Experience (yrs): long (nullable = true)
 |-- Salary ($): long (nullable = true)
 |-- Manager_ID: long (nullable = true)
```

```
df1.join(df2, df1.Employee_ID == df2.Employee_ID, "left").show()
```

```
+-----------+-----+----------+----+----------------+----------+----------+-----------+-----+----------+----+----------------+----------+----------+
|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|Employee_ID| Name|Department| Age|Experience (yrs)|Salary ($)|Manager_ID|
+-----------+-----+----------+----+----------------+----------+----------+-----------+-----+----------+----+----------------+----------+----------+
|          1| John|        IT|  29|               4|      5000|       101|          1| John|        IT|  29|               4|      5000|       101|
|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|          3| NULL|     Sales|  35|            NULL|      7000|      NULL|
|          2|Alice|        HR|NULL|               2|      NULL|       102|          2|Alice|        HR|NULL|               2|      NULL|       102|
|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|          6| NULL|      NULL|NULL|            NULL|      NULL|      NULL|
|          5|  Eve|   Finance|  40|              10|     10000|       105|          5|  Eve|   Finance|  40|              10|     10000|       105|
|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|          4|  Bob|      NULL|NULL|            NULL|      NULL|      NULL|
+-----------+-----+----------+----+----------------+----------+----------+-----------+-----+----------+----+----------------+----------+----------+
```

If ID is NULL in either DataFrame, it will not match. Use fillna() or coalesce() before joining if required.

When - Otherwise

- when is a conditional function from pyspark.sql.functions.
- It is used to apply "if-else logic" (similar to SQL CASE WHEN) to transform or create new columns.
- Always used with .otherwise() for the "else" part (if no conditions match).

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("James","M",60000),
        ("Michael","M",70000),
        ("Robert",None,400000),
        ("Maria","F",500000),
        ("Jen","",None)]

columns = ["name","gender","salary"]

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

```
+-------+------+------+
|   name|gender|salary|
+-------+------+------+
|  James|     M| 60000|
|Michael|     M| 70000|
| Robert|  NULL|400000|
|  Maria|     F|500000|
|    Jen|      |  NULL|
+-------+------+------+
```

```
from pyspark.sql.functions import when,col
df2 = df.withColumn(
    "new_gender",
    when(df.gender == "M", "Male")
    .when(df.gender == "F", "Female")
    .when(df.gender.isNull(), "Not Specified")
    .when(df.gender == "", "Not Specified")
    .otherwise(df.gender))
df2.show()
```

```
+-------+------+------+-------------+
|   name|gender|salary|   new_gender|
+-------+------+------+-------------+
|  James|     M| 60000|         Male|
|Michael|     M| 70000|         Male|
| Robert|  NULL|400000|Not Specified|
|  Maria|     F|500000|       Female|
|    Jen|      |  NULL|Not Specified|
+-------+------+------+-------------+
```

Alternative Method: Adding New Column with select

```
df2 = df.select(
    col("*"),
    when(df.gender == "M", "Male")
    .when(df.gender == "F", "Female")
    .when(df.gender.isNull(), "Not Specified")
    .when(df.gender == "", "Not Specified")
    .otherwise(df.gender)
    .alias("New_Gender"))
df2.show()
```

```
+-------+------+------+-------------+
|   name|gender|salary|   New_Gender|
+-------+------+------+-------------+
|  James|     M| 60000|         Male|
|Michael|     M| 70000|         Male|
| Robert|  NULL|400000|Not Specified|
|  Maria|     F|500000|       Female|
|    Jen|      |  NULL|Not Specified|
+-------+------+------+-------------+
```

Using SQL CASE WHEN with expr

```
from pyspark.sql.functions import expr

df3 = df.withColumn(
    "New_gender",
    expr("CASE WHEN gender == 'M' THEN 'Male' "
         "WHEN gender == 'F' THEN 'Female' "
         "WHEN gender IS NULL THEN 'Not Specified' "
         "WHEN gender == '' THEN 'Not Specified' "
         "ELSE gender END")
)
df3.show()
```

```
+-------+------+------+-------------+
|   name|gender|salary|   New_gender|
+-------+------+------+-------------+
|  James|     M| 60000|         Male|
|Michael|     M| 70000|         Male|
| Robert|  NULL|400000|Not Specified|
|  Maria|     F|500000|       Female|
|    Jen|      |  NULL|Not Specified|
+-------+------+------+-------------+
```

Using select with SQL Expression

```
df4 = df.select(
    col("*"),
    expr(
        "CASE WHEN gender == 'M' THEN 'Male' "
        "WHEN gender == 'F' THEN 'Female' "
        "WHEN gender IS NULL THEN 'Not Specified' "
        "WHEN gender == '' THEN 'Not Specified' "
        "ELSE gender END").alias("New_Gender"))
df4.show()
```

```
+-------+------+------+-------------+
|   name|gender|salary|   New_Gender|
+-------+------+------+-------------+
|  James|     M| 60000|         Male|
|Michael|     M| 70000|         Male|
| Robert|  NULL|400000|Not Specified|
|  Maria|     F|500000|       Female|
|    Jen|      |  NULL|Not Specified|
+-------+------+------+-------------+
```

Using Spark SQL Directly

```
df.createOrReplaceTempView("Emp")
spark.sql("""
    select name,
    case when gender = 'M' then 'Male'
         when gender = 'F' then 'Female'
         when gender = '' then 'Not Specified'
         when gender is null then 'Not Specified'
         else gender end as new_gender
    from Emp
""").show()
```

```
+-------+-------------+
|   name|   new_gender|
+-------+-------------+
|  James|         Male|
|Michael|         Male|
| Robert|Not Specified|
|  Maria|       Female|
|    Jen|Not Specified|
+-------+-------------+
```

Union() and UnionALL()

What is union() in PySpark?

- union() is a transformation used to merge two or more DataFrames.
- Both DataFrames must have the same schema (same column names and data types).
- It returns all rows from both DataFrames, including duplicates.
- If you want only unique rows, you must call .distinct() after union().

What is unionAll()?

- unionAll() was used in older Spark versions (before 2.0.0).
- It is deprecated and replaced with union().
- In PySpark, union() and unionAll() behave the same: both keep duplicates.
- If you come across unionAll() in legacy code, you should replace it with union().

Difference Between PySpark and SQL Union

Feature	PySpark `union()`	SQL `UNION`	SQL `UNION ALL`
Duplicates	Kept	Removed	Kept
Schema Match	Required	Required	Required

Creating Two DataFrames

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

# First DataFrame
data1 = [("James","Sales","NY",90000,34,10000),
         ("Michael","Sales","NY",86000,56,20000),
         ("Robert","Sales","CA",81000,30,23000),
         ("Maria","Finance","CA",90000,24,23000)]
columns = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data1, schema=columns)

# Second DataFrame
data2 = [("James","Sales","NY",90000,34,10000),
         ("Maria","Finance","CA",90000,24,23000),
         ("Jen","Finance","NY",79000,53,15000),
         ("Jeff","Marketing","CA",80000,25,18000),
         ("Kumar","Marketing","NY",91000,50,21000)]
df2 = spark.createDataFrame(data2, schema=columns)
```

```
union_df = df.union(df2)
union_df.show(truncate=False)
```

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
+-------------+----------+-----+------+---+-----+
```

Removing Duplicates in union

```
union_df = df.union(df2).distinct()
union_df.show(truncate=False)
```

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
+-------------+----------+-----+------+---+-----+
```

8 What if Schemas are Different?

If DataFrames have different column order or names, use `unionByName()`:

python

```
df.unionByName(df2)
```

If one DataFrame has extra columns:

python

```
df.unionByName(df2, allowMissingColumns=True)
```

Union All Example

```
unionall_df = df.unionAll(df2)
unionall_df.show(truncate=False)
```

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
+-------------+----------+-----+------+---+-----+
```

Removing Duplicates in unionAll()

```
unionall_df = df.unionAll(df2).distinct()
unionall_df.show(truncate=False)
```

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
+-------------+----------+-----+------+---+-----+
```

Window Functions

Window functions let you:

- Look at neighboring rows (before/after).
- Perform aggregations without grouping into fewer rows.
- Use row numbers, ranks, and lag/lead functions across a defined "window" of data.

Types of Window Functions

1. Ranking Functions

Function	Description	Example Use Case
`row_number()`	Sequential numbering (1, 2, 3...)	Find the 1st highest salary per department
`rank()`	Ranking with gaps for ties	Competition ranking (1, 2, 2, 4)
`dense_rank()`	Ranking without gaps for ties	Consistent ranking (1, 2, 2, 3)
`percent_rank()`	Relative rank in percentage (0 to 1)	Calculate percentile for salaries
`ntile(n)`	Divides rows into `n`	Divide employees into 4 quartiles

2. Analytic Functions

Function	Description	Example Use Case
`cume_dist()`	Cumulative distribution (0 to 1)	See what percentage of employees earn ≤ salary X
`lag()`	Value from previous row	Compare current and previous salary
`lead()`	Value from next row	Compare current and next salary

3. Aggregate Functions

Any aggregate function (e.g. `sum`, `avg`, `min`, `max`) can be used in a window context.

Function	Description	Example Use Case
`sum()`	Cumulative sum	Total salary per department
`avg()`	Cumulative average	Average salary per department
`min()`	Minimum value	Min salary in a department
`max()`	Maximum value	Max salary in a department

3 How Window Functions Work

1. **Partition the data →** `Window.partitionBy("column")`
   Splits data into groups (like `GROUP BY` in SQL but doesn't collapse rows).
2. **Order within each partition →** `orderBy("column")`
   Needed for ranking, lag, and lead functions.
3. **Apply the window function →** `.over(windowSpec)`

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('WindowFunctions').getOrCreate()
data = [
    ("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)]
columns = ["employee_name", "department", "salary"]
df = spark.createDataFrame(data, schema=columns)
df.show()
```

```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|        James|     Sales|  3000|
|      Michael|     Sales|  4600|
|       Robert|     Sales|  4100|
|        Maria|   Finance|  3000|
|        James|     Sales|  3000|
|        Scott|   Finance|  3300|
|          Jen|   Finance|  3900|
|         Jeff| Marketing|  3000|
|        Kumar| Marketing|  2000|
|         Saif|     Sales|  4100|
+-------------+----------+------+
```

Ranking Functions

```python
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number, rank, dense_rank, percent_rank, ntile

windowSpec = Window.partitionBy("department").orderBy("salary")

#row number
df.withColumn("row_number", row_number().over(windowSpec)).show()

#rank
df.withColumn("rank", rank().over(windowSpec)).show()

#dense rank
df.withColumn("Dense_rank", dense_rank().over(windowSpec)).show()

#percent rank
df.withColumn("Percent_rank", percent_rank().over(windowSpec)).show()

# ntile
df.withColumn("Ntile", ntile(2).over(windowSpec)).show()
```

```
+-------------+----------+------+----+
|        Maria|   Finance|  3000|   1|
|        Scott|   Finance|  3300|   2|
|          Jen|   Finance|  3900|   3|
|        Kumar| Marketing|  2000|   1|
|         Jeff| Marketing|  3000|   2|
|        James|     Sales|  3000|   1|
|        James|     Sales|  3000|   1|
|       Robert|     Sales|  4100|   3|
|         Saif|     Sales|  4100|   3|
|      Michael|     Sales|  4600|   5|
+-------------+----------+------+----+

+-------------+----------+------+----------+
|employee_name|department|salary|Dense_rank|
+-------------+----------+------+----------+
|        Maria|   Finance|  3000|         1|
|        Scott|   Finance|  3300|         2|
|          Jen|   Finance|  3900|         3|
|        Kumar| Marketing|  2000|         1|
|         Jeff| Marketing|  3000|         2|
|        James|     Sales|  3000|         1|
|        James|     Sales|  3000|         1|
|       Robert|     Sales|  4100|         2|
|         Saif|     Sales|  4100|         2|
|      Michael|     Sales|  4600|         3|
+-------------+----------+------+----------+
+-------------+----------+------+----------+
|employee_name|department|salary|Dense_rank|
+-------------+----------+------+----------+
|        Maria|   Finance|  3000|         1|
|        Scott|   Finance|  3300|         2|
|          Jen|   Finance|  3900|         3|
|        Kumar| Marketing|  2000|         1|
|         Jeff| Marketing|  3000|         2|
|        James|     Sales|  3000|         1|
|        James|     Sales|  3000|         1|
|       Robert|     Sales|  4100|         2|
|         Saif|     Sales|  4100|         2|
|      Michael|     Sales|  4600|         3|
+-------------+----------+------+----------+
```

```
+-------------+----------+------+-----+
|employee_name|department|salary|Ntile|
+-------------+----------+------+-----+
|        Maria|   Finance|  3000|    1|
|        Scott|   Finance|  3300|    1|
|          Jen|   Finance|  3900|    2|
|        Kumar| Marketing|  2000|    1|
|         Jeff| Marketing|  3000|    2|
|        James|     Sales|  3000|    1|
|        James|     Sales|  3000|    1|
|       Robert|     Sales|  4100|    1|
|         Saif|     Sales|  4100|    2|
|      Michael|     Sales|  4600|    2|
+-------------+----------+------+-----+
```

Analytic Functions

```
from pyspark.sql.functions import cume_dist, lag, lead
#cume_dist
df.withColumn("cume_dist", cume_dist().over(windowSpec)).show()
#lag
df.withColumn("Lag", lag("salary", 2).over(windowSpec)).show()
#lead
df.withColumn("Lead", lead("salary",2).over(windowSpec)).show()
```

```
+-------------+----------+------+------------------+
|employee_name|department|salary|         cume_dist|
+-------------+----------+------+------------------+
|        Maria|   Finance|  3000|0.3333333333333333|
|        Scott|   Finance|  3300|0.6666666666666666|
|          Jen|   Finance|  3900|               1.0|
|        Kumar| Marketing|  2000|               0.5|
|         Jeff| Marketing|  3000|               1.0|
|        James|     Sales|  3000|               0.4|
|        James|     Sales|  3000|               0.4|
|       Robert|     Sales|  4100|               0.8|
|         Saif|     Sales|  4100|               0.8|
|      Michael|     Sales|  4600|               1.0|
+-------------+----------+------+------------------+

+-------------+----------+------+----+
|employee_name|department|salary| Lag|
+-------------+----------+------+----+
|        Maria|   Finance|  3000|NULL|
|        Scott|   Finance|  3300|NULL|
|          Jen|   Finance|  3900|3000|
|        Kumar| Marketing|  2000|NULL|
|         Jeff| Marketing|  3000|NULL|
|        James|     Sales|  3000|NULL|
|        James|     Sales|  3000|NULL|
|       Robert|     Sales|  4100|3000|
|         Saif|     Sales|  4100|3000|
|      Michael|     Sales|  4600|4100|
+-------------+----------+------+----+
```

```
+-------------+----------+------+----+
|employee_name|department|salary|Lead|
+-------------+----------+------+----+
|        Maria|   Finance|  3000|3900|
|        Scott|   Finance|  3300|NULL|
|          Jen|   Finance|  3900|NULL|
|        Kumar| Marketing|  2000|NULL|
|         Jeff| Marketing|  3000|NULL|
|        James|     Sales|  3000|4100|
|        James|     Sales|  3000|4100|
|       Robert|     Sales|  4100|4600|
|         Saif|     Sales|  4100|NULL|
|      Michael|     Sales|  4600|NULL|
+-------------+----------+------+----+
```

Aggregate Functions

```
from pyspark.sql.functions import col, avg, sum, min, max
windowSpecAgg = Window.partitionBy("department")
df.withColumn("avg_salary", avg(col("salary")).over(windowSpecAgg))\
  .withColumn("total_salary", sum(col("salary")).over(windowSpecAgg))\
  .withColumn("min_salary", min(col("salary")).over(windowSpecAgg))\
  .withColumn("max_salary", max(col("salary")).over(windowSpecAgg))\
  .show()
```

```
+-------------+----------+------+----------+------------+----------+----------+
|employee_name|department|salary|avg_salary|total_salary|min_salary|max_salary|
+-------------+----------+------+----------+------------+----------+----------+
|        Maria|   Finance|  3000|    3400.0|       10200|      3000|      3900|
|        Scott|   Finance|  3300|    3400.0|       10200|      3000|      3900|
|          Jen|   Finance|  3900|    3400.0|       10200|      3000|      3900|
|         Jeff| Marketing|  3000|    2500.0|        5000|      2000|      3000|
|        Kumar| Marketing|  2000|    2500.0|        5000|      2000|      3000|
|        James|     Sales|  3000|    3760.0|       18800|      3000|      4600|
|      Michael|     Sales|  4600|    3760.0|       18800|      3000|      4600|
|       Robert|     Sales|  4100|    3760.0|       18800|      3000|      4600|
|        James|     Sales|  3000|    3760.0|       18800|      3000|      4600|
|         Saif|     Sales|  4100|    3760.0|       18800|      3000|      4600|
+-------------+----------+------+----------+------------+----------+----------+
```

Date and Time Functions

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

# Create SparkSession
spark = SparkSession.builder \
            .appName('SparkByExamples.com') \
            .getOrCreate()
data=[["1","2020-02-01"],["2","2019-03-01"],["3","2021-03-01"]]
df=spark.createDataFrame(data,["id","input"])
df.show()
```

```
+---+----------+
| id|     input|
+---+----------+
|  1|2020-02-01|
|  2|2019-03-01|
|  3|2021-03-01|
+---+----------+
```

current_date()

Use current_date() to get the current system date. By default, the data will be returned in yyyy-dd-mm format.

```
[ ]  df.select(current_date().alias("current date")).show(1)
```

```
+------------+
|current date|
+------------+
|  2025-08-04|
+------------+
only showing top 1 row
```

date_format()

The below example uses date_format() to parses the date and converts from yyyy-dd-mm to MM-dd-yyyy format.

```
[ ]  df.select(col("input"), date_format(col("input"), "MM-dd-yyyy").alias("date_format")).show()
```

```
+----------+-----------+
|     input|date_format|
+----------+-----------+
|2020-02-01| 02-01-2020|
|2019-03-01| 03-01-2019|
|2021-03-01| 03-01-2021|
+----------+-----------+
```

to_date()

Below example converts string in date format yyyy-MM-dd to a DateType yyyy-MM-dd using to_date(). You can also use this to convert into any specific format.

```
[ ]  df.select(col("input"),
        to_date(col("input"), "yyy-MM-dd").alias("to_date")
      ).show()
```

```
+----------+----------+
|     input|   to_date|
+----------+----------+
|2020-02-01|2020-02-01|
|2019-03-01|2019-03-01|
|2021-03-01|2021-03-01|
+----------+----------+
```

datediff()

The below example returns the difference between two dates using datediff().

```
df.select(col("input"),
            datediff(current_date(), col("input")).alias("Date_difference")).show()
```

```
+----------+---------------+
|     input|Date_difference|
+----------+---------------+
|2020-02-01|           2011|
|2019-03-01|           2348|
|2021-03-01|           1617|
+----------+---------------+
```

months_between()

The below example returns the months between two dates using months_between().

```
df.select(col("input"),
          months_between(current_date(), col("input")).alias("month_between")
          ).show()
```

```
+----------+-------------+
|     input|month_between|
+----------+-------------+
|2020-02-01|  66.09677419|
|2019-03-01|  77.09677419|
|2021-03-01|  53.09677419|
+----------+-------------+
```

trunc()

The below example truncates the date at a specified unit using trunc().

```
df.select(col("input"),
          trunc(col("input"),"Month").alias("Month_trunc"),
          trunc(col("input"), "Year").alias("Year_Trunc"),
          ).show()
```

```
+----------+-----------+----------+
|     input|Month_trunc|Year_Trunc|
+----------+-----------+----------+
|2020-02-01| 2020-02-01|2020-01-01|
|2019-03-01| 2019-03-01|2019-01-01|
|2021-03-01| 2021-03-01|2021-01-01|
+----------+-----------+----------+
```

add_months() , date_add(), date_sub()

```
[ ]  df.select(col("input"),
               add_months(col("input"),3).alias("add_3_months"),
               add_months(col("input"),-3).alias("sub 3 months"),
               date_add(col("input"), 4).alias("date add"),
               date_sub(col("input"),4).alias("date sub")).show()
```

```
+----------+------------+------------+----------+----------+
|     input|add_3_months|sub 3 months|  date add|  date sub|
+----------+------------+------------+----------+----------+
|2020-02-01|  2020-05-01|  2019-11-01|2020-02-05|2020-01-28|
|2019-03-01|  2019-06-01|  2018-12-01|2019-03-05|2019-02-25|
|2021-03-01|  2021-06-01|  2020-12-01|2021-03-05|2021-02-25|
+----------+------------+------------+----------+----------+
```

year(), month(), month(),next_day(), weekofyear()

```
df.select(col("input"),
          year(col("input")).alias("year"),
          month(col("input")).alias("month"),
          next_day(col("input"),"Sunday").alias("Next day"),
          weekofyear(col("input")).alias("week of year")
          ).show()
```

```
+----------+----+-----+----------+------------+
|     input|year|month|  Next day|week of year|
+----------+----+-----+----------+------------+
|2020-02-01|2020|    2|2020-02-02|           5|
|2019-03-01|2019|    3|2019-03-03|           9|
|2021-03-01|2021|    3|2021-03-07|           9|
+----------+----+-----+----------+------------+
```