PySpark

PySpark is the Python API for Apache Spark, an open-source, designed for big data processing and analytics.

It allows developers to write Spark applications using Python, making it easier for data scientists and engineers familiar with Python to perform big data processing.

PySpark supports data manipulation, SQL queries, machine learning, and stream processing — all in a distributed environment.

Role of PySpark in Big Data Processing

- In the era of Big Data, datasets are too large for traditional tools like Pandas or R to handle efficiently.
- PySpark enables:
    - Parallel processing of large datasets across a cluster of machines.
    - Fault-tolerant and scalable operations.
    - Seamless integration with tools like HDFS, Hive, Kafka, and S3.

Use Cases:

- ETL pipelines
- Batch processing of logs
- Real-time analytics
- Machine learning on large datasets

Core APIs available in PySpark:

- RDD API – For low-level transformations (more control, less optimized).
- DataFrame API – High-level abstraction (recommended).
- SQL API – Query data using SQL syntax.
- MLlib – Machine learning algorithms.
- Streaming API – Real-time data processing.

Speed: PySpark can process huge datasets in parallel across multiple machines. Your laptop never stood a chance.

Scale: Whether you're working with 10MB or 10TB of data, PySpark's got your back.

Power: SQL-like queries, machine learning, real-time analytics-all rolled into one package.

Spark Components

Apache Spark is made up of several modular components, each handling specific types of big data workloads. These components work together under the Spark ecosystem.

1. **Spark Core**

- The foundation of the Spark platform.
- Provides:
    - Memory management

    - Fault recovery
    - Job scheduling
    - Task dispatching
- Contains the RDD (Resilient Distributed Dataset) API — a low-level data structure used to represent distributed data.

Key Concept: Every other component in Spark is built on top of Spark Core.

2. Spark SQL

- A module for structured data processing.
- Allows you to run SQL queries over data and use DataFrames and Datasets.
- Optimized using Catalyst Optimizer for query planning.
- You can mix SQL with Python, Scala, or Java code.

3. Spark Streaming

- Used for real-time stream processing.
- Processes data from sources like:
    - Kafka
    - Flume
    - HDFS
- Works by dividing the data stream into small batches (micro-batching).

4. MLlib (Machine Learning Library)

- Built-in scalable **machine learning** library.
- Supports:
    - Classification
    - Regression
    - Clustering
- Works on DataFrames and can be combined with Spark SQL.

5. GraphX

- API for graph processing and graph-parallel computation.
- Enables computation on graphs like:
    - PageRank
    - Shortest Path
    - Connected Components
- Available only in Scala and Java (limited support in PySpark).

All Spark components share the same execution engine and run on the same SparkContext or SparkSession.

Key components of PySpark

RDD (Resilient Distributed Datasets) –

RDDs are the fundamental data structure in spark. They are immutable distributed collections of objects that can be processed in parallel.

```
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder.appName("RDD Example").getOrCreate()

# Create RDD from a Python list
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)

# Show contents
print(rdd.collect())
```

```
# Filter to create a new RDD
even_rdd = rdd.filter(lambda x: x % 2 == 0)

print(even_rdd.collect())
```

RDD transformations (like filter) create a **new RDD** without changing the original (immutability).

DataFrames –

DataFrames are similar to RDDs but with additional features like named columns, and support fro a wide range of data sources. They are analogous to tables in a relational database and provide a higher-level abstraction for data manipulation.

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

rdd = spark.sparkContext.parallelize([(1,"a"),(2,"b")])
rdd.collect()

print("rdd to dataframe")
rdd_to_df = rdd.toDF(["id","name"])
rdd_to_df.show()
```

```
print("dataframe to rdd")
df = spark.createDataFrame([(1,"h"),(2,"g")],["id","name"])

df_to_rdd = df.rdd
df_to_rdd.collect()
```

SparkSession in PySpark

SparkSession is a entry point for working with spark, and it comes with a wide range of configuration options. These options help you control resources, optimize data processing, and customize logging for better debugging.

It provides a **single unified interface** to work with:

- RDDs (Resilient Distributed Datasets)
- DataFrames
- SQL
- Datasets

introduced in Spark 2.0 to replace older contexts like:

- SparkContext
- SQLContext
- Without creating a SparkSession, you **cannot** use Spark's features in PySpark.
- It handles:
  - Session configurations
  - Connection to the cluster
  - Data reading and writing
  - Execution of transformations and actions

Step1: Creating a SparkSession (basic spark session setup)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Intro").getOrCreate()
```

. builder → A builder pattern used to configure and create the SparkSession instance

SparkSession.builder → start configuring a new SparkSession

appName(“Intro”) → Name of your spark application

getorCreate() → Get an existing session or creates a new session. Returns an existing one if already created (to avoid duplicate sessions).

Step2: Configuration options for SparkSession

Memory and Resource management

These settings let you control memory allocation and CPU usage for spark’s distributed process:

- **Executor memory** (spark.executor.memory): sets the memory allocation for each executor
- **Driver memory** (spark.driver.memory): allocates memory for the driver.
- **Core allocation** (spark.executor.cores): specifies the number of CPU cores per executor.

Spark = session.builder.appName(“Optimized app”).config(“spark.executor.memory”, “2g”).config(“spark.driver.memory”,”1g”).config(“spark.executor.cores”,2).getOrCreate()

Step3: Finalizing your SparkSession setup

Once configured, call .getOrCreate() to initialize the spark session with your specified setting.

Create DataFrame

The DataFrame API in Apache Spark (especially PySpark) is a high-level abstraction for working with structured and semi-structured data. It's similar to a table in a relational database or a DataFrame in pandas but optimized for distributed processing across a cluster.

```
from pyspark.sql import SparkSession

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

data = [("Arun", 25), ("Dinesh", 30)]
df = spark.createDataFrame(data, ["name", "age"])
df.show()
```

```
+------+---+
|  name|age|
+------+---+
|  Arun| 25|
|Dinesh| 30|
+------+---+
```

Working with CSV files

To read a CSV file, use **spark.read.csv().**

Pyspark offers various options to customize how the csv is read, so you can handle headers, delimiter, schemas and more.

Header → If your csv has a header row, set **header = True** to use it as column names.

InferSchema → To automatically determine the data types of each column, set **inferSchema = True.**

Delimiter: To use a custom delimiter (such as ; or |), set the sep option.

```
# "C:\Users\GowdhamanBJ\Downloads\customers-100.csv"
df_csv = spark.read.csv( path: "C:\\Users\\GowdhamanBJ\\Downloads\\customers-100.csv", header=True, inferS
df_csv.printSchema()
df_csv.show(5)
```

PySpark Data Structure

PySpark, the Python API for Apache Spark, provides various data structures to handle large-scale, distributed data. These structures range from low-level collections like RDDs to high-level schemas like StructType.

Data Structure	Description	Use Case
RDD	Low-level collection of elements	Custom transformations, fine control
DataFrame	High-level table-like structure	SQL operations, structured data analysis
Row	Represents a single record	Used in DataFrame
Column	Represents column expressions	Select, filter, transform operations
StructType	Defines schema of a DataFrame	DataFrame creation, file reading
StructField	Describes a field in StructType	Column name, type, nullability
ArrayType	Defines an array/list field in schema	Nested or repeated data
MapType	Defines key-value field in schema	JSON, NoSQL-style data

1. RDD (Resilient Distributed Dataset)

- Definition: A low-level, fault-tolerant distributed collection of elements.
- Immutable and can be operated on in parallel.
- Supports transformations (like map, filter) and actions (collect, count)

2. DataFrame

- Definition: A distributed collection of data organized into named columns.
- High-level abstraction built on top of RDDs.
- Allows SQL queries, optimized execution via Catalyst engine.

3. Row

- Represents a record in a DataFrame.
- Acts like a named tuple, accessed by column name or index.

4. Column

- Represents a column expression in a DataFrame.
- Used in select, filter, and other DataFrame transformations.

A **schema** in PySpark defines the **structure of a DataFrame**.

It describes each column's name, data type, and nullability.

Schemas ensure data consistency, enable optimization, and allow PySpark to validate data during read/write operations.

5. StructType

- Used to define the schema of a DataFrame.
- Acts like a table schema with multiple fields.

6. StructField

- Defines an **individual field** in a StructType.
- Includes field name, type, and nullability.

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
schema = StructType([
StructField("name", StringType(), True),
StructField("age", IntegerType(), False)
])
```

7. ArrayType

- Represents a column with array values (list of elements).
- All elements in the array must be of the same data type.

```
from pyspark.sql.types import ArrayType
ArrayType(StringType())
```

8. MapType

- Represents a column with **key-value pairs** (like a dictionary).

```
from pyspark.sql.types import MapType
MapType(StringType(), IntegerType())
```

Spark Context

SparkContext is the **entry point** to using **Apache Spark's RDD (Resilient Distributed Dataset) API**.
It **represents the connection** between the **driver program** and the **Spark cluster**.
It's responsible for:

- Initializing Spark execution environment
- Coordinating the **job scheduling**
- **Distributing data** and code across the cluster
- Managing **cluster resources**

Creating and Configuring SparkContext

In modern PySpark, SparkContext is **automatically created** as part of SparkSession. But it can also be created manually when needed (usually in older versions or lower-level APIs).

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Intro").getOrCreate()
sc = spark.sparkContext   # Get the underlying SparkContext
print(sc)
```

```
<SparkContext master=local[*] appName=Intro>
```

Method	Description
sc.textFile("path")	Read a text file as RDD
sc.parallelize(data)	Convert Python collection to RDD
sc.getConf()	Get the configuration
sc.version	Check Spark version
sc.appName	Get the application name
sc.master	Get the master URL

sparkContext acts as the entry point to the lower-level spark RDD API,
created via – automatically via SparkSession, or manually using **SparkConf**
Responsibility – Cluster connection, job scheduling, resource management.

RDD vs Dataframe
Both are same in immutability, in memory, resilient, distributed computing

Dataframe differs from RDD in maintaining the structure of the underlying data in the form of schema. Thus, Dataframe is equivalent to table in Database

When data is structured, information about the data can be collected and used to derive the best execution plan through RBO, CBO etc., in Database

PySpark DataFrames

- A DataFrame is a distributed collection of data organized into named columns, similar to a table in a relational database or a DataFrame in pandas.
- It is built on top of RDDs but offers higher-level API, optimizations, and support for SQL-like operations.

Key Features

Feature	Description
Schema-aware	Columns have names and types
Lazy Execution	Operations are not computed until an action is triggered
Optimized	Uses Catalyst optimizer for query optimization
Interoperable	Can run SQL queries using `spark.sql()`
Efficient	Better performance than RDDs due to optimization and Tungsten execution engine

```python
df = spark.createDataFrame(data, ["name", "age"])
df.show()

df.select("name").show()

# Filtering Rows
df.filter(df.age == 25).show()

# SQL Syntax
df.createOrReplaceTempView("details")
spark.sql("SELECT name FROM details WHERE age = 30").show()

#Sorting
df.orderBy("name").show()

# Schema Inspection
df.printSchema()
```

PySpark SQL?

- PySpark SQL allows you to run SQL queries on DataFrames and RDD.
- It enables SQL-style programming using Spark's distributed computing engine.
- You can combine SQL with DataFrame APIs for flexible and powerful data processing

Create both temporary and permanent table creation – based on whether it''s for SQL querying or for storage.

1) Temporary table (for sql queries)
   df.createOrReplaceTempView("employees")

2) Global Temporary view (accessible across sessions)
   df.createOrReplaceGlobalTempView("employees")

3) Save as a permanent Table
   df.write.saveAsTable("employee_table")

1. createTempView(name)

- Purpose: Creates a session-scoped temporary view.
- Scope: Exists only in the current SparkSession.
  When the session ends, the view disappears.
- Error behavior:
  If a temp view with the same name already exists, Spark will throw an error.

2. createOrReplaceTempView(name)

- Purpose: Similar to createTempView but overwrites the view if it already exists in the session.
- Scope: Same session-only scope as createTempView.
- Error behavior:
  No error if the view exists — it simply replaces it.

```
from pyspark.sql import SparkSession

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

data = [(1, "Arun", "Chennai"), (2, "Kavin", "Bangalore")]
columns = ["id", "name", "city"]

df = spark.createDataFrame(data, columns)

# Register the DataFrame as a temporary table
df.createOrReplaceTempView("people")

# Now you can use SQL queries!
result = spark.sql("select name, city from people where city = 'Chennai'")
result.show()
```

```
+----+-------+
|name|   city|
+----+-------+
|Arun|Chennai|
+----+-------+
```

Techniques for caching and persisting RDDs and DataFrames.

Why do we need caching/persistence?

- Spark is lazy: transformations are not executed until an action is triggered.
- **Each time you call an action (count(), collect(), show()), Spark recomputes the full lineage (DAG) of transformations.**
- If the same DataFrame/RDD is reused multiple times → huge overhead.

Caching/persistence helps by storing intermediate results in memory (or disk) to avoid recomputation.

Why use caching and persistence in pyspark?
Pyspark lazy evaluation means transformations are not executed immediately but are recorded as lineage (DAG).
When you run an action (like count(), collect() or show()), Spark computes the result by tracing back through all the transformations (lineage) applied on the original dataset. If you run another action on the same dataset, Spark will recompute everything again, unless you cache or persist the data.

What is Caching and persistence in PySpark?
Caching and persistence in pyspark refer to the process of storing intermediate DataFrame or RDD results in memory or disk so they can be reused without being recomputed.

Caching

- Shortcut for persist() with **MEMORY_ONLY** storage level.
- Stores the RDD/DataFrame in **memory only**.
- If data doesn't fit in memory, some partitions will be recomputed when needed.

Persistence

- More **flexible** than cache().
- You can choose where to store data (memory, disk, both, serialized).

```
from pyspark import StorageLevel

df.persist(StorageLevel.MEMORY_AND_DISK)   # stores in memory, spills to disk if needed
df.count()
```

Caching and persistence address this by:
Reducing computation overhead: storing intermediate results eliminates the need to recompute transformations, saving cpu resources.
Improving query performance: frequently accessed data is readily available, reducing latency in iterative or interactive applications.

DAG (Directed Acyclic Graph) – not the actual data or any execution happens.

```
df = spark.read.csv("data.csv", header=True)

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

DAG – Lineage (Execution plan)

- **Spark knows: "To get df_filterd, I must read the file and filter age > 30"**

No computation has happened yet means

- No csv file read,
- No rows are filtered
- Nothing is cached or stored yet,
- The data is not even loaded into memory.

When does computation start?

```
df_filtered.cache()

df_filtered.count()    #first action triggers actual DAG execution
```

Then:

- Spark executes the DAG up to df_filtered

- After computing, it stores the result in memory (because of cache()).

Caching in pyspark:
What it does:

- Stores the computed result of an RDD or DataFrame in memory.
- On the first action, spark computes and saves the result.
- On subsequent actions, spark uses the cached version of instead of recomputing.

Default behaviour:

- When you call ".cache()" on a DataFrame or RDD, spark stores it in memory if there's enough RAM.

```
df = spark.read.csv( path: "data.csv", header=True)
df.cache()  # Marks it to be cached in memory
df.count()  # Triggers computation and caches result
df.show()   # Uses cached result
```

df = spark.read.csv("/path/data.csv", header=True)
df.cache() # stored in memory
df.count() # first action → triggers computation + caching
df.show() # now fetched from cache, faster

Persistence in PySpark

What it does:

- Like caching but gives more control over how and where data is stored.
- You can choose storage levels: Memory, disk, serialized, off-heap, etc.

MEMORY_ONLY – Stores deserialized objects in memory. Fastest, but data is lost if memory is full.
MEMORY_AND_DISK – Tries memory first, spills to disk if needed. Default for dataframes.
DISK_ONLY – stores only on disk. Useful for very large data.
MEMORY_ONLY_SER – stores serialized objects in memory. Saves space, but slower access.
MEMORY_AND_DISK_SER – serialized memory + disk overflow. Balance between speed and memory usage.
OFF_HEAD – uses off-JVM memory (requires special setup).

```
from pyspark import StorageLevel
rdd = sc.textFile("data.txt")
rdd.persist(StorageLevel.MEMORY_AND_DISK)
rdd.count()
```

Feature	cache ()	persist()

Storage Level	Fixed (MEMORY_AND_DISK or MEMORY_ONLY)	You choose from many options
Flexibility	Limited	More configurable

When to Use

- Use ".cache()" when:
  - You're working with a **moderate-sized** dataset.
  - You'll reuse the data multiple times.
  - Default memory behavior is fine.

- Use ". persist(StorageLevel)" when:
  - You need **control over storage**.
  - Data is **too big for memory only**.
  - You want to optimize **CPU vs memory vs disk** tradeoffs.

Caching: Stores the data **in memory (RAM)**.
Persisting: Stores the data in **memory and/or disk** depending on storage level.

General DataFrame functions

Create DataFrame

```
spark  = SparkSession.builder.appName("general dataframe funciton").getOrCreate()
data = [
    (1, "Arun", "Chennai", "India", 30),
    (2, "Karthi", "Hosur", "India", 23),
    (3, "Kavin", "Salem", "India", 31),
    (4, "Ravi", "Bangalore", "India", 25),
    (5, "Kumar", "Bangalore", "India", 25),
    (6, "Arun", "Chennai", "India", 30)
]

df = spark.createDataFrame(data,["id","name","city","country","age"])
df.show()
```

1) df.show() -- Displays the top rows of the DataFrame in a tabular format.

```
+---+------+---------+-------+---+
| id|  name|     city|country|age|
+---+------+---------+-------+---+
|  1|  Arun|  Chennai|  India| 30|
|  2|Karthi|    Hosur|  India| 23|
|  3| Kavin|    Salem|  India| 31|
|  4|  Ravi|Bangalore|  India| 25|
|  5| Kumar|Bangalore|  India| 25|
|  6|  Arun|  Chennai|  India| 30|
+---+------+---------+-------+---+
```

df.show(3)

```
+---+------+-------+-------+---+
| id|  name|   city|country|age|
+---+------+-------+-------+---+
|  1|  Arun|Chennai|  India| 30|
|  2|Karthi|  Hosur|  India| 23|
|  3| Kavin|  Salem|  India| 31|
+---+------+-------+-------+---+
only showing top 3 rows
```

2) Collect () – Returns all rows in the DataFrame as a list of Row objects to the driver.

```
row = df.collect()
print(row)
```

```
[Row(id=1, name='Arun', city='Chennai', country='India', age=30), Row(id=2, name='Karthi', city='Hosur', country='India', age=23), Row(id=3, na
```

```
row = df.collect()
for r in row:
    print(r)
```

```
Row(id=1, name='Arun', city='Chennai', country='India', age=30)
Row(id=2, name='Karthi', city='Hosur', country='India', age=23)
Row(id=3, name='Kavin', city='Salem', country='India', age=31)
Row(id=4, name='Ravi', city='Bangalore', country='India', age=25)
Row(id=5, name='Kumar', city='Bangalore', country='India', age=25)
Row(id=6, name='Arun', city='Chennai', country='India', age=30)
```

3) take(n)
Return the first n rows as a list (similar to collect(), but limited).

```
# take(n)
some_row = df.take(3)
print(some_row)
for s in some_row:
    print(s)
```

```
[Row(id=1, name='Arun', city='Chennai', country='India', age=30), Row(id=2, name='Karthi', city='Hosur', country='India', age=23)]
Row(id=1, name='Arun', city='Chennai', country='India', age=30)
Row(id=2, name='Karthi', city='Hosur', country='India', age=23)
```

4) printSchema() – prints the schema (column names and data types) of the dataframe

```
# schema
df.printSchema()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- city: string (nullable = true)
 |-- country: string (nullable = true)
 |-- age: long (nullable = true)
```

PySpark default behavior
When you create a DataFrame **without explicitly defining the schema**, like:
df = spark.createDataFrame(data, ["id", "name","city"])

PySpark automatically **infers the schema** and **sets all fields as nullable = true by default**.
Because Spark doesn't know if future data might have nulls — it's designed for distributed systems and evolving data. So it keeps fields nullable unless you **explicitly tell it not to**.

If you want nullable = false
You must define the schema manually:

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

schema = StructType([
    StructField("id", IntegerType(), False),
    StructField("order_date", StringType(), False),
    StructField("delivery_date", StringType(), False)
])

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

5) count() – returns the number of rows in dataframe

```
# count
df.count()
```

6) select – select one or more specific columns from dataframe. (Returns a new DataFrame with only selected columns).

```
# select
df.select("name","age").show()
df.select(df["name"].alias("user name")).show()
df.select(df.id).show()
```

```
+------+---+
|  name|age|
+------+---+
|  Arun| 30|
|Karthi| 23|
| Kavin| 31|
|  Ravi| 25|
| Kumar| 25|
|  Arun| 30|
+------+---+
```

7) filter/ where () – filters rows based on conditions

```
df.filter(df["age"] > 30).show()
df.where(df["city"] == "Chennai").show()
```

```
+---+-----+-----+-------+---+
| id| name| city|country|age|
+---+-----+-----+-------+---+
|  3|Kavin|Salem|  India| 31|
+---+-----+-----+-------+---+
```

```
+---+----+-------+-------+---+
| id|name|   city|country|age|
+---+----+-------+-------+---+
|  1|Arun|Chennai|  India| 30|
|  6|Arun|Chennai|  India| 30|
+---+----+-------+-------+---+
```

```
# Filter rows with salary greater than 6000
print("Filter: Salary > 6000")
df.filter(df.salary > 6000).show()

# Filter rows belonging to a specific department
print("Filter: Department = 'IT'")
df.filter(df.department == "IT").show()

# Combine multiple filter conditions
print("Filter: Salary > 5000 and Department = 'HR'")
df.filter((df.salary > 5000) & (df.department == "HR")).show()
```

```
# Filter using isin (e.g., department is either IT or HR)
print("Filter: Department in ('IT', 'HR')")
df.filter(df.department.isin( *cols: "IT", "HR")).show()
```

from PySpark.sql.functions import col

```
print("Filter: Rows where department is not null")
df_with_nulls.filter(col("department").isNotNull()).show()

print("Filter: Rows where name is null")
df_with_nulls.filter(col("name").isNull()).show()
```

8) like -- used inside filter() to match string patterns.

```
df.filter(df["name"].like("A%")).show()
```

```
+---+----+-------+-------+---+
| id|name|   city|country|age|
+---+----+-------+-------+---+
|  1|Arun|Chennai|  India| 30|
+---+----+-------+-------+---+
```

9) sort – sort rows by one or more columns. (return new dataframe with sorted rows).

```
# df.sort().show() # not possible
df.sort("age").show()
df.sort(df["age"].desc()).show()
```

Asc() - default

```
+---+------+---------+-------+---+
| id|  name|     city|country|age|
+---+------+---------+-------+---+
|  2|Karthi|    Hosur|  India| 23|
|  4|  Ravi|Bangalore|  India| 25|
|  5| Kumar|Bangalore|  India| 25|
|  1|  Arun|  Chennai|  India| 30|
|  6| Vijay|  Chennai|  India| 30|
|  3| Kavin|    Salem|  India| 31|
+---+------+---------+-------+---+
```

```
+---+------+---------+-------+---+
| id|  name|     city|country|age|
+---+------+---------+-------+---+
|  3| Kavin|    Salem|  India| 31|
|  6| Vijay|  Chennai|  India| 30|
|  1|  Arun|  Chennai|  India| 30|
|  5| Kumar|Bangalore|  India| 25|
|  4|  Ravi|Bangalore|  India| 25|
|  2|Karthi|    Hosur|  India| 23|
+---+------+---------+-------+---+
```

10) describe() – returns summary statistics for numeric columns (count, mean, stddev, min, max)

```
df.describe().show()
```

```
+-------+------------------+-----+---------+-------+------------------+
|summary|                id| name|     city|country|               age|
+-------+------------------+-----+---------+-------+------------------+
|  count|                 6|    6|        6|      6|                 6|
|   mean|               3.5| NULL|     NULL|   NULL|27.333333333333332|
| stddev|1.8708286933869707| NULL|     NULL|   NULL| 3.386246693120078|
|    min|                 1| Arun|Bangalore|  India|                23|
|    max|                 6|Vijay|    Salem|  India|                31|
+-------+------------------+-----+---------+-------+------------------+
```

11) columns – returns a list of column names in the dataframe (this no need ())

```
print(df.columns)
# df.columns.show()
```

```
['id', 'name', 'city', 'country', 'age']
```

withcolumn()

In PySpark, the terms “withcolumn” used to add a new column or update an existing column in a DataFrame.

It takes two arguments:

1. Column name(string)
2. Expression or column logic

df.withColumn("new_column", some_expression)

```
from pyspark.sql.functions import col
df = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["id", "name"])
df2 = df.withColumn( colName: "id_plus_10", col("id") + 10)
df2.show()
```

```
+---+-----+----------+
| id| name|id_plus_10|
+---+-----+----------+
|  1|Alice|        11|
|  2|  Bob|        12|
+---+-----+----------+
```

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

data = [("raja", 25), ("ravi", 28)]
df = spark.createDataFrame(data, ["name", "age"])

# Add/replace multiple columns at once
df2 = df.withColumns({
    "country": lit("India"),          # new column
    "age": col("age") + 5,            # replace existing
    "is_adult": col("age") >= 18      # new column with expression
})

df2.show()
```

```
print("column rename")
df2 = df2.withColumnRenamed( existing: "id_plus_10", new: "new_employee_id")
df2.show()

print("drop column")
df2 = df2.drop("new_employee_id")
df2.show()

print("Alias name")
df.select(col('name').alias("employee_name")).show()
```

from pyspark.sql.functions import col, when

```
df3 = df.withColumn( colName: "level",
                    when(col("age")> 35,  value: "senior")
                    .when(col("age")<=34,  value: "junior")
                    .when(col("age").isNull(), value: "Null Record")
                    .otherwise("Unknown"))
df3.show()
```

```
+---+----+---+------+
| id|name|age| level|
+---+----+---+------+
|  1| AAA| 20|junior|
|  2| BBB| 40|senior|
|  3| CCC| 50|senior|
+---+----+---+------+
```

Function	Purpose	Use Case
show()	Display rows	View sample data
collect()	Get all rows to driver	Use only for small data
take(n)	Get first n rows	Sampling/debug
printSchema()	Show schema	Understand structure
count()	Count rows	Summary/validation
select()	Select columns	Column subset
filter()	Filter rows by condition	Subset rows
like()	SQL-style pattern match	String search
sort()	Sort rows	Ranking/ordering
describe()	Summary stats	Quick analysis
columns	List column names	Looping, validation

String functions

1) upper() – converts a string column to uppercase
2) lower() – coverts a string column to lowercase

```
df.select(upper(col("name"))).show()
df.select(lower(col("country"))).show()
```

```
+------------+
| upper(name)|
+------------+
|     ARUN   |
|      KARTHI|
|       KAVIN|
|        RAVI|
|       KUMAR|
|       VIJAY|
+------------+
```

```
+--------------+
|lower(country)|
+--------------+
|         india|
|         india|
|         india|
|         india|
|         india|
|         india|
+--------------+
```

3) initcap()

```
print("init cap case")
df.select(initcap(col("email"))).show() #first letter of word cap
```

```
+--------------+
|initcap(email)|
+--------------+
|Arun@gmail.com|
|Arun@gmail.com|
|Arun@gmail.com|
|Arun@gmail.com|
|Arun@gmail.com|
|Arun@gmail.com|
+--------------+
```

4) length() – return the length of the string

```
df.select(length(col("name"))).show()
```

```
+------------+
|length(name)|
+------------+
|          12|
|           6|
|           5|
|           4|
|           5|
|           5|
+------------+
```

5) trim() – removes both leading and trailing spaces from a string
6) ltrim() – removes leading (left-side) spaces from a string.
7) rtrim() – removes trailing (right-side) spaces from a string

```
df.select(trim("name")).show()
df.select(ltrim("name")).show()
df.select(rtrim(col("name"))).show()
```

```
+----------+
|trim(name)|
+----------+
|      Arun|
|    Karthi|
|     Kavin|
|      Ravi|
|     kumar|
|     vijay|
+----------+
```

```
+-----------+
|ltrim(name)|
+-----------+
|    Arun   |
|     Karthi|
|      Kavin|
|       Ravi|
|      kumar|
|      vijay|
+-----------+
```

```
+------------+
|rtrim(name)|
+------------+
|        Arun|
|      Karthi|
|       Kavin|
|        Ravi|
|       kumar|
|       vijay|
+------------+
```

8) substring() – extracts a substring from a string using position and length.

- **str** → Column containing the string.
- **pos** → Starting position (**1-based index**).
- **len** → Number of characters to return.

```
df.select(substring(col("name"), pos: 1, len: 3)).show()  # First 3 characters

df.select(initcap(col("name"))).show()
```

```
+--------------------+
|substring(name, 1, 3)|
+--------------------+
|                    |
|                 Kar|
|                 Kav|
|                 Rav|
|                 kum|
|                 vij|
+--------------------+
```

9) substring_index() – Returns part of a string before/after a delimiter.
substring_index(col, delim, n)
Positive n → before nth occurrence; Negative n → after nth.

```
df.select(substring_index(col("email"), delim: "@", count: 1)).show()  # username part
```

```
+---------------------------+
|substring_index(email, @, 1)|
+---------------------------+
|                       arun|
|                       arun|
|                       arun|
|                       arun|
|                       arun|
|                       arun|
+---------------------------+
```

10) split() – splits a string into an array using a delimiter

```
df.select(split(col("email"), pattern: "@")).show()
```

```
+------------------+
|split(email, @, -1)|
+------------------+
|  [arun, gmail.com]|
|  [arun, gmail.com]|
|  [arun, gmail.com]|
|  [arun, gmail.com]|
|  [arun, gmail.com]|
|  [arun, gmail.com]|
+------------------+
```

11) repeat() – repeats a string “n” times

```
df.select(repeat(col("name"), n: 2)).show()
```

```
+--------------------+
|     repeat(name, 2)|
+--------------------+
|     Arun        ...|
|        KarthiKarthi|
|          KavinKavin|
|            RaviRavi|
|          kumarkumar|
|          vijayvijay|
+--------------------+
```

12) rpad() – right-pads a string with a specified character to a desired length

13) lpad() -- left-pads a string

```
df.select(rpad(col("name"), len: 10, pad: "*")).show()
df.select(lpad(col("name"), len: 10, pad: "*")).show()
```

```
+-----------------+
|rpad(name, 10, *)|
+-----------------+
|            Arun |
|       Karthi****|
|       Kavin*****|
|       Ravi******|
|       kumar*****|
|       vijay*****|
+-----------------+
```

```
+-----------------+
|lpad(name, 10, *)|
+-----------------+
|            Arun |
|       ****Karthi|
|       *****Kavin|
|       ******Ravi|
|       *****kumar|
|       *****vijay|
+-----------------+
```

14) regexp_extract() – extracts a regex pattern group from a string.

```
df.select(regexp_extract(col("email"), pattern: '([a-zA-Z0-9._%+-]+)', idx: 1)).show()
```

```
+-----------------------------------------------+
|regexp_extract(email, ([a-zA-Z0-9._%+-]+), 1)|
+-----------------------------------------------+
|                                           arun|
|                                           arun|
|                                           arun|
|                                           arun|
|                                           arun|
|                                           arun|
+-----------------------------------------------+
```

15) Regexp_replace() – replace substrings matching a regex pattern.

```
df.select(regexp_replace(col("city"), pattern: "-", replacement: "")).show()
```

```
+---------------------------+
|regexp_replace(city, -, , 1)|
+---------------------------+
|                 ChennaiTN|
|                   Hosurtn|
|                   Salemtn|
|                Bangaloreka|
|                Bangaloreka|
|                   Chennaitn|
+---------------------------+
```

Numeric Functions

SUM()
AVG()
MIN()
MAX()
ROUND()
ABS()

```
data = [
    (1,-10001.5),
    (2,-10000.7),
    (3,20000.1),
    (4,50000.6),
    (5,30000.3)
]

column = ["id", "salary"]
```

```
numeric = [sum, max, min, avg, round, abs]

def numeric_function(function_list):  1 usage  new *
    for current_function in function_list:
            df.select(current_function("salary").alias(f"{current_function}")).show()

numeric_function(numeric)
```

```
+-----------------------------------+
|<function sum at 0x0000020BAC9A5760>|
+-----------------------------------+
|                            79998.8|
+-----------------------------------+

+-----------------------------------+
|<function max at 0x0000020BAC9A5120>|
+-----------------------------------+
|                            50000.6|
+-----------------------------------+

+-----------------------------------+
|<function min at 0x0000020BAC9A5260>|
+-----------------------------------+
|                           -10001.5|
+-----------------------------------+
```

```
+-----------------------------------+
|<function avg at 0x0000020BAC9A58A0>|
+-----------------------------------+
|                           15999.76|
+-----------------------------------+

+---------------------------------------+
|<function round at 0x0000020BAC9C2A20>|
+---------------------------------------+
|                             -10002.0|
|                             -10001.0|
|                              20000.0|
|                              50001.0|
|                              30000.0|
+---------------------------------------+
```

```
+-----------------------------------+
|<function abs at 0x0000020BAC9A4EA0>|
+-----------------------------------+
|                            10001.5|
|                            10000.7|
|                            20000.1|
|                            50000.6|
|                            30000.3|
+-----------------------------------+
```

Math function

```
data = [(1, 25.3), (2, -9.4), (3, 0.5)]
df = spark.createDataFrame(data, ["id", "value"])

df.withColumn( colName: "abs", abs("value")) \
  .withColumn( colName: "ceil", ceil("value")) \
  .withColumn( colName: "floor", floor("value")) \
  .withColumn( colName: "exp", exp("value")) \
  .withColumn( colName: "log", log("abs")) \
  .withColumn( colName: "power_2", pow( col1: "value",  col2: 2)) \
  .withColumn( colName: "sqrt", sqrt("abs")) \
  .show()
```

```
+---+-----+----+----+-----+--------------------+------------------+----------+------------------+
| id|value| abs|ceil|floor|                 exp|               log|   power_2|              sqrt|
+---+-----+----+----+-----+--------------------+------------------+----------+------------------+
|  1| 25.3|25.3|  26|   25|9.719644755919391E10| 3.2308043957334744|    640.09| 5.029910535983717|
|  2| -9.4| 9.4|  -9|  -10|8.272406555663223E-5| 2.2407096892759584|88.360000000000001|3.0659419433511785|
|  3|  0.5| 0.5|   1|    0|  1.6487212707001282|-0.6931471805599453|      0.25|0.7071067811865476|
+---+-----+----+----+-----+--------------------+------------------+----------+------------------+
```

Function	Description	Example Input → Output
abs()	Returns the **absolute value** (no negative sign)	-10 → 10, 5 → 5
ceil()	Rounds **up** to the nearest integer	4.3 → 5, 7.9 → 8
floor()	Rounds **down** to the nearest integer	4.8 → 4, 7.1 → 7
exp()	Returns **e to the power of x** (e ≈ 2.718)	1 → 2.718, 2 → 7.389 || exp(1) ≈ 2.718, exp(0) = 1
log()	Returns **natural log** (base e) of a number	e → 1, 1 → 0
pow(x, y)	Returns **x raised to the power y**	pow(2, 3) → 8, pow(5, 2) → 25
sqrt()	Returns the **square root** of a number	9 → 3, 16 → 4

Date and Time function

1) Parsing and converting dates

- **to_date(column, format):** convert a string to date using the specified format.
- **to_timestamp(column, format**): convert a string to timestamp.

```
# to_date
df = df.withColumn( colName: "order_date", to_date(col("order_date_string"),  format: "yyyy-MM-dd"))
df = df.withColumn( colName: "delivery_date", to_date(col("delivery_date"),  format: "yyyy-MM-dd"))
df.printSchema()
print("to_date: string to date convertion")
df.show()
```

withColumn overwrites the delivery_date

```
root
 |-- id: long (nullable = true)
 |-- order_date: string (nullable = true)
 |-- delivery_date: string (nullable = true)

root
 |-- id: long (nullable = true)
 |-- order_date: string (nullable = true)
 |-- delivery_date: date (nullable = true)
 |-- order_dates: date (nullable = true)
```

```
to_date: string to date convertion
+---+-----------------+-------------+----------+
| id|order_date_string|delivery_date|order_date|
+---+-----------------+-------------+----------+
|  1|       2024-09-23|   2024-11-20|2024-09-23|
|  2|       2023-10-05|   2023-10-15|2023-10-05|
|  3|       2022-05-01|   2022-05-10|2022-05-01|
+---+-----------------+-------------+----------+
```

to_timestamp

```
# to_timestamp
data = [("2024-07-30 14:25:00",), ("2024-07-30 09:05:00",)]
df = spark.createDataFrame(data, ["datetime_str"])

df = df.withColumn( colName: "timestamp_col", to_timestamp("datetime_str", "yyyy-MM-dd HH:mm:ss"))
```

```
+-------------------+-------------------+
|       datetime_str|      timestamp_col|
+-------------------+-------------------+
|2024-07-30 14:25:00|2024-07-30 14:25:00|
|2024-07-30 09:05:00|2024-07-30 09:05:00|
+-------------------+-------------------+
```

2) Formatting Dates

- **date_formate(date, format):** formats a date into a string with the specified pattern (eg: “yyyy-M-d”, “d/M/yyyy”)

```
#date formatting
print("date formating")
df.select( date_format(df["order_date"], format: "d/M/yy").alias("formatted date")).show()
df = df.withColumn( colName: "format_date",date_format(df["order_date"], format: 'dd-MM-yy'))
df.show()
```

```
+--------------+
|formatted date|
+--------------+
|       23/9/24|
|       5/10/23|
|        1/5/22|
+--------------+
```

```
+---+-----------------+-------------+----------+-----------+
| id|order_date_string|delivery_date|order_date|format_date|
+---+-----------------+-------------+----------+-----------+
|  1|       2024-09-23|   2024-11-20|2024-09-23|   23-09-24|
|  2|       2023-10-05|   2023-10-15|2023-10-05|   05-10-23|
|  3|       2022-05-01|   2022-05-10|2022-05-01|   01-05-22|
+---+-----------------+-------------+----------+-----------+
```

3) Current Date and Time

- **current_date():** returns the current date.
- **current_timestamp():** return the current timestamp

```
df = df.withColumn( colName: "current date", current_date())
df = df.withColumn( colName: "current timestamp", current_timestamp())

df.show(truncate = False)
df.printSchema()
```

```
+---+----------------+-------------+------------+-------------------------+
|id |order_date_string|delivery_date|current date|current timestamp        |
+---+----------------+-------------+------------+-------------------------+
|1  |2024-09-23      |2024-11-20   |2025-07-30  |2025-07-30 16:44:26.485026|
|2  |2023-10-05      |2023-10-15   |2025-07-30  |2025-07-30 16:44:26.485026|
|3  |2022-05-01      |2022-05-10   |2025-07-30  |2025-07-30 16:44:26.485026|
+---+----------------+-------------+------------+-------------------------+
```

```
root
 |-- id: long (nullable = true)
 |-- order_date_string: string (nullable = true)
 |-- delivery_date: string (nullable = true)
 |-- current date: date (nullable = false)
 |-- current timestamp: timestamp (nullable = false)
```

4) Extracting parts of dates

- **year()**: Extracts the year from a date.
- **month()**: Extracts the month.
- **dayofmonth()**: Extracts the day of the month.
- **dayofweek()**: Returns the day of the week as an integer (1 = Sunday, 7 = Saturday).
- **weekofyear()**: Extracts the week number of the year.

```
data = [
    (1, "2024-09-23", "2024-11-20"),
    (2, "2023-10-05", "2023-10-15"),
    (3, "2022-05-01", "2022-05-10")
]
```

```
df.select(
    year("order_date").alias("extract_year"),
    weekofyear("order_date").alias("week_number"),
    dayofyear("order_date").alias("day_number")
).show()
```

```
+------------+-----------+----------+
|extract_year|week_number|day_number|
+------------+-----------+----------+
|        2024|         39|       267|
|        2023|         40|       278|
|        2022|         17|       121|
+------------+-----------+----------+
```

Month

```
#month
df.select(
    month("order_date").alias("extract_month"),
    dayofweek("order_date").alias("week_of_day")
).show()
```

```
+-------------+-----------+
|extract_month|week_of_day|
+-------------+-----------+
|            9|          2|
|           10|          5|
|            5|          1|
+-------------+-----------+
```

dayofweek – 1 = Sunday, 2 = Monday, … 7 = Saturday

day

```
df.select(
    day("order_date").alias("day")
).show()
```

```
+---+
|day|
+---+
| 23|
|  5|
|  1|
+---+
```

5) Date Arithmetic

- datediff(end, start) – return the number of days between two dates.
- date_add(date, days) – adds a specific number of days to a date.
- add_months(date, numMonths) – adds or sub months from a date.
- date_sub(date, days) – sub a specific number of days from a date.

```
df.select(
    datediff(df["delivery_date"], df["order_date"]).alias("days_diff")
).show()
```

```
+---------+
|days_diff|
+---------+
|       58|
|       10|
|        9|
+---------+
```

Aggregate Function

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    avg, mean, sum, min, max, count, countDistinct,
    first, last, collect_list, collect_set
)

# Initialize Spark session
spark = SparkSession.builder.appName("Aggregate Functions Example").getOrCreate()
```

.agg() is the only way to apply **multiple** aggregate functions in a single call after groupBy().

```
# Aggregate functions
print("Aggregate Results")
df.groupBy("department").agg(
    mean("salary").alias("mean_salary"),
    avg("salary").alias("avg_salary"),
    sum("salary").alias("total_salary"),
    min("salary").alias("min_salary"),
    max("salary").alias("max_salary"),
    count("salary").alias("salary_count"),
    countDistinct("salary").alias("distinct_salaries"),
    first("name").alias("first_employee"),
    last("name").alias("last_employee"),
    collect_list("name").alias("all_names"),
    collect_set("name").alias("unique_names")
).show(truncate=False)
```

```
Original Data:
+----------+----+------+
|department|name|salary|
+----------+----+------+
|        HR|  Aa| 30000|
|        HR|  Bb| 40000|
|        IT|  Dd| 50000|
|        IT|  Cc| 60000|
|        IT|  Ee| 50000|
|     Sales|  Ff| 45000|
|     Sales|  Gg| 45000|
|     Sales|  Hh| 45000|
+----------+----+------+
```

```
Aggregate Results
+----------+-----------------+-----------------+------------+----------+----------+------------+----------------+--------------+
|department|mean_salary      |avg_salary       |total_salary|min_salary|max_salary|salary_count|distinct_salaries|first_employee|
+----------+-----------------+-----------------+------------+----------+----------+------------+----------------+--------------+
|Sales     |45000.0          |45000.0          |135000      |45000     |45000     |3           |1               |Ff            |
|HR        |35000.0          |35000.0          |70000       |30000     |40000     |2           |2               |Aa            |
|IT        |53333.333333333336|53333.333333333336|160000      |50000     |60000     |3           |2               |Dd            |
+----------+-----------------+-----------------+------------+----------+----------+------------+----------------+--------------+
```

```
+-----------------+--------------+-------------+------------+------------+
|distinct_salaries|first_employee|last_employee|all_names   |unique_names|
+-----------------+--------------+-------------+------------+------------+
|1                |Ff            |Hh           |[Ff, Gg, Hh]|[Gg, Hh, Ff]|
|2                |Aa            |Bb           |[Aa, Bb]    |[Aa, Bb]    |
|2                |Dd            |Cc           |[Dd, Ee, Cc]|[Cc, Dd, Ee]|
+-----------------+--------------+-------------+------------+------------+
```

Function	Description	Example Usage
`mean()`	Returns the average of the values (same as `avg()`)	`df.agg(mean("salary")).show()`
`avg()`	Returns the average of the values	`df.groupBy("dept").agg(avg("salary")).show()`
`collect_list()`	Collects all values into a list (duplicates included)	`df.groupBy("dept").agg collect_list("name")).show()`
`collect_set()`	Collects all values into a set (removes duplicates)	`df.groupBy("dept").agg (collect_set("name")).show()`
`countDistinct()`	Counts the number of distinct (unique) values	`df.agg(countDistinct("salary")).show()`
`count()`	Counts the number of rows	`df.count()` or `df.groupBy("dept").agg(count("*"))`
`first()`	Returns the first value in the group/order (not sorted)	`df.groupBy("dept").agg(first("name")).show()`
`last()`	Returns the last value in the group/order (not sorted)	`df.groupBy("dept").agg(last("name")).show()`
`max()`	Returns the maximum value	`df.agg(max("salary")).show()`
`min()`	Returns the minimum value	`df.agg(min("salary")).show()`

Join

In PySpark, joins are used to combine rows from two or more DataFrames based on a common key or condition.

Join Type	Description
inner	Matches rows from both DataFrames.
left	All rows from the left DataFrame, with null for non-matches.
right	All rows from the right DataFrame, with null for non-matches.
outer	All rows from both DataFrames, with null for non-matches.
cross	Cartesian product of both DataFrames.
left_semi	This is just an inner join of the two DataFrames, but only returns columns of left table DataFrame
left_anti	Rows from the left table where no match exists in the right.

Syntax:
DataFrame1.join(DataFrame2, DataFrame1.key == DataFrame2.key, "join_type")

```
employee dataframe
+------+------+---------------+-----------+-----------+------+------+
|emp_id|  name|superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+------+---------------+-----------+-----------+------+------+
|     1|   Red|              1|       2018|         10|     M|  3000|
|     2| Green|              2|       2010|         20|     M|  4000|
|     3|Yellow|              1|       2010|         10|     M|  1000|
|     4| White|              2|       2005|         10|     F|  2000|
|     5| Black|              2|       2010|         40|      |    -1|
|     6| Brown|              2|       2010|         50|      |    -1|
+------+------+---------------+-----------+-----------+------+------+
```

```
department dataframe
+----------+-------+
|department|dept_id|
+----------+-------+
|   Finance|     10|
| Marketing|     20|
|     Sales|     30|
|        IT|     40|
+----------+-------+
```

Inner join – Matches rows from two dataframe

```
#inner join
print("Inner join")
df1.join(df2, df1.emp_dept_id == df2.dept_id, how: 'inner').show()
```

```
Inner join
+------+------+--------------+-----------+-----------+------+------+----------+-------+
|emp_id|  name|superior_emp_id|year_joined|emp_dept_id|gender|salary|department|dept_id|
+------+------+--------------+-----------+-----------+------+------+----------+-------+
|     1|   Red|             1|       2018|         10|     M|  3000|   Finance|     10|
|     3|Yellow|             1|       2010|         10|     M|  1000|   Finance|     10|
|     4| White|             2|       2005|         10|     F|  2000|   Finance|     10|
|     2| Green|             2|       2010|         20|     M|  4000| Marketing|     20|
|     5| Black|             2|       2010|         40|      |    -1|        IT|     40|
+------+------+--------------+-----------+-----------+------+------+----------+-------+
```

Left join

```
#left, left_outer, leftouter
print("left join")
df1.join(df2, df1.emp_dept_id == df2.dept_id, how: 'left').show()
```

```
left join
+------+------+--------------+-----------+-----------+------+------+----------+-------+
|emp_id|  name|superior_emp_id|year_joined|emp_dept_id|gender|salary|department|dept_id|
+------+------+--------------+-----------+-----------+------+------+----------+-------+
|     1|   Red|             1|       2018|         10|     M|  3000|   Finance|     10|
|     2| Green|             2|       2010|         20|     M|  4000| Marketing|     20|
|     3|Yellow|             1|       2010|         10|     M|  1000|   Finance|     10|
|     4| White|             2|       2005|         10|     F|  2000|   Finance|     10|
|     5| Black|             2|       2010|         40|      |    -1|        IT|     40|
|     6| Brown|             2|       2010|         50|      |    -1|      NULL|   NULL|
+------+------+--------------+-----------+-----------+------+------+----------+-------+
```

Right join

```
#right, rightouter, right_outer
print("right join")
df1.join(df2, df1.emp_dept_id == df2.dept_id, how: 'right').show()
```

```
right join
+------+------+--------------+-----------+-----------+------+------+----------+-------+
|emp_id|  name|superior_emp_id|year_joined|emp_dept_id|gender|salary|department|dept_id|
+------+------+--------------+-----------+-----------+------+------+----------+-------+
|     4| White|             2|       2005|         10|     F|  2000|   Finance|     10|
|     3|Yellow|             1|       2010|         10|     M|  1000|   Finance|     10|
|     1|   Red|             1|       2018|         10|     M|  3000|   Finance|     10|
|     2| Green|             2|       2010|         20|     M|  4000| Marketing|     20|
|  NULL|  NULL|          NULL|       NULL|       NULL|  NULL|  NULL|     Sales|     30|
|     5| Black|             2|       2010|         40|      |    -1|        IT|     40|
+------+------+--------------+-----------+-----------+------+------+----------+-------+
```

Left semi join

```
#left semi --> semi, leftsemi, left_semi
print("left semi join")
df1.join(df2, df1.emp_dept_id == df2.dept_id, how: "semi").show()
```

```
left semi join
+------+------+--------------+-----------+-----------+------+------+
|emp_id|  name|superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+------+--------------+-----------+-----------+------+------+
|     1|   Red|             1|       2018|         10|     M|  3000|
|     3|Yellow|             1|       2010|         10|     M|  1000|
|     4| White|             2|       2005|         10|     F|  2000|
|     2| Green|             2|       2010|         20|     M|  4000|
|     5| Black|             2|       2010|         40|      |    -1|
+------+------+--------------+-----------+-----------+------+------+
```

Left anti join

```
# left anti --> anti, leftanti, left_anti
print("left anti join")
df1.join(df2, df1.emp_dept_id == df2.dept_id, how: "anti").show()
```

```
left anti join
+------+-----+--------------+-----------+-----------+------+------+
|emp_id| name|superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+-----+--------------+-----------+-----------+------+------+
|     6|Brown|             2|       2010|         50|      |    -1|
+------+-----+--------------+-----------+-----------+------+------+
```

```
df_join= df_join.fillna({"salary":0})
df_join = df_join.groupBy("department").agg(sum("salary").alias("total_salary"))
df_join.show()
```

df = df.fillna({"salary": 0, "gender": "Unknown"})
df = df.dropna() # drop rows with any null
df = df.dropna(subset=["salary"]) # drop rows only if salary is null

Conversion Function: cast() in pyspark
The cast() function in pyspark is used to convert a column from one data type to another.

Syntax:
df.withColumn("new_column", col("old_column").cast("new_type"))

or using sql

df.selectExpr("CAST(old_column AS new_type)")

From	To (Examples)
String	"int", "float", "double", "boolean", "timestamp", "date"
Integer	"string", "float", "double"
Date/Time	"string", "timestamp", "date"

```
data = [("1","1000.5"),("2","2000.9")]
columns = ["id","salary"]

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

df_casted = df.withColumn( colName: "salary_float", col("salary").cast("float"))
df_casted = df_casted.withColumn( colName: "salary_int", col("salary_float").cast("int"))
df_casted.printSchema()
df_casted.show()
```

Window function

Window functions in PySpark allow you to perform computations over subsets of rows related to the current row, grouped by a specific partition and ordered within that partition.
These functions are conceptually like SQL window functions, offering powerful tools for ranking, aggregation, and analytical computations.

To define a window in PySpark, you use the Window specification, which includes:

1. **partitionBy**: Similar to SQL's PARTITION BY, it divides data into groups based on one or more columns.
2. **orderBy**: Similar to SQL's ORDER BY, it orders the rows within each partition.

```
window_spec = Window.partitionBy("Category").orderBy("Internal ID")
df_spec = df.withColumn( colName: "rank", rank().over(window_spec))
df_spec.show(10)
```

```
+--------------------+-----+--------+-----+-------------+-----------+----------+------------+-----------+----+
|            Category|Price|Currency|Stock|          EAN|      Color|      Size|Availability|Internal ID|rank|
+--------------------+-----+--------+-----+-------------+-----------+----------+------------+-----------+----+
|Accessories (Bags...|  206|     USD|  677|8282848236311|MidnightBlue|   8x10 in|discontinued|          6|   1|
|          Automotive|   12|     USD|  439|3997003361528|       Plum|        XS|   pre_order|         40|   1|
|          Automotive|  159|     USD|  584|9883725074294|MediumOrchid|   8x10 in|   pre_order|         50|   2|
|          Automotive|  350|     USD|  756|3486711401836|       Snow|        XL|   pre_order|         61|   3|
|          Automotive|  279|     USD|  818|1941032767914|  Gainsboro|       XXL|   pre_order|         66|   4|
|          Automotive|  982|     USD|   47|7887280730963|  OliveDrab|        XL|out_of_stock|         79|   5|
|          Automotive|  748|     USD|  326|6855062072649|     Sienna|         S|out_of_stock|         81|   6|
|Beauty & Personal...|  293|     USD|  700|7935749142557| GhostWhite|       XXL|out_of_stock|          5|   1|
|Beauty & Personal...|  289|     USD|  727|9265397743935|      White|Extra Large|   backorder|         25|   2|
|Beauty & Personal...|  408|     USD|  922|8183313115592|  Chocolate|    Medium|   backorder|         34|   3|
+--------------------+-----+--------+-----+-------------+-----------+----------+------------+-----------+----+
```

Function	Description	Example
rank	Assigns a rank to each row, leaving gaps for ties.	df.withColumn("rank", rank().over(windowSpec)) Output: 1, 2, 2, 4 for tied rows.
dense_rank	Assigns a rank to each row without leaving gaps.	df.withColumn("dense_rank", dense_rank().over(windowSpec)) Output: 1, 2, 2, 3 for tied rows.
row_number	Assigns a unique sequential number to each row.	df.withColumn("row_number", row_number().over(windowSpec)) Output: 1, 2, 3, 4.

Dense_rank()

```
print("dense_rank")
df_spec = df.withColumn( colName: "dense_rank", dense_rank().over(window_spec))
df_spec.show(10)
```

```
+------------------+-----+--------+-----+-------------+------------+-----------+------------+-----------+----------+
|          Category|Price|Currency|Stock|          EAN|       Color|       Size|Availability|Internal ID|dense_rank|
+------------------+-----+--------+-----+-------------+------------+-----------+------------+-----------+----------+
|cessories (Bags...|  206|     USD|  677|8282848236311|MidnightBlue|    8x10 in|discontinued|          6|         1|
|       Automotive|   12|     USD|  439|3997003361528|        Plum|         XS|   pre_order|         40|         1|
|       Automotive|  159|     USD|  584|9883725074294|MediumOrchid|    8x10 in|   pre_order|         50|         2|
|       Automotive|  350|     USD|  756|3486711401836|        Snow|         XL|   pre_order|         61|         3|
|       Automotive|  279|     USD|  818|1941032767914|   Gainsboro|        XXL|   pre_order|         66|         4|
|       Automotive|  982|     USD|   47|7887280730963|   OliveDrab|         XL|out_of_stock|         79|         5|
|       Automotive|  748|     USD|  326|6855062072649|      Sienna|          S|out_of_stock|         81|         6|
|auty & Personal...|  293|     USD|  700|7935749142557|  GhostWhite|        XXL|out_of_stock|          5|         1|
|auty & Personal...|  289|     USD|  727|9265397743935|       White|Extra Large|   backorder|         25|         2|
|auty & Personal...|  408|     USD|  922|8183313115592|   Chocolate|     Medium|   backorder|         34|         3|
+------------------+-----+--------+-----+-------------+------------+-----------+------------+-----------+----------+
```

Row_number()

```
print("row number")
df_spec = df.withColumn( colName: "row number", row_number().over(window_spec))
df_spec.show(10)
```

```
+------------------+-----+--------+-----+-------------+------------+-----------+------------+-----------+----------+
|          Category|Price|Currency|Stock|          EAN|       Color|       Size|Availability|Internal ID|row number|
+------------------+-----+--------+-----+-------------+------------+-----------+------------+-----------+----------+
|cessories (Bags...|  206|     USD|  677|8282848236311|MidnightBlue|    8x10 in|discontinued|          6|         1|
|       Automotive|   12|     USD|  439|3997003361528|        Plum|         XS|   pre_order|         40|         1|
|       Automotive|  159|     USD|  584|9883725074294|MediumOrchid|    8x10 in|   pre_order|         50|         2|
|       Automotive|  350|     USD|  756|3486711401836|        Snow|         XL|   pre_order|         61|         3|
|       Automotive|  279|     USD|  818|1941032767914|   Gainsboro|        XXL|   pre_order|         66|         4|
|       Automotive|  982|     USD|   47|7887280730963|   OliveDrab|         XL|out_of_stock|         79|         5|
|       Automotive|  748|     USD|  326|6855062072649|      Sienna|          S|out_of_stock|         81|         6|
|auty & Personal...|  293|     USD|  700|7935749142557|  GhostWhite|        XXL|out_of_stock|          5|         1|
|auty & Personal...|  289|     USD|  727|9265397743935|       White|Extra Large|   backorder|         25|         2|
|auty & Personal...|  408|     USD|  922|8183313115592|   Chocolate|     Medium|   backorder|         34|         3|
+------------------+-----+--------+-----+-------------+------------+-----------+------------+-----------+----------+
```

- **This works for aggregation functions** like sum(), count(), etc.
- But for **ranking or row-based functions**, it's **incomplete**, because **orderBy() is required** to define how to rank/order rows.
- Without orderBy(), you'll get an **error or unpredictable results** with functions like rank(), row_number(), lead(), etc.

Only partition (wrong for rank)
Why partitionBy() alone isn't enough (Not able to use the partitionBy separately)
Without an orderBy(), Spark has **no criteria** to decide **which row is "first" or "best"**.
So, for functions that depend on **row positions**, like rank(), **you must define the order** explicitly.

Rank() as separate

```
data = [
    ("Electronics", "phone", 1000),
    ("Electronics", "Laptop", 1500),
    ("Electronics", "Tablet", 800),
    ("Furniture", "Chair", 300),
    ("Furniture", "Table", 300),
    ("Furniture", "Desk", 600)
]
```

```
window_specs = Window.orderBy("price")
df_window = df\
    .withColumn( colName: "rank", rank().over(window_specs)) \
    .withColumn( colName: "dense rank", dense_rank().over(window_specs)) \
    .withColumn( colName: "row number", row_number().over(window_specs))

df_window.show()
```

Will give warning message for only using the orderBy()

```
|   category|product|price|rank|dense rank|row number|
+-----------+-------+-----+----+----------+----------+
|  Furniture|  Chair|  300|   1|         1|         1|
|  Furniture|  Table|  300|   1|         1|         2|
|  Furniture|   Desk|  600|   3|         2|         3|
|Electronics| Tablet|  800|   4|         3|         4|
|Electronics|  phone| 1000|   5|         4|         5|
|Electronics| Laptop| 1500|   6|         5|         6|
+-----------+-------+-----+----+----------+----------+
```

PartitionBy with orderby

```
window_specs = Window.partitionBy("category").orderBy("price")
df_window = df\
    .withColumn( colName: "rank", rank().over(window_specs)) \
    .withColumn( colName: "dense rank", dense_rank().over(window_specs)) \
    .withColumn( colName: "row number", row_number().over(window_specs))
```

```
+-----------+-------+-----+----+----------+----------+
|   category|product|price|rank|dense rank|row number|
+-----------+-------+-----+----+----------+----------+
|Electronics| Tablet|  800|   1|         1|         1|
|Electronics|  phone| 1000|   2|         2|         2|
|Electronics| Laptop| 1500|   3|         3|         3|
|  Furniture|  Chair|  300|   1|         1|         1|
|  Furniture|  Table|  300|   1|         1|         2|
|  Furniture|   Desk|  600|   3|         2|         3|
+-----------+-------+-----+----+----------+----------+
```

Lead and Lag

Lead and lag functions are essential tools for analysing sequential data, particularly in scenarios where comparisons between current, previous, and next rows are needed.
These functions operate over **windows** and allow you to perform time-series analysis, trend detection, and other row-relative operations.

1. lead(column, offset, default)

- Fetches the value of a specified column from the **next row** within the same window.
- **Parameters**:
    - column: Column to fetch the value from.
    - offset: The number of rows to look ahead (default is 1).
    - default: Value to return if the offset is out of bounds.

Use Cases:

- Compare current and next values to identify growth or decline trends.
- Predict the next transaction or event for a user or entity.

2. lag(column, offset, default)

- Fetches the value of a specified column from the **previous row** within the same window.
- **Parameters**:
    - column: Column to fetch the value from.
    - offset: The number of rows to look back (default is 1).
    - default: Value to return if the offset is out of bounds.

Use Cases:

- Compare current and previous values for calculating differences or trends.
- Retrieve historical context for an entity or event.

Common use cases of lag and lead function

1. Calculate next and previous transactions amount
2. Calculate purchase difference
3. Predict future events
4. Fill missing data with defaults
5. Identify changes in status

```
df = df \
    .withColumn( colName: "next transaction",  F.lead( col: "amount", offset: 1).over(window_spec)) \
    .withColumn( colName: "previous transaction", F.lag( col: "amount",  offset: 1).over(window_spec))

df.show()
```

```
+-----------+----------+------+----------------+--------------------+
|customer_id|      date|amount|next transaction|previous transaction|
+-----------+----------+------+----------------+--------------------+
|          1|2024-01-01|   100|             200|                NULL|
|          1|2024-01-02|   200|             300|                 100|
|          1|2024-01-03|   300|            NULL|                 200|
|          2|2024-01-01|    50|              80|                NULL|
|          2|2024-01-02|    80|             120|                  50|
|          2|2024-01-03|   120|            NULL|                  80|
+-----------+----------+------+----------------+--------------------+
```

2) Purchase difference

```
#purchase difference
df = df.withColumn( colName: "purchase difference", F.col("amount") - F.lag( col: "amount", offset: 1).over(window_spec))
df.show()
```

```
+-----------+----------+------+----------------+--------------------+-------------------+
|customer_id|      date|amount|next transaction|previous transaction|purchase difference|
+-----------+----------+------+----------------+--------------------+-------------------+
|          1|2024-01-01|   100|             200|                NULL|               NULL|
|          1|2024-01-02|   200|             300|                 100|                100|
|          1|2024-01-03|   300|            NULL|                 200|                100|
|          2|2024-01-01|    50|              80|                NULL|               NULL|
|          2|2024-01-02|    80|             120|                  50|                 30|
|          2|2024-01-03|   120|            NULL|                  80|                 40|
+-----------+----------+------+----------------+--------------------+-------------------+
```

3)Predict Future Events

df = df.withColumn("next_status", F.lead("status", 1).over(window_spec))

```
+------------+----------+----------+--------------+
| customer_id|      date|    status|   next_status|
+------------+----------+----------+--------------+
|           1|2024-01-01|   browsing|       carted|
|           1|2024-01-02|     carted|    purchased|
|           1|2024-01-03|  purchased|         null|
+------------+----------+----------+--------------+
```

4) Fill missing data with defaults

```
df = df.withColumn( colName: "missing value", F.lag( col: "amount", offset: 1,  default: 0).over(window_spec))
df.show()
```

```
|customer_id|      date|amount|next transaction|previous transaction|purchase difference|missing value|
+-----------+----------+------+----------------+--------------------+-------------------+-------------+
|          1|2024-01-01|   100|             200|                NULL|               NULL|            0|
|          1|2024-01-02|   200|             300|                 100|                100|          100|
|          1|2024-01-03|   300|            NULL|                 200|                100|          200|
|          2|2024-01-01|    50|              80|                NULL|               NULL|            0|
|          2|2024-01-02|    80|             120|                  50|                 30|           50|
|          2|2024-01-03|   120|            NULL|                  80|                 40|           80|
+-----------+----------+------+----------------+--------------------+-------------------+-------------+
```

5) Identify the status change

```
print("status change")
df = df.withColumn( colName: "status_change", F.col("status") != F.lag( col: "status",  offset: 1).over(window_spec))
df.show()
```

```
status change
+-----------+----------+------+---------+----------------+--------------------+-------------------+-------------+-------------+
|customer_id|      date|amount|   status|next transaction|previous transaction|purchase difference|missing value|status_change|
+-----------+----------+------+---------+----------------+--------------------+-------------------+-------------+-------------+
|          1|2024-01-01|   100| browsing|             200|                NULL|               NULL|            0|         NULL|
|          1|2024-01-02|   200| browsing|             300|                 100|                100|          100|        false|
|          1|2024-01-03|   300|   carted|            NULL|                 200|                100|          200|         true|
|          2|2024-01-01|    50|   carted|              80|                NULL|               NULL|            0|         NULL|
|          2|2024-01-02|    80|   carted|             120|                  50|                 30|           50|        false|
|          2|2024-01-03|   120|purchased|            NULL|                  80|                 40|           80|         true|
+-----------+----------+------+---------+----------------+--------------------+-------------------+-------------+-------------+
```

```
df.select("customer_id","amount","status","status_change").show()
```

```
status change
+-----------+------+---------+-------------+
|customer_id|amount|   status|status_change|
+-----------+------+---------+-------------+
|          1|   100| browsing|         NULL|
|          1|   200| browsing|        false|
|          1|   300|   carted|         true|
|          2|    50|   carted|         NULL|
|          2|    80|   carted|        false|
|          2|   120|purchased|         true|
+-----------+------+---------+-------------+
```

Array

Function	Description
array()	Create an array from columns
array_contains()	Check if a value exists in the array
size()	Get the number of elements in the array
array_position()	Get index of a value (1-based), 0 if not found
array_remove()	Remove all occurrences of a specific element

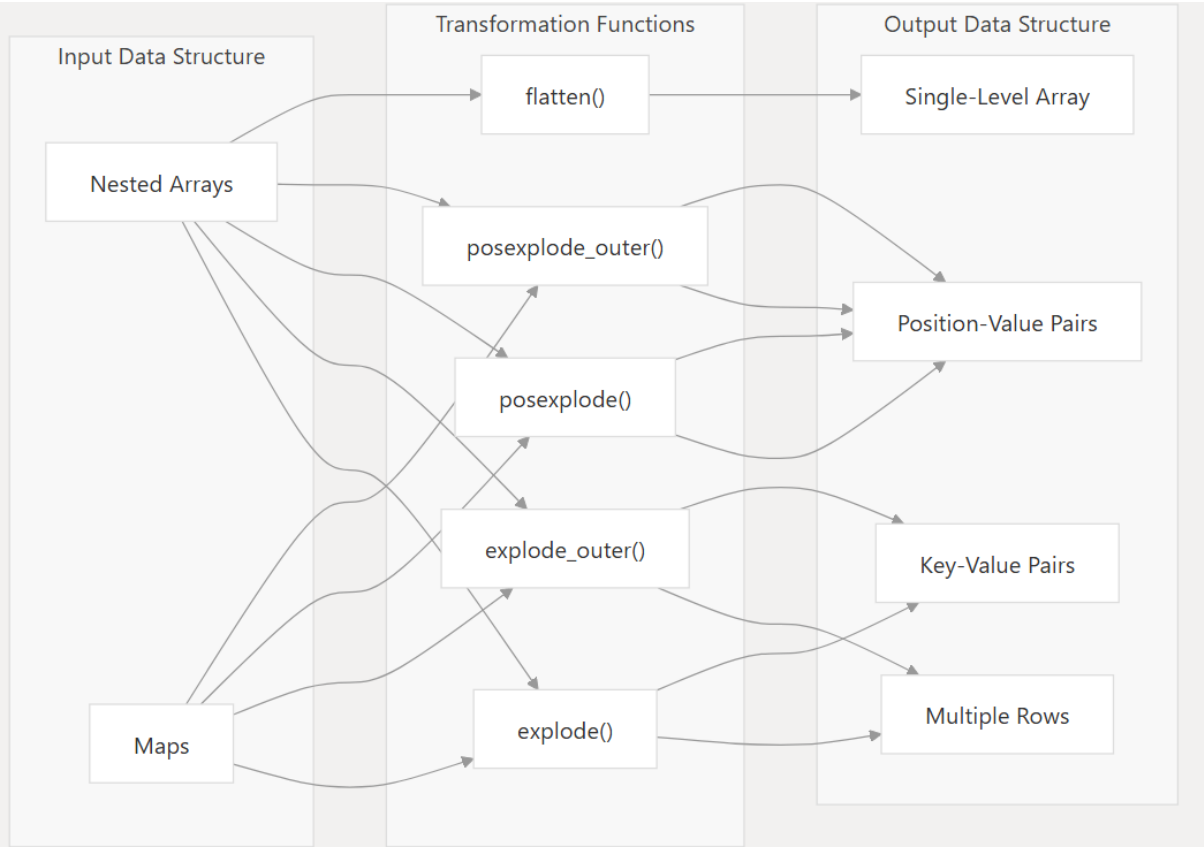
```
df = df \
    .withColumn( colName: "integrated_col", array("category","product"))\
    .withColumn( colName: "check_array_contains", array_contains(col("integrated_col"), value: "Tablet"))\
    .withColumn( colName: "size", size(col("integrated_col"))) \
    .withColumn( colName: "pos_of_phone", array_position(col("integrated_col"), value: "phone")) \
    .withColumn( colName: "array_removed", array_remove(col("integrated_col"), element: "phone"))

df.show(truncate=False)
```

```
+-----------+-------+-----+--------------------+--------------------+----+------------+--------------------+
|category   |product|price|integrated_col      |check_array_contains|size|pos_of_phone|array_removed       |
+-----------+-------+-----+--------------------+--------------------+----+------------+--------------------+
|Electronics|phone  |1000 |[Electronics, phone] |false               |2   |2           |[Electronics]       |
|Electronics|Laptop |1500 |[Electronics, Laptop]|false               |2   |0           |[Electronics, Laptop]|
|Electronics|Tablet |800  |[Electronics, Tablet]|true                |2   |0           |[Electronics, Tablet]|
|Furniture  |Chair  |300  |[Furniture, Chair]   |false               |2   |0           |[Furniture, Chair]  |
|Furniture  |Table  |300  |[Furniture, Table]   |false               |2   |0           |[Furniture, Table]  |
|Furniture  |Desk   |600  |[Furniture, Desk]    |false               |2   |0           |[Furniture, Desk]   |
+-----------+-------+-----+--------------------+--------------------+----+------------+--------------------+
```

Explode Array and Maps Function

explode functions in PySpark, which are super useful for flattening **arrays** and **maps** — commonly needed in JSON data or nested schemas.

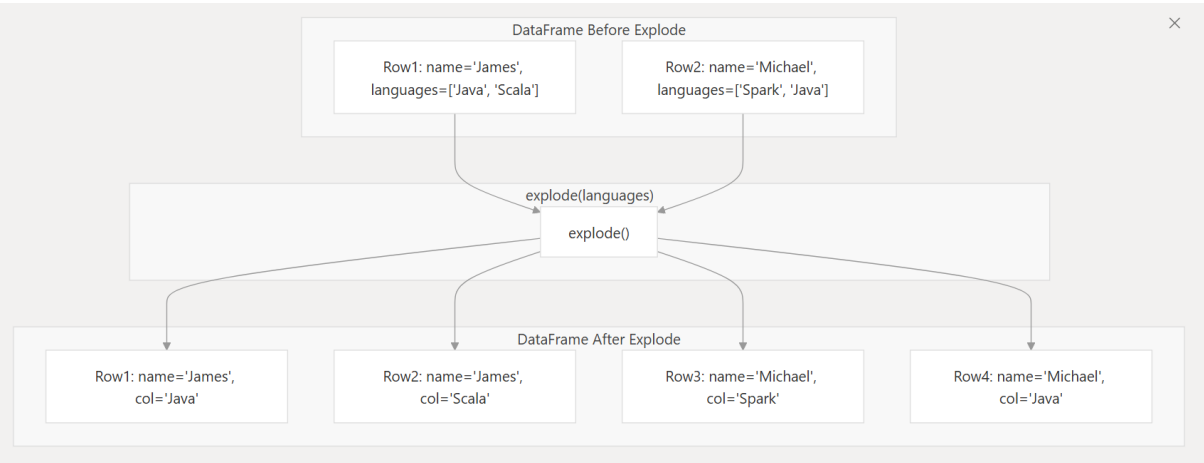
The **explode()** family of functions converts array elements or map entries into separate rows, while the **flatten()** function converts nested arrays into single-level arrays.



Explode Functions
The **explode()** function and its variants transform array or map columns by creating a new row for each element in the array or each key-value pair in the map.

The **explode()** function:

- Takes a single column containing an array or a map
- Returns a new row for each element in the array or each key-value pair in the map
- Skips null values and empty arrays/maps
- Is imported from **pyspark.sql.functions**



```
data = [
    (1, ["red", "blue", "green"]),
    (2, ["yellow", "black"]),
    (3, [])  # Empty array
]

df = spark.createDataFrame(data, ["id", "colors"])
df_exploded = df.select("id", explode("colors").alias("color"))
df_exploded.show()
```

```
+---+------+
| id| color|
+---+------+
|  1|   red|
|  1|  blue|
|  1| green|
|  2|yellow|
|  2| black|
+---+------+
```

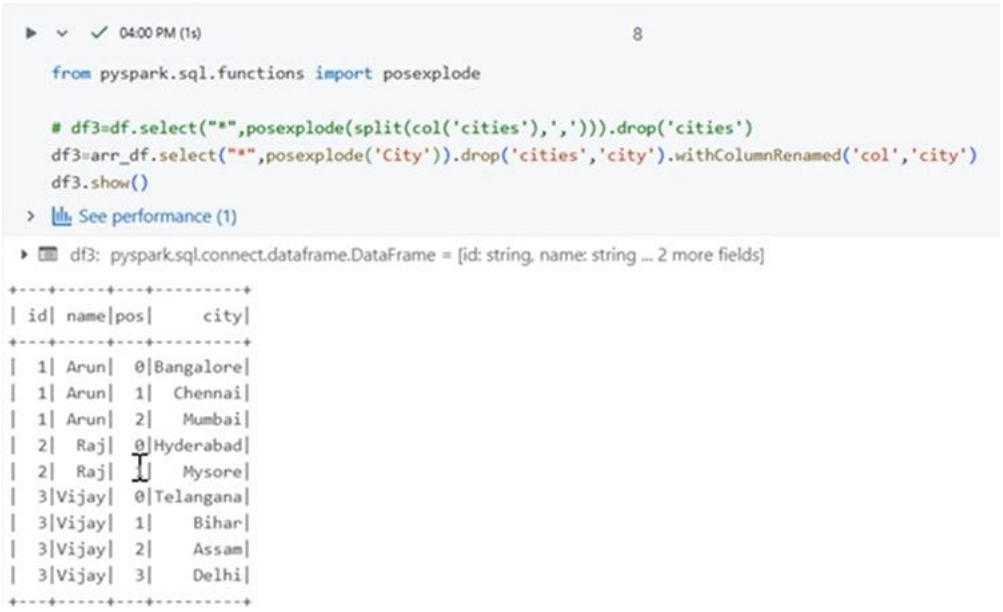
Row with empty array (id = 3) is **excluded**.

explode_outer()
Like explode(), but preserves rows even if the array/map is null or empty by returning **NULL**.

```
print("explode outer")
df_exploded_outer = df.select("id", explode_outer("colors").alias("exploded_outer"))
df_exploded_outer.show()
```

```
+---+--------------+
| id|exploded_outer|
+---+--------------+
|  1|           red|
|  1|          blue|
|  1|         green|
|  2|        yellow|
|  2|         black|
|  3|          NULL|
+---+--------------+
```

posexplode_outer()
Similar to explode_outer(), but adds **position (index)** of each element in the array.



```
✓ 04:00 PM (1s)                                                    8

from pyspark.sql.functions import posexplode

# df3=df.select("*",posexplode(split(col('cities'),',','))).drop('cities')
df3=arr_df.select("*",posexplode('City')).drop('cities','city').withColumnRenamed('col','city')
df3.show()

> See performance (1)

df3:  pyspark.sql.connect.dataframe.DataFrame = [id: string, name: string ... 2 more fields]

+---+-----+---+---------+
| id| name|pos|     city|
+---+-----+---+---------+
|  1| Arun|  0|Bangalore|
|  1| Arun|  1|  Chennai|
|  1| Arun|  2|   Mumbai|
|  2|  Raj|  0|Hyderabad|
|  2|  Raj|  1|   Mysore|
|  3|Vijay|  0|Telangana|
|  3|Vijay|  1|    Bihar|
|  3|Vijay|  2|    Assam|
|  3|Vijay|  3|    Delhi|
+---+-----+---+---------+
```

Why use lit()?
PySpark DataFrame operations work on **columns**, so if you want to add a constant value as a column or use it in expressions, you must **wrap it with lit()**.

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

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

data = [("Arun", 25), ("Kavin", 30)]
df = spark.createDataFrame(data, ["name", "age"])

# Add a constant column
df_with_country = df.withColumn("country", lit("India"))
df_with_country.show()
```

User Defined Function (UDF):

UDF allows you to define you own function in python and use it in pyspark just like a built-in function.

When to use UDF in pyspark?
1 Built-in function are not enough: If PySpark's SQL or Dataframe function (like withcolumn, select, filter,etc) don't support the transformation logic you need.
2 You need complex python logic: when you want to reuse existing python function, or logic is better expressed in regular python syntax.
For example, if you want to:

- Reverse a string
- Format a date in a custom way
- Categorize people based on complex business rules

When not to use UDF:
Built-in functions are much faster and optimized to run in spark's execution engine.
UDFs break the catalyst optimizer, so performance may degrade.

UDF is expensive operation, try to minimize the usage of UDF

```
spark = SparkSession.builder.appName("udf").getOrCreate()
df = spark.createDataFrame([("arun",38),("ravi",62)], ["name","age"])

#python function
def reverse_string(value):  1 usage  new *
    return value[::-1]

#convert to UDF
reverse_udf = udf(reverse_string, StringType())

df_reverse = df.withColumn( colName: "reversed_name", reverse_udf(df["name"]))
df_reverse.show()
```

Reverse_udf = udf(function, return type)

```
def age_category(age):  1 usage  new *
    if age<18:
        return 'child'
    elif 18 <= age < 50:
        return 'young'
    else:
        return 'senior'


age_udf = udf(age_category, StringType())
df_age = df.withColumn( colName: "category", age_udf(df["age"]))
df_age.show()
```

```
+----+---+--------+
|name|age|category|
+----+---+--------+
|arun| 38|   young|
|ravi| 62|  senior|
+----+---+--------+
```

```python
#---use decorator
@udf(returnType=StringType())  1 usage  new *
def age_category_udf(age):
    if age<18:
        return 'child'
    elif 18 <= age < 50:
        return 'young'
    else:
        return 'senior'


df_age_de = df.withColumn( colName: "category_de", age_category_udf(df["age"]))
df_age_de.show()
```

```
+----+---+-----------+
|name|age|category_de|
+----+---+-----------+
|arun| 38|      young|
|ravi| 62|     senior|
+----+---+-----------+
```

Different File formats with schema

Schema Definition: a schema defines the structure of your DataFrame — i.e., the column names, data types, and whether the columns can be null.

StructType:

- It is the container for a full schema.
- It holds multiple StructField (i.e., multiple columns)

StructField:

Describes an individual columns: name, data type, nullability

DataType:

Type	Description
StringType()	Text string
IntegerType()	Whole number
FloatType()	Decimal number
DoubleType()	Double-precision float
BooleanType()	True/False
DateType()	Date (yyyy-MM-dd)
TimestampType()	Date + time

Schema = StructType([StructField(“col name”, data type, can accept NULL or not (Boolean))])

```python
data1 = [
    ("James", "", "Smith", "36636", "M", 3000),
    ("Michael", "Rose", "", "40288", "M", 4000),
    ("Robert", "", "Williams", "42114", "M", 4000),
    ("Maria", "Anne", "Jones", "39192", "F", 4000),
    ("Jen", "Mary", "Brown", "", "F", -1)
]
schema1 = StructType([
    StructField( name: "firstname", StringType(),  nullable: False),
    StructField( name: "middlename", StringType(),  nullable: True),
    StructField( name: "lastname", StringType(),  nullable: True),
    StructField( name: "id", StringType(),  nullable: True),
    StructField( name: "gender", StringType(),  nullable: True),
    StructField( name: "salary", IntegerType(),  nullable: True)
])

basic_df = spark.createDataFrame(data=data1, schema=schema1)
basic_df.printSchema()
basic_df.show()
```

```
root
 |-- firstname: string (nullable = false)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

```
+---------+----------+--------+-----+------+------+
|firstname|middlename|lastname|   id|gender|salary|
+---------+----------+--------+-----+------+------+
|    James|          |   Smith|36636|     M|  3000|
|  Michael|      Rose|        |40288|     M|  4000|
|   Robert|          |Williams|42114|     M|  4000|
|    Maria|      Anne|   Jones|39192|     F|  4000|
|      Jen|      Mary|   Brown|     |     F|    -1|
+---------+----------+--------+-----+------+------+
```

Nested_df

```
Schema2 = StructType([
    StructField( name: "name", StructType([
        StructField( name: "firstname", StringType(),  nullable: False),
        StructField( name: "middlename", StringType(),  nullable: True),
        StructField( name: "lastname", StringType(),  nullable: True)
    ])),
    StructField( name: "id", StringType(),  nullable: True),
    StructField( name: "gender", StringType(),  nullable: True),
    StructField( name: "salary", IntegerType(),  nullable: True)
])

nested_df = spark.createDataFrame(data=data2, schema=Schema2)
nested_df.printSchema()
nested_df.show()
```

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = false)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

Array and map type

```
arrayStructureData = [
    (("James", "", "Smith"), ["music", "reading", "chess"], {"salary": "3000", "dept": "HR"}),
    (("Michael", "Rose", ""), ["music", "playing", "chess"], {"salary": "3000", "dept": "HR"}),
    (("Robert", "", "Williams"), ["music", "reading", "chess"], {"salary": "3000", "dept": "HR"}),
    (("Maria", "Anne", "Jones"), ["music", "reading", "chess"], {"salary": "3000", "dept": "HR"}),
    (("Jen", "Mary", "Brown"), ["music", "reading", "chess"], {"salary": "3000", "dept": "HR"})
]

Schema3 = StructType([
    StructField( name: "name", StructType([
        StructField( name: "firstname", StringType(),  nullable: True),
        StructField( name: "middlename", StringType(),  nullable: True),
        StructField( name: "lastname", StringType(),  nullable: True)
    ])),
    StructField( name: "hobbies", ArrayType(StringType()),  nullable: True),
    StructField( name: "properties", MapType(StringType(), StringType()),  nullable: True)
])
```

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- hobbies: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)

+--------------------+----------------------+----------------------------+
|name                |hobbies               |properties                  |
+--------------------+----------------------+----------------------------+
|{James, , Smith}    |[music, reading, chess]|{salary -> 3000, dept -> HR}|
|{Michael, Rose, }   |[music, playing, chess]|{salary -> 3000, dept -> HR}|
|{Robert, , Williams}|[music, reading, chess]|{salary -> 3000, dept -> HR}|
|{Maria, Anne, Jones}|[music, reading, chess]|{salary -> 3000, dept -> HR}|
|{Jen, Mary, Brown}  |[music, reading, chess]|{salary -> 3000, dept -> HR}|
+--------------------+----------------------+----------------------------+
```

Second Method of Creating Schema

Cmd 8

```
1  inlineSchema = "manufacturerName STRING, country STRING"
2
3  df = spark.read.format("csv").option("header",True).schema(inlineSchema).load("/FileStore/tables/manufacturers.csv
```

Create Schema Along with Dataframe

Cmd 12

```
1   df = spark.createDataFrame([
2     ("Mazda RX4",21,4,4),
3     ("Hornet 4 Drive",22,3,2),
4     ("Merc 240D",25,4,2),
5     ("Lotus Europa",31,5,2),
6     ("Ferrari Dino",20,5,6),
7     ("Volvo 142E",22,4,2)
8   ],["Car Name","mgp","gear","carb"])
9
10  df.display()
```

```
print(basic_df.schema)
```

```
StructType([StructField('firstname', StringType(), False), StructField('middlename', StringType(), True), StructField('lastname', StringType()
```

```
print("df.schema.json()")
print(basic_df.schema.json())
```

```
df.schema.json()
{"fields":[{"metadata":{},"name":"firstname","nullable":false,"type":"string"},{"metadata":{},"name":"middlename","nullable":true,"type":"string
```

```
#comparing schema
if basic_df.schema == nested_df.schema:
    print("schema match: do some operation")
else:
    print("perform any other operations")

print("To list the columns that where missing in nested_df")
print(list(set(basic_df.schema) - set(nested_df.schema)))
```

```
To list the columns that where missing
[StructField('middlename', StringType(), True), StructField('firstname', StringType(), False), StructField('lastname', StringType(), True)]
```

In order to perform **union operation**, the column should be match, in case there is mismatch in column as a result entire pipeline will be fails. To avoid that dynamically we can compare the schema between two dataframe in case there is a mismatch we have to add those missing columns with null value so that union will succeed.

```
allColumns = basic_df.columns + nested_df.columns
uniqueColumns = list(set(allColumns))
print(uniqueColumns)

from pyspark.sql.functions import lit

for col in uniqueColumns:
    if col not in basic_df.columns:
        basic_df = basic_df.withColumn(col, lit(None))
    if col not in nested_df.columns:
        nested_df = nested_df.withColumn(col, lit(None))

basic_df.show()
nested_df.show()
```

Basic_df

```
+---------+----------+--------+-----+------+------+----+
|firstname|middlename|lastname|   id|gender|salary|name|
+---------+----------+--------+-----+------+------+----+
|    James|          |   Smith|36636|     M|  3000|NULL|
|  Michael|      Rose|        |40288|     M|  4000|NULL|
|   Robert|          |Williams|42114|     M|  4000|NULL|
|    Maria|      Anne|   Jones|39192|     F|  4000|NULL|
|      Jen|      Mary|   Brown|     |     F|    -1|NULL|
+---------+----------+--------+-----+------+------+----+
```

Nested_df

```
+--------------------+-----+------+------+----------+---------+--------+
|                name|   id|gender|salary|middlename|firstname|lastname|
+--------------------+-----+------+------+----------+---------+--------+
|    {James, , Smith}|36636|     M|  3100|      NULL|     NULL|    NULL|
|   {Michael, Rose, }|40288|     M|  4300|      NULL|     NULL|    NULL|
|{Robert, , Williams}|42114|     M|  1400|      NULL|     NULL|    NULL|
|{Maria, Anne, Jones}|39192|     F|  5500|      NULL|     NULL|    NULL|
|  {Jen, Mary, Brown}|     |     F|    -1|      NULL|     NULL|    NULL|
+--------------------+-----+------+------+----------+---------+--------+
```

Reading Different File Formats with Schema

Using csv("path") or format("csv").load("path") of DataFrameReader, you can read a csv file into a pyspark Dataframe

```
df = spark.read.format("csv")
                .load("/tmp/resources/zipcodes.csv")
//      or
df = spark.read.format("org.apache.spark.sql.csv")
                .load("/tmp/resources/zipcodes.csv")
df.printSchema()
```

```
df2 = spark.read.option("header",True) \
     .csv("/tmp/resources/zipcodes.csv")
```

Handling the corrupt (bad) records

Option("mode", "")

1. **Permissive – include corrupt record in separate column.**
2. **Drop malformed – ignore corrupt records**
3. **Fail fast – throw exception if corrupt records.**

```
Df = spark.read.format("csv")
        .option("mode", "DROPMALFORMED")
        .option("header", "true")
        .schema(schema)
        .load(path)
```

Sample data

```
Month,Emp_count,Production_unit,Expense
JAN,340,2000,30,
FEB,318,2500,29,
MAR,362,1500,32,
APR,348,3000,26,
MAY,363,2200,35,test_msg
JUN,435,3300,27,
JUL,491,1600,23,
AUG,505,Thousand,33,
SEP,404,3500,36,
OCT,359,2900,28,
NOV,310,4000,25,
DEC,337,3400,31,
```

5th column is their

```
schema = StructType([ \
    StructField("Month",StringType(),True), \
    StructField("Emp_count",IntegerType(),True), \
    StructField("Production_unit",IntegerType(),True), \
    StructField("Expense",IntegerType(),True),\
    StructField("_corrupt_record", StringType(), True)
 ])
```

Permissive Mode

```
Cmd 7

df = spark.read.format("csv").option("mode", "PERMISSIVE").option("header", "true").schema(schema).load("/FileStore/tables/production_data_corrupt.csv")

display(df)
```

▸ (1) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [Month: string, Emp_count: integer ... 3 more fields]

	Month	Emp_count	Production_unit	Expense	_corrupt_record
5	MAY	363	2200	35	MAY,363,2200,35,test_msg
6	JUN	435	3300	27	null
7	JUL	491	1600	23	null
8	AUG	505	null	33	AUG,505,Thousand,33
9	SEP	404	3500	36	null
10	OCT	359	2900	28	null

Showing all 12 rows.

```
df = spark.read.format("csv").option("mode", "DROPMALFORMED").option("header", "true").schema(schema).load("/FileStore/tables/production_data_corrupt.csv")
display(df)
```

(1) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Month: string, Emp_count: integer ... 3 more fields]

	Month	Emp_count	Production_unit	Expense	_corrupt_record
1	JAN	340	2000	30	null
2	FEB	318	2500	29	null
3	MAR	362	1500	32	null
4	APR	348	3000	26	null
5	JUN	435	3300	27	null
6	JUL	491	1600	23	null
7	SEP	404	3500	36	null

Showing all 10 rows.

FailFast Mode

```
df = spark.read.format("csv").option("mode", "FAILFAST").option("header", "true").schema(schema).load("/FileStore/tables/production_data_corrupt.csv")
display(df)
```

(1) Spark Jobs

SparkException: Job aborted due to stage failure: Task 0 in stage 25.0 failed 1 times, most recent failure: Lost task 0.0 in stage 25.0 (TID 35) (ip-10-172-215-8
pute.internal executor driver): com.databricks.sql.io.FileReadException: Error while reading file dbfs:/FileStore/tables/production_data_corrupt.csv.

Command took 0.83 seconds -- by audaciousazure@gmail.com at 21/08/2021, 14:57:16 on My Cluster

Shift+Enter to run

```
df = spark.read \
    .format("csv") \
    .option("header", True) \
    .option("inferSchema", True) \
    .option("quote", "\"") \        # fields enclosed in quotes
    .option("escape", "\\") \       # escape character for quotes
    .option("nullValue", "NULL") \  # treat "NULL" as null
    .option("nanValue", "NaN") \    # treat "NaN" as NaN
    .load("/mnt/data/customers.csv")

df.show()
```

Method – 1

```
#method -1
df = spark.read.csv(path="C:\\Users\\GowdhamanBJ\\Downloads\\customers-100.csv", header=True)
df.printSchema()
df.show(5)
```

If header = True not mentioned, then column header is not display (display like co, c1)

If inferSchema not given, then data type is taken as string for all.

```
root
 |-- Index: string (nullable = true)
 |-- Customer Id: string (nullable = true)
 |-- First Name: string (nullable = true)
 |-- Last Name: string (nullable = true)
 |-- Company: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Phone 1: string (nullable = true)
 |-- Phone 2: string (nullable = true)
 |-- Email: string (nullable = true)
 |-- Subscription Date: string (nullable = true)
 |-- Website: string (nullable = true)
```

Inferschema = True

```
spark.read.csv(path="C:\\Users\\GowdhamanBJ\\Downloads\\customers-100.csv", header=True, inferSchema=True)
```

```
root
 |-- Index: integer (nullable = true)
 |-- Customer Id: string (nullable = true)
 |-- First Name: string (nullable = true)
 |-- Last Name: string (nullable = true)
 |-- Company: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Phone 1: string (nullable = true)
 |-- Phone 2: string (nullable = true)
 |-- Email: string (nullable = true)
 |-- Subscription Date: date (nullable = true)
 |-- Website: string (nullable = true)
```

Method – 2

```
#method - 2
df1 = (spark.read.format("csv")
       .option(key='header', value=True)
       .option(key='inferSchema', value=True)
       .load(path="C:\\Users\\GowdhamanBJ\\Downloads\\customers-100.csv"))
df1.show(4)
```

Multiple CSV file reading

```
df_multi = spark.read.csv(path=["C:\\Users\\GowdhamanBJ\\Downloads\\Employee-Q1.csv",
                                "C:\\Users\\GowdhamanBJ\\Downloads\\Department-Q1.csv"],
                          header=True, inferSchema=True)
df_multi.show()
```

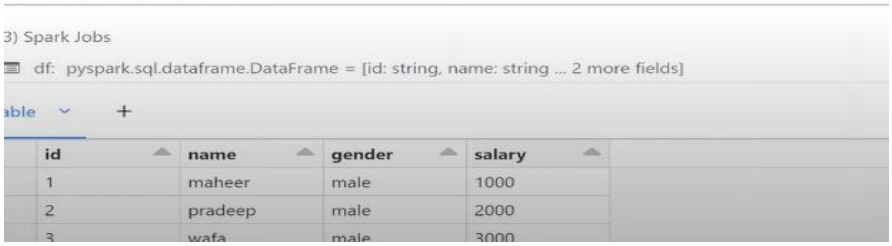
It combines the data of two csv based on the datatype

Example :
If csv file 1 → int, string, string, string, int, int
Csv file 2 → int, string
Means int type will merge with the int and string will merge with string,
For extra columns Null will be assigned.

```
|EmployeeID|EmployeeName|Department|Country|Salary| Age|
+----------+------------+----------+-------+------+----+
|         1|       James|      D101|     IN|  9000|  25|
|         2|      Michel|      D102|     SA|  8000|  26|
|         3|   James son|      D101|     IN| 10000|  35|
|         4|      Robert|      D103|     MY| 11000|  34|
|         5|       Scott|      D104|     MA|  6000|  36|
|         6|         Gen|      D105|     JA| 21345|  24|
|         7|        John|      D102|     MY| 87654|  40|
|         8|       Maria|      D105|     SA| 38144|  38|
|         9|       Soffy|      D103|     IN| 23456|  29|
|        10|         Amy|      D103|     CN| 21345|  24|
|      D101|       Sales|      NULL|   NULL|  NULL|NULL|
|      D102|   Marketing|      NULL|   NULL|  NULL|NULL|
|      D103|     Finance|      NULL|   NULL|  NULL|NULL|
|      D104|     Support|      NULL|   NULL|  NULL|NULL|
|      D105|          HR|      NULL|   NULL|  NULL|NULL|
+----------+------------+----------+-------+------+----+
```

By this we can read the folder and defined the schema.

```
from pyspark.sql.types import *

schema = StructType().add(field='id',data_type=IntegerType())\
                     .add(field='name',data_type=StringType())\
                     .add(field='gender',data_type=StringType())\
                     .add(field='salary',data_type=IntegerType())

df = spark.read.csv(path='dbfs:/FileStore/data', schema=schema, header=True)
display(df)
df.printSchema()
```



```python
#read csv file with the schema
# Define schema
csv_schema = StructType([
    StructField( name: "name", StringType(),  nullable: True),
    StructField( name: "age", IntegerType(),  nullable: True)
])

# Read CSV file with schema
df_csv = spark.read.csv( path: "path/to/file.csv", header=True, schema=csv_schema)
```

Reading Parquet

Parquet is a binary, open-source, and efficient file format designed for large-scale data processing. It is commonly used with big data frameworks like Apache Spark.
Data is stored column-wise instead of row-wise.

Parquet files **already store schema**, so you don't need to specify it manually.

🔹 Row vs Column Format Example:

Suppose you have this data:

ID	Name	Age
1	Alice	25
2	Bob	30

In Row format (like CSV):

csharp

```
[1, Alice, 25]
[2, Bob, 30]
```

In Column format (Parquet):

makefile

```
ID: [1, 2]
Name: [Alice, Bob]
Age: [25, 30]
```

```
#read Parquet file
# df_parquet = spark.read.parquet("C:\\Users\\GowdhamanBJ\\Downloads\\mtcars.parquet")
df_parquet = spark.read.format("parquet").load("C:\\Users\\GowdhamanBJ\\Downloads\\mtcars.parquet")
print("Parquet file")
df_parquet.printSchema()
df_parquet.show()
```

```
root
 |-- model: string (nullable = true)
 |-- mpg: double (nullable = true)
 |-- cyl: integer (nullable = true)
 |-- disp: double (nullable = true)
 |-- hp: integer (nullable = true)
 |-- drat: double (nullable = true)
 |-- wt: double (nullable = true)
 |-- qsec: double (nullable = true)
 |-- vs: integer (nullable = true)
 |-- am: integer (nullable = true)
 |-- gear: integer (nullable = true)
 |-- carb: integer (nullable = true)
```

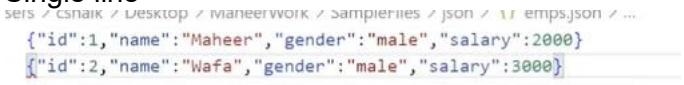
```
+-----------------+----+---+-----+---+----+-----+-----+---+---+----+----+
|            model| mpg|cyl| disp| hp|drat|   wt| qsec| vs| am|gear|carb|
+-----------------+----+---+-----+---+----+-----+-----+---+---+----+----+
|        Mazda RX4|21.0|  6|160.0|110| 3.9| 2.62|16.46|  0|  1|   4|   4|
|    Mazda RX4 Wag|21.0|  6|160.0|110| 3.9|2.875|17.02|  0|  1|   4|   4|
|       Datsun 710|22.8|  4|108.0| 93|3.85| 2.32|18.61|  1|  1|   4|   1|
|   Hornet 4 Drive|21.4|  6|258.0|110|3.08|3.215|19.44|  1|  0|   3|   1|
|Hornet Sportabout|18.7|  8|360.0|175|3.15| 3.44|17.02|  0|  0|   3|   2|
+-----------------+----+---+-----+---+----+-----+-----+---+---+----+----+
```

Reading JSON file

Using read.json("path") or read.format("json").load("path") you can read a JSON file into a pyspark dataframe.
Use **multiline** option, to read JSON files scattered across multiple lines, By default multiline option, is set to false. Gives error if file is like
Multiline

```
{
    "name": "John",
    "age": 25,
    "city": "Sampleville",
    "married": false,
    "hobbies": ["reading", "traveling", "programming"],
    "address": {
      "street": "123 Main Street",
      "city": "Sample City",
      "postal_code": "12345"
    }
}
```

Single line



```
{"id":1,"name":"Maheer","gender":"male","salary":2000}
{"id":2,"name":"Wafa","gender":"male","salary":3000}
```

```
df_json = (spark.read.format("json")
           .load("C:\\Users\\GowdhamanBJ\\Downloads\\sample1.json"))
print('json file')
df_json.show()
```

```
json file
Traceback (most recent call last):
  File "C:\Users\GowdhamanBJ\Desktop\pyspark\read_diff_file.py", line 54, in <module>
```

Need use

```
df_json = (spark.read.format("json")
           .option(key="multiline", value="True")
           .load("C:\\Users\\GowdhamanBJ\\Downloads\\sample1.json"))
print('json file')
df_json.show()
```

```python
schema_json = StructType([
    StructField( name: "name", StringType(),  nullable: False),
    StructField( name: "age", IntegerType(),  nullable: False),
    StructField( name: "city", StringType(), nullable: False),
    StructField( name: "married",BooleanType(),  nullable: False),
    StructField( name: "hobbies",ArrayType(StringType()), nullable: True),
    StructField( name: "address", StructType([
        StructField( name: "street", StringType(),  nullable: True),
        StructField( name: "city", StringType(),  nullable: True),
        StructField( name: "postal_code", StringType(),  nullable: True),
    ]),  nullable: True)
])

df_schema_json = (spark.read.schema(schema_json)
                  .option(key="multiline", value="True")
                  .json("C:\\Users\\GowdhamanBJ\\Downloads\\sample1.json"))
print("schema")
df_schema_json.printSchema()
```

```
schema
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- city: string (nullable = true)
 |-- married: boolean (nullable = true)
 |-- hobbies: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- address: struct (nullable = true)
 |    |-- street: string (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- postal_code: string (nullable = true)

+----+---+----------+-------+--------------------------------+--------------------------------------+
|name|age|city      |married|hobbies                         |address                               |
+----+---+----------+-------+--------------------------------+--------------------------------------+
```

Write method

The ".write" method is used to **save a DataFrame** to **a file**, **table**, or **external system**, in formats like **CSV**, **JSON**, **Parquet**, **ORC**, **Delta**, **JDBC**, etc.

Syntax
df.write.format("format").save("path")
formats are csv, json, parquet

1.**header**: Specifies whether to include a header row with column names in the CSV file. Example: option("header", "true").
2.**delimiter**: Specifies the delimiter to use between fields in the CSV file. Example: option("delimiter", ",").

3.**quote**: Specifies the character used for quoting fields in the CSV file. Example: option("quote", """).
4.**escape**: Specifies the escape character used in the CSV file. Example: option("escape", "\").
5.**nullValue**: Specifies the string to represent null values in the CSV file. Example: option("nullValue", "NA").
6.**dateFormat**: Specifies the date format to use for date columns. Example: option("dateFormat", "yyyy-MM-dd").
7.**mode**: Specifies the write mode for the output. Options include “overwrite”, “append”, “ignore”, and “error”. Example: option("mode", "overwrite").

Option	Default	Purpose
header	`false`	Writes column names as first row (`True` = include header).
delimiter	`,`	Field separator (can be `'"
quote	`"`	Character for quoting values.
quoteAll	`false`	Whether to quote **all** fields.
escape	`\`	Escape character for quotes in values.
nullValue	empty string	String to represent `null` values.
emptyValue	empty string	String to represent empty strings (`""`).
dateFormat	`yyyy-MM-dd`	Format for date columns.
timestampFormat	`yyyy-MM-dd'T'HH:mm:ss.SSSXXX`	Format for timestamp columns.
ignoreLeadingWhiteSpace	`true`	Trim leading spaces in values.
ignoreTrailingWhiteSpace	`true`	Trim trailing spaces in values.
multiLine	`false`	Allow fields with multi-line values.
compression	none	Compression type (`gzip`, `bzip2`, etc.).
encoding	`UTF-8`	File encoding.

File Descriptions:
1. part-00000-tid-xxxx...
   - This is the actual data file containing your DataFrame content.
   - Format depends on what you used (e.g., .csv, .parquet, or .json) — extensions may not show by default.
   - Since Spark is distributed, it writes part files per partition.
   - In your case, .coalesce(1) likely led to only one file (part-00000).
2. _SUCCESS
   - A small marker file written by Spark.
   - Its presence means the write operation completed successfully.
   - No data in it; often used by downstream systems (like Hadoop or Airflow) to verify success.
3. _started_4012292023...
   - A job tracking file created at the beginning of a write transaction.
   - Helps Spark keep track of when the job started.

- Usually seen when using commit protocols (like when saving Delta or writing in atomic fashion).

4. _committed_4012292...
   - Indicates that the job committed successfully.
   - Timestamp or ID in the filename matches the *started** file.
   - Used to finalize the transaction and verify atomicity (especially in distributed write protocols).

Modes of writing

overwrite	Replace existing data (delete available record fully and write new thing in file)
append	Adds data to existing data
ignore	Ignores write if data already exists
Error or errorifexists	Throws error if data already exists

csv	parquet	json
df.write \ .mode("overwrite") \ .option("header", True) \ .option("delimiter", ",") \ .csv("output/csv_path")	df.write \ .mode("overwrite") \ .parquet("output/parquet_path")	df.write \ .mode("overwrite") \ .json("output/json_path")

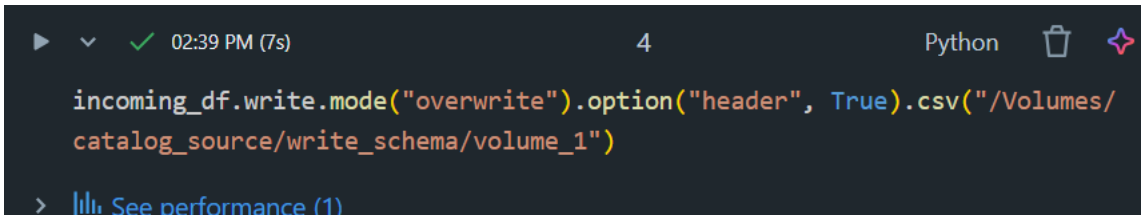
Writing with partitioning

df.write.partitionBy("column1", "column2").mode("overwrite").csv("path")

Advance: conditional partition overwrite (replace where)

```
df.write \
  .option("replaceWhere", "city = 'New York'") \
  .mode("overwrite") \
  .partitionBy("city") \
  .parquet("output_path")
```

- Only overwrites rows in the specified partition (city = 'New York')
- Used for **efficient updates**



Append

- Adds new records to the existing data.
- Does not delete or update any previous data.

```
2 minutes ago (8s)                5                Python
data2 = [
    (5, "East", "exam@example.com"),
    (6, "West", "west@example.com")
]
new_df = spark.createDataFrame(data2, schema1)

new_df.write.mode("append").option("header", True).csv("/Volumes/catalog_source/
write_schema/volume_1")
```

Use Case:

- Ingesting new logs or transaction records.

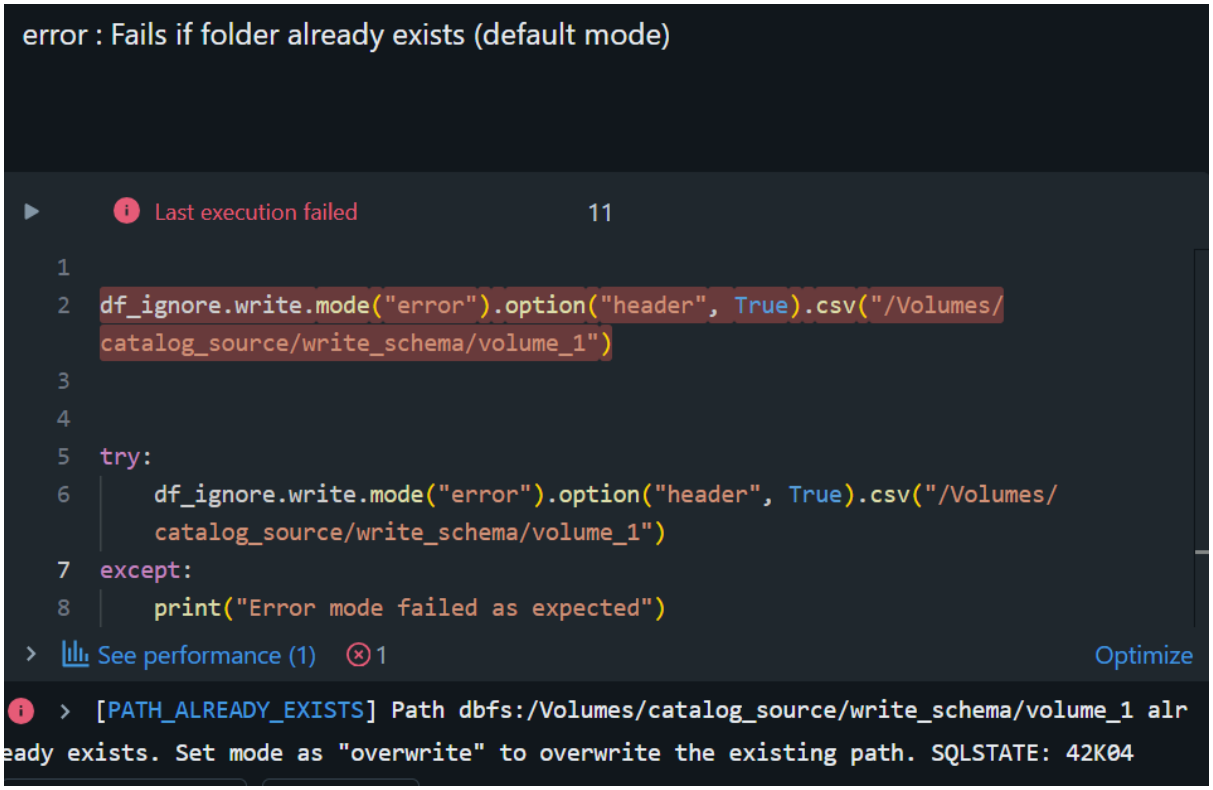
```
# overwrite
df.write \
    .mode("overwrite") \
    .option( key: "header",  value: True) \
    .csv("C:/Users/GowdhamanBJ/Desktop/pyspark_output/csv_output")

# append (requires same schema; run overwrite first)
df.write \
    .mode("append") \
    .option( key: "header",  value: True) \
    .csv("C:/Users/GowdhamanBJ/Desktop/pyspark_output/csv_output")
```

```
# ignore (skips if folder exists)
df.write \
    .mode("ignore") \
    .option( key: "header",  value: True).csv("C:/Users/GowdhamanBJ/Desktop/pyspark_output/csv_output")

# error (throws error if folder exists)
df.write.mode("errorifexists").option( key: "header",  value: True)  \
    .csv("C:/Users/GowdhamanBJ/Desktop/pyspark_output/csv_output")
```

3.ignore : Skips writing if folder already exists

```
2 minutes ago (2s)                8

data = [
    (1, "A", "a@example.com"),
    (2, "B", "b@example.com"),
    (3, "C", "c@example.com")]
columns = ["id","name", "email"]
df_ignore = spark.createDataFrame(data, columns)
display(df_ignore)


df_ignore.write.mode("ignore").option("header", True).csv("/Volumes/catalog_source/
write_schema/volume_1")
```

error : Fails if folder already exists (default mode)



```
Last execution failed             11

1
2  df_ignore.write.mode("error").option("header", True).csv("/Volumes/
   catalog_source/write_schema/volume_1")
3
4
5  try:
6      df_ignore.write.mode("error").option("header", True).csv("/Volumes/
       catalog_source/write_schema/volume_1")
7  except:
8      print("Error mode failed as expected")

> See performance (1)  1                                   Optimize

> [PATH_ALREADY_EXISTS] Path dbfs:/Volumes/catalog_source/write_schema/volume_1 alr
eady exists. Set mode as "overwrite" to overwrite the existing path. SQLSTATE: 42K04
```

Upsert (merge)
Description:

- Performs an insert or update based on a matching condition.
- Requires Delta Lake (or similar support for ACID).
- **Update if match, insert if not.**

Use Case:

- Implementing **Slowly Changing Dimensions (SCD Type 1 or 2)**.

- Data lake table synchronization and deduplication.

SCD – slowly changing dimension

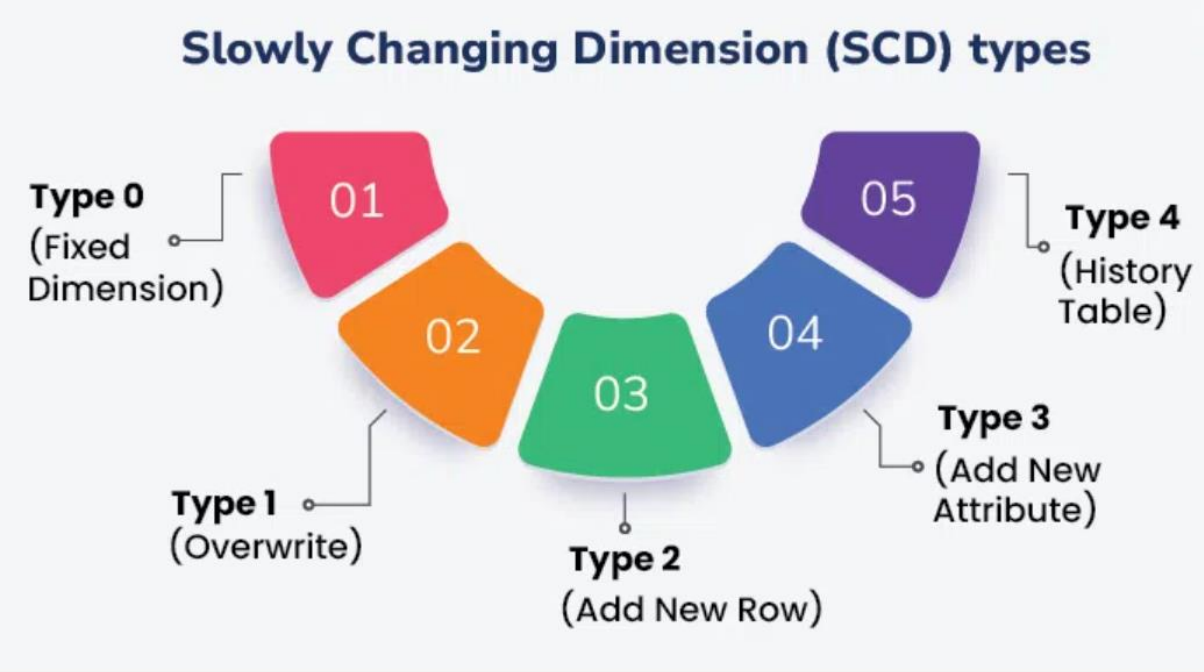
Slowly Changing Dimensions (SCD) are a critical concept in data warehousing and business intelligence.
They refer to the methods used to manage and track changes in dimension data over time. This is essential for maintaining historical accuracy and ensuring data integrity in a data warehouse.

In a typical data warehouse, dimension data such as customer information, product details, or employee records can change. Managing these changes efficiently and effectively is where Slowly Changing Dimensions (SCD) come into play. There are several types of SCDs, each providing a different way to handle changes, ensuring that historical data is preserved and accurately reflected.

(A SCD is a dimension that stores and manages both current and historical data over time in a data warehouse).

Features of Slowly Changing Dimensions:
- Historical Tracking
- Data Integrity
- Type-Based Classification
- Versioning
- Temporal Analysis

SCD Types	Brief
Type 0	Passive method
Type 1	Overwrite old value
Type 2	Add new row
Type 3	Add new column
Type 4	Use Historical table
Type 6	Combine SCD type 1+2+3




```
data = [
    (1, "Alice", "alice@example.com", "2024-01-01", None, True, None),
    (2, "Bob", "bob@example.com", "2024-01-01", None, True, None),
    (3, "Charlie", "charlie@example.com", "2024-01-01", None, True, None)
]
schema = ["customer_id", "name", "email", "start_date", "end_date", "is_current", "previous_name"]
# Existing dimension table
dim_df = spark.createDataFrame(data, schema=dim_schema)
```

```
# Incoming updated data
data1 = [
    (1, "Alice A", "alice@example.com"),
    (2, "Bob", "bob.new@example.com"),
    (4, "Diana", "diana@example.com")
]
schema1 = ["customer_id", "name", "email"]
incoming_df = spark.createDataFrame(data1, schema1)
```

Type 0 (Fixed Dimension)

- Description: In the case of Type 0 dimensions, there are no changes whatsoever. The primary uses for static data are when the data does not change over time, for example, states, zip codes, county codes, and date of birth etc.
- Implementation: The records located in these tables are unalterable and no modification can be made to the record.

```
#Type 0 - no change
scd0_df = dim_df
scd0_df.show()
```

```
+-----------+-------+-------------------+----------+--------+----------+-------------+
|customer_id|   name|              email|start_date|end_date|is_current|previous_name|
+-----------+-------+-------------------+----------+--------+----------+-------------+
|          1|  Alice|  alice@example.com|2024-01-01|    NULL|      true|         NULL|
|          2|    Bob|    bob@example.com|2024-01-01|    NULL|      true|         NULL|
|          3|Charlie|charlie@example.com|2024-01-01|    NULL|      true|         NULL|
+-----------+-------+-------------------+----------+--------+----------+-------------+
```

Type 1 (Overwrite) – No History

- Description: For changes in Type 1 dimensions; overwriting is used where the new value simply replaces the old value that was already stored. Unfortunately, this method does not save any previous data (history); that is, it cannot illustrate changes over time.
- Implementation: When an update happens, then the new value replaces the previous value in the database without any interference. It is applied in situations where the historical data is irrelevant to the task, for example, changing a customer's current address.

```
#type 1 - No history
scd1_df = dim_df.alias("dim").join(incoming_df.alias("inc"), on: "customer_id", how: "outer") \
    .select(
        col("customer_id"),
        when(col("inc.name").isNotNull(), col("inc.name")).otherwise(col("dim.name")).alias("name"),
        when(col("inc.email").isNotNull(), col("inc.email")).otherwise(col("dim.email")).alias("email"),
        col("dim.start_date"),
        col("dim.end_date"),
        col("dim.is_current"),
        col("dim.previous_name")
    )
scd1_df.show()
```

```
+-----------+-------+-------------------+----------+--------+----------+-------------+
|customer_id|   name|              email|start_date|end_date|is_current|previous_name|
+-----------+-------+-------------------+----------+--------+----------+-------------+
|          1|Alice A|  alice@example.com|2024-01-01|    NULL|      true|         NULL|
|          2|    Bob|bob.new@example.com|2024-01-01|    NULL|      true|         NULL|
|          3|Charlie|charlie@example.com|2024-01-01|    NULL|      true|         NULL|
|          4|  Diana|  diana@example.com|      NULL|    NULL|      NULL|         NULL|
+-----------+-------+-------------------+----------+--------+----------+-------------+
```

SCD Type 0 → Fixed (No Change)

- Once inserted, data never changes.
- Use when history must **never be updated** (e.g., Date of Birth).

Example: If name changes → no update allowed.

CustomerID	Name	DOB
101	Raja	1990-05-10

SCD Type 1 → Overwrite (No History)

- Old value replaced with new value.
- No history kept.

CustomerID	Name	City
101	Raja	Chennai

If city changes to "Bangalore" →

| 101 | Raja | Bangalore |

Example: If city changes to "Bangalore" from “chennai”

SCD Type 2 → Add New Record (Full History)

- New row inserted for every change.
- Historical tracking with **valid_from, valid_to, current_flag**.

Name	City	valid_from	valid_to	current_flag
Raja	Chennai	2018-01-01	2020-05-01	0
Raja	Bangalore	2020-05-02	9999-12-31	1

Bangalore → delhi

Name	City	valid_from	valid_to	current_flag
Raja	Chennai	2018-01-01	2020-05-01	0
Raja	Bangalore	2020-05-02	2023-08-17	0
Raja	Delhi	2023-08-18	9999-12-31	1

Best when full **history tracking** is needed.

SCD Type 3 → Limited History (New Column)

- Add new column to keep only **previous value**.
- Tracks limited history.

CustomerID	Name	Current_City	Previous_City
101	Raja	Bangalore	Chennai

SCD Type 4 → History Table

- Current table keeps only latest values.
- Separate **history table** stores old records.

📍 Current Table

CustomerID	Name	City
101	Raja	Bangalore

📍 History Table

CustomerID	Name	City	valid_from	valid_to
101	Raja	Chennai	2018-01-01	2020-05-01

Useful when history is required but main table must remain small.

SCD Type 6 → Hybrid (1 + 2 + 3)

- Combination of **Type 1, Type 2, Type 3**.
- Maintains full history (like Type 2) + overwrites some attributes (Type 1) + keeps previous values in separate columns (Type 3).

TABLES:

Managed Table (Internal table)

A Managed Table is one where both the table schema (definition) and the data are managed by the database system (e.g., Spark, Hive).

Spark manages both the data and metadata.
Characteristics:

- The database controls both schema and data.
- Data is stored in the default warehouse location.
- If the table is dropped, both the schema and the underlying data are deleted.

Use Case:

- Ideal when the system is responsible for managing the entire data lifecycle.
- Suitable when you don't need to retain the raw data after the table is deleted.

External Table

An External Table is one where the schema is managed by the database system, but the data resides outside the system (e.g., in HDFS, S3, or DBFS).

Characteristics:

- Data is stored outside the database system (in external storage).
- Dropping the table will remove only the metadata/schema, while data remains intact.
- The location of the data must be explicitly provided.

Use Case:

- Best suited when you want to retain original data even if the table is dropped.
- Useful when data is shared across multiple tools or systems or managed externally.

Pros and Cons

Managed Tables

1. Storage is managed by Databricks
2. Data drops on deletion
3. Portability Limited
4. Ideal for Temporary/Dev Workflows

External Tables

1. Storage is managed by You
2. Data doesn't drop on deletion
3. High Portability
4. Ideal for Production/Shared Data Lakes

Key Differences Between Managed and External Tables

Feature	Managed Table	External Table
Data Location	Data is stored **within the database** system itself.	Data is stored **outside** the database (e.g., cloud storage, HDFS).
Dropping Table	Deleting the table removes both the schema and the data.	Deleting the table only removes the schema, not the data.
Data Management	Database manages both data and schema.	Database only manages schema; data is managed externally.
Use Case	Suitable for transient data where both schema and data are managed together.	Useful when data is stored outside the system but needs to be queried or accessed.
Example Storage Locations	Internal DB storage, managed file system.	Cloud (S3), HDFS, Local File System.

Example of Use Cases in Real Life:

1. **Managed Tables Use Case:**
   - **Data Warehouse:** If you're loading data into a table where the database system is responsible for the lifecycle of both the schema and the data (e.g., sales data for a month), a managed table works well. After the month's data is processed and no longer needed, you can drop the table, and the data is deleted automatically.
2. **External Tables Use Case:**
   - **Data Lake or ETL Pipeline:** Imagine you have a massive amount of historical data stored in Amazon S3 or HDFS, and you want to analyze it without copying it into your database. In this case, you would create an **external table** in your system, pointing to the location of the data in S3 or HDFS, allowing you to query and work with it as though it were in your database, without modifying or deleting the original data.

Conclusion:

- **Managed Tables** are ideal when the database manages both schema and data, and when you want the data to be tightly coupled with the table, ensuring that deleting the table also deletes the data.
- **External Tables** are useful when the data is stored outside of the database and you want to keep it intact even after dropping the table schema. This is ideal for accessing large amounts of data that reside in external systems or cloud storage without moving or copying the data into the database.

Types of Loads

Full load:

Entire data in target system is replaced from source system data in every load.

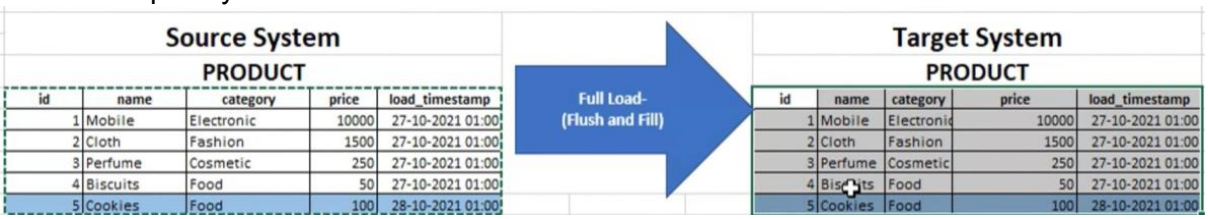
A **full load** involves loading **all records** from the source system into the target system every time the load process runs.

Characteristics:

- All data is extracted, transformed (if needed), and reloaded.
- Existing data in the target is typically **truncated** or **overwritten**.
- Simple to implement but **resource intensive**.
- Commonly used in small datasets or during **initial data loads**.

Use Case:

- First-time load into a data warehouse.
- When data volume is small or data changes frequently.
- Frequency of load is less.



Incremental Load:

Incremental data load refers to the process of adding new or modified data to an existing dataset without reloading the entire dataset.

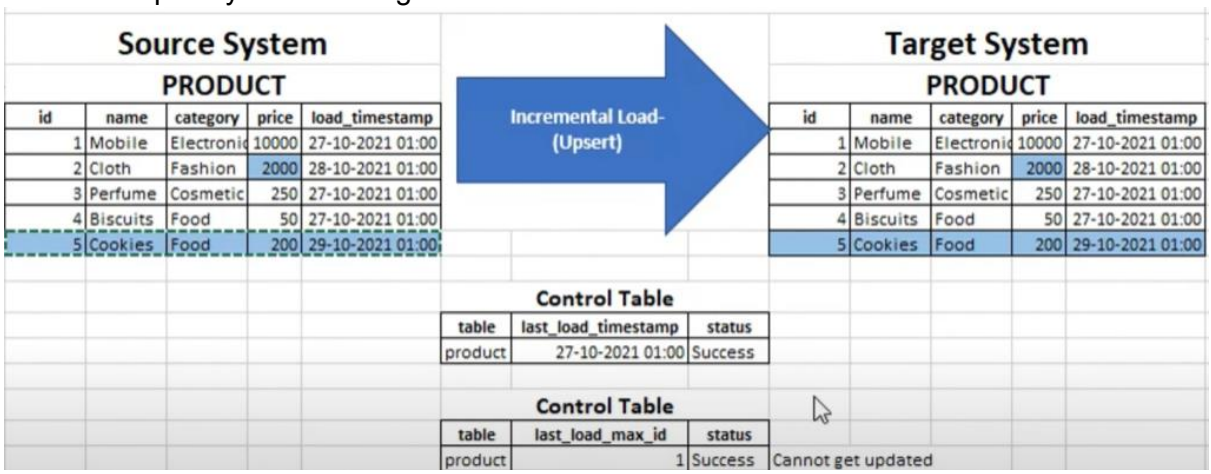
An **incremental load** brings only the **new or changed records** (based on timestamps, IDs, or other indicators) from the source into the target.

Characteristics:

- More efficient than full load.
- Relies on a **change tracking mechanism** (e.g., last modified date, version number).
- Requires logic to **detect and merge** new/updated data.

Use Case:

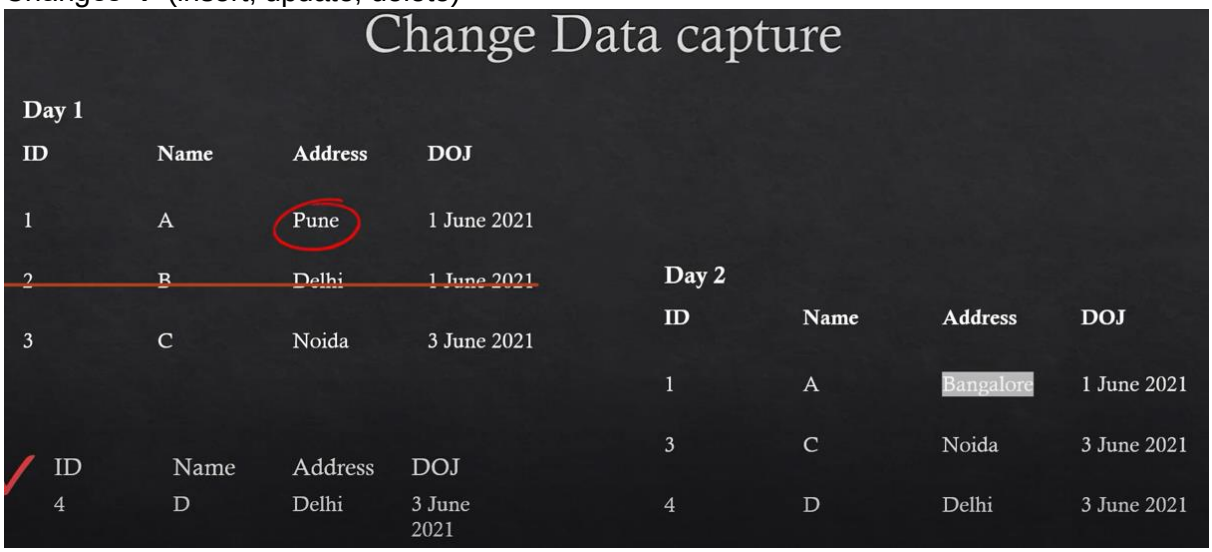
- Periodic updates in production environments.
- Datasets with large volume but relatively fewer changes.
- Frequency of load is high.



<u>Change Data Capture</u>

CDC is a technique that identifies and captures **only the data that has changed** in the source system (inserts, updates, deletes).

Changes → (insert, update, delete)



Characteristics:

- Uses database logs, triggers, timestamps, or tools like Debezium, GoldenGate, or SQL Server CDC.

- Ensures **real-time or near real-time** data replication.
- Efficient and accurate.

Use Case:

- Streaming pipelines.
- Real-time analytics and reporting.
- Replication

Snapshot load

A **snapshot** is a **point-in-time copy** of a dataset that captures the exact state of the data **at a specific moment**. (Think of it like taking a daily photo of your entire customer table — you can compare these photos to track changes over time).

Why Use Snapshot Loads?

- To **track historical changes** in data (e.g., customer status, product prices, employee assignments).
- To **audit changes** for regulatory or business reporting.
- To enable **trend or time-series analysis**.
- For **SCD Type 2** style tracking (historical dimension values).

Types of snapshot loads

1. Full snapshot load

Takes a complete copy of the source data at regular intervals (daily, weekly, monthly, etc.) and stores it as a new version or partition in the target system.

Characteristics:

- Simple and reliable
- Stores redundant data (lots of duplication)
- Requires more storage
- Great for complete audit trails or point-in-time recovery.

2. Incremental snapshot load

Captures only the new or changed records since the last snapshot but appends them with a timestamp/version so history is preserved.

Characteristics:

- SCD type 2 methodology.
- Tracking changes without duplicating the entire dataset.
- Storage-efficient than full snapshot.

3. Rolling snapshot load

Maintains only the last N-snapshots (e.g., last 7 days or 12 months) and deletes or archives older ones.

Characteristics:

- Prevents unbounded storage growth.
- Good for short-term trend analysis.
- You decide the “window” (rolling period) you care about.

Snapshot Type	Captures	Storage Needs	Historical Tracking	Use Case Example
Full Snapshot	Entire dataset	High	Yes	Daily backup, audit logs
Incremental Snapshot	Only new/changed rows	Moderate	Yes	SCD Type 2, historical dimensions

Rolling Snapshot	Recent N snapshots	Controlled	Partial	Rolling dashboards, short-term trends

Load Type	Definition	Captures	Storage Needs	Use Case
Full Load	Loads entire dataset every time, usually by truncating and reloading. (overwrite)	All data	High	Initial loads, simple small datasets, non-frequent changes
Incremental Load	Loads only new or changed data since last load.	New/updated data only	Low to Moderate	Ongoing updates, efficient large-scale loading
Full Snapshot Load	Captures entire dataset at a point in time and stores it separately.	Full state per load	High	Audit, point-in-time tracking, time-series analysis
Incremental Snapshot	Captures only new/changed records with versioning/history tracking.	Changed records with timestamps	Moderate	SCD Type 2, historical dimension tracking
Rolling Snapshot	Keeps only last N snapshots, removes or archives older ones.	Sliding window of snapshots	Controlled	Recent trend dashboards, short-term insights

Library

cookiecutter

A command-line utility that creates projects from project templates, e.g. creating a Python package project from a Python package project template.

pip install cookiecutter

Using Cookiecutter offers major productivity and consistency benefits, especially when working on multiple projects, in a team, or in a larger organization.

i. Use an Existing Template
cookiecutter <https://github.com/ >

ii. Create Your Own Template (Optional)

```
cookiecutter-mytemplate/
 ├── {{ cookiecutter.project_slug }}/
│    ├── __init__.py
│    └── main.py
 ├── README.md
 └── cookiecutter.json
```

```
# {{ cookiecutter.project_name }}
def main():
    print("Hello from {{ cookiecutter.project_slug }}!")
```

Use Cases:

- Creating new Python packages
- Standardizing project structure across teams or organizations

Pydantic

Pydantic is a Python library that provides **data validation** and **data parsing** using **Python type annotations**.

Use cases:
Data validation (automatically checks if input data matches expected type/formats).
Type safety (parses JSON or raw input typed python objects).
Error reporting (clearly tells you what's wrong with bad data input).
Built-in defaults
Integration-friendly

```
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name: str
    is_active: bool = True

# Valid input
user = User(id='101', name='Alice')  # auto-converts id to int
print(user.dict())  # {'id': 101, 'name': 'Alice', 'is_active': True}

# Invalid input
User(id='not_a_number', name='Bob')  # raises ValidationError
```

Tox

Tox is a test automation tool that helps you:

- Test your code in multiple python environments.
- Automate running test, linter, formatters, etc.
- Simulate a CI/CD environment locally

Feature
Multi-version testing – runs tests across different python versions (e.g., 3.8, 3.9, 3.10)
Dependency isolation – creates separate virtual environments per test environment.

Command automation – runs any shell command (test runners, formatters, linters, etc)

Tox Use Cases:

- Ensuring your code works in multiple Python versions
- Automating test + lint + coverage in one command
- Pre-commit hooks and CI pipelines
- Your library must work on Python 3.8 and 3.10. Tox automatically tests in both environments during CI/CD.

tox.ini example:

```
[tox]

envlist = py38, py39, lint



[testenv]

deps = pytest

commands = pytest tests/



[testenv:lint]

deps = black

commands = black --check.
```

Black

Black is an opinionated code formatter for Python.
It automatically formats your code to follow PEP8 + consistent styling — without debating over style.

Features:
Auto-formatting (reformats code with consistent spacing, quotes, etc)
Deterministic (output is always the same for the same code).
Fast & simple (works with one command, no config needed).
Editor-friendly

```
# Before
def foo(x,y):return(x+  y)

# After
def foo(x, y):
    return x + y
```

Broadcast Join

A **broadcast join** in PySpark is an **optimization technique** used to speed up join operations when one dataset is **much smaller** than the other.
By **broadcasting** (i.e., copying) the smaller dataset to **all worker nodes**, Spark avoids expensive **shuffling** of the larger dataset across the cluster.
This makes the **join faster and more efficient**, especially in distributed environments.

```
large_df = spark.createDataFrame([(1, "Aa"), (2, "Bb"), (3, "Cc")], ["id", "name"])
small_df = spark.createDataFrame([(1, "HR"), (2, "Finance")], ["id", "department"])
# Start timer
start_time = time.time()
result_df = large_df.join(broadcast(small_df), on="id", how="inner")
result_df.show()
# End timer
end_time = time.time()
print(f"Join execution time: {end_time - start_time:.4f} seconds")
```

```
+---+----+----------+
| id|name|department|
+---+----+----------+
|  1|  Aa|        HR|
|  2|  Bb|   Finance|
+---+----+----------+

Join execution time: 17.6974 seconds
```

Without broadcast join

```
large_df = spark.createDataFrame([(1, "Aa"), (2, "Bb"), (3, "Cc")], ["id", "name"])
small_df = spark.createDataFrame([(1, "HR"), (2, "Finance")], ["id", "department"])
# Start timer
start_time = time.time()
result_df = large_df.join(small_df, on="id", how="inner")
result_df.show()
# End timer
end_time = time.time()
print(f"Join execution time: {end_time - start_time:.4f} seconds")
```

```
+---+----+----------+
| id|name|department|
+---+----+----------+
|  1|  Aa|        HR|
|  2|  Bb|   Finance|
+---+----+----------+

Join execution time: 18.6662 seconds
```

Null handling

1. Drop rows with Nulls

Use **.dropna()** to remove rows that have NULLs in one or more columns.

```
print("dropna -- drops the row if their is a single Null element")
df_dropna = df.dropna()
df_dropna.show()
```

```
dropna -- drops the row if their is a single Null element
+---+-----+---+-----+-----------+
| id| name|age|state|signup_date|
+---+-----+---+-----+-----------+
|  4|David| 45|   CA| 2025-08-04|
+---+-----+---+-----+-----------+
```

```
print("drop rows where all columns are null")
df_dropall = df.dropna(how="all")
df_dropall.show()
```

```
drop rows where all columns are null
+----+-------+----+-----+-----------+
|  id|   name| age|state|signup_date|
+----+-------+----+-----+-----------+
|   1|  Alice|  30| NULL| 2025-08-01|
|   2|   NULL|NULL|   NY| 2025-08-02|
|   3|Charlie|NULL| NULL|       NULL|
|   4|  David|  45|   CA| 2025-08-04|
|   5|   NULL|  29| NULL| 2025-08-05|
|NULL|    Eva|  35|   TX|       NULL|
|   7|  Frank|NULL|   FL| 2025-08-07|
|   8|   NULL|NULL| NULL|       NULL|
+----+-------+----+-----+-----------+
```

```
print("drop if name or id is null")
df_drop = df.dropna(subset=["name","id"])
df_drop.show()
```

```
drop if name or id is null
+---+-------+----+-----+-----------+
| id|   name| age|state|signup_date|
+---+-------+----+-----+-----------+
|  1|  Alice|  30| NULL| 2025-08-01|
|  3|Charlie|NULL| NULL|       NULL|
|  4|  David|  45|   CA| 2025-08-04|
|  7|  Frank|NULL|   FL| 2025-08-07|
+---+-------+----+-----+-----------+
```

Filter null columns

```
print("filter null columns")
df.filter(F.col("name").isNull()).show()
df.filter(F.col("name").isNotNull()).show()
```

```
+----+----+----+-----+-----------+
|  id|name| age|state|signup_date|
+----+----+----+-----+-----------+
|   2|NULL|NULL|   NY| 2025-08-02|
|   5|NULL|  29| NULL| 2025-08-05|
|   8|NULL|NULL| NULL|       NULL|
|NULL|NULL|NULL| NULL|       NULL|
+----+----+----+-----+-----------+
```

```
+----+-------+----+-----+-----------+
|  id|   name| age|state|signup_date|
+----+-------+----+-----+-----------+
|   1|  Alice|  30| NULL| 2025-08-01|
|   3|Charlie|NULL| NULL|       NULL|
|   4|  David|  45|   CA| 2025-08-04|
|NULL|    Eva|  35|   TX|       NULL|
|   7|  Frank|NULL|   FL| 2025-08-07|
+----+-------+----+-----+-----------+
```

Fillna -- fill null values

```python
mean_age = df.agg(F.mean("age")).collect()[0][0]
print(type(mean_age), mean_age)
df.fillna({"age":mean_age}).show()
```

```
+----+-------+---+-----+-----------+
|  id|   name|age|state|signup_date|
+----+-------+---+-----+-----------+
|   1|  Alice| 30| NULL| 2025-08-01|
|   2|   NULL| 34|   NY| 2025-08-02|
|   3|Charlie| 34| NULL|       NULL|
|   4|  David| 45|   CA| 2025-08-04|
|   5|   NULL| 29| NULL| 2025-08-05|
|NULL|    Eva| 35|   TX|       NULL|
|   7|  Frank| 34|   FL| 2025-08-07|
|   8|   NULL| 34| NULL|       NULL|
|NULL|   NULL| 34| NULL|       NULL|
+----+-------+---+-----+-----------+
```

```python
df.fillna("Unknown").show()  # fill all string columns
df.fillna(0).show()  # fill numeric columns
df.fillna({"name":"unknown","age":0}).show() #fill default value
```

```
+----+-------+----+-------+-----------+
|  id|   name| age|  state|signup_date|
+----+-------+----+-------+-----------+
|   1|  Alice|  30|Unknown| 2025-08-01|
|   2|Unknown|NULL|     NY| 2025-08-02|
|   3|Charlie|NULL|Unknown|    Unknown|
|   4|  David|  45|     CA| 2025-08-04|
|   5|Unknown|  29|Unknown| 2025-08-05|
|NULL|    Eva|  35|     TX|    Unknown|
|   7|  Frank|NULL|     FL| 2025-08-07|
|   8|Unknown|NULL|Unknown|    Unknown|
|NULL|Unknown|NULL|Unknown|    Unknown|
+----+-------+----+-------+-----------+
```

```
+---+-------+---+-----+-----------+
| id|   name|age|state|signup_date|
+---+-------+---+-----+-----------+
|  1|  Alice| 30| NULL| 2025-08-01|
|  2|   NULL|  0|   NY| 2025-08-02|
|  3|Charlie|  0| NULL|       NULL|
|  4|  David| 45|   CA| 2025-08-04|
|  5|   NULL| 29| NULL| 2025-08-05|
|  0|    Eva| 35|   TX|       NULL|
|  7|  Frank|  0|   FL| 2025-08-07|
|  8|   NULL|  0| NULL|       NULL|
|  0|   NULL|  0| NULL|       NULL|
+---+-------+---+-----+-----------+
```

```
|  id|   name|age|state|signup_date|
+----+-------+---+-----+-----------+
|   1|  Alice| 30| NULL| 2025-08-01|
|   2|unknown|  0|   NY| 2025-08-02|
|   3|Charlie|  0| NULL|       NULL|
|   4|  David| 45|   CA| 2025-08-04|
|   5|unknown| 29| NULL| 2025-08-05|
|NULL|    Eva| 35|   TX|       NULL|
|   7|  Frank|  0|   FL| 2025-08-07|
|   8|unknown|  0| NULL|       NULL|
|NULL|unknown|  0| NULL|       NULL|
+----+-------+---+-----+-----------+
```

How to Handle NULLs in Spark

1. Remove NULLs
   - df.na.drop() → drop if **any column** is null.
   - df.na.drop(how="all") → drop if **all columns** are null.
   - df.na.drop(subset=["col1","col2"]) → drop only if those specific columns have

2. fill nulls
   - df.na.fill(0) → replace all nulls with 0 (works on numeric columns).
   - df.na.fill("unknown") → replace all nulls with "unknown" (works on string columns).
   - df.na.fill({"age": 0, "name": "unknown"}) → replace nulls **per column with different values**.

3. Imputation
   - Replace NULLs with **mean/median/mode**

```
from pyspark.ml.feature import Imputer
imputer = Imputer(inputCols=["age"], outputCols=["age_filled"]).setStrategy("mean")
df_imputed = imputer.fit(df).transform(df)
```

4. Conditional Handling

- Using when + otherwise:

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

df = df.withColumn("city", when(col("city").isNull(), "Unknown").otherwise(col("city")))
```

Benefits of Removing / Handling NULLs

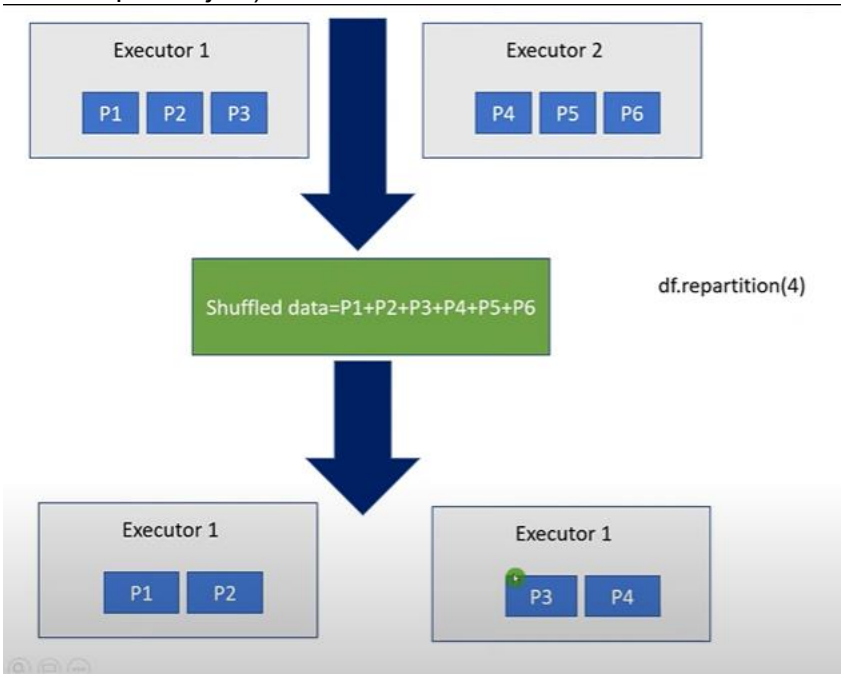
Data Quality – NULLs can cause wrong aggregations, averages, and joins.
Avoid Runtime Errors – Some Spark functions (like casting or mathematical operations) fail on NULLs.
Performance – Dropping unnecessary NULL rows reduces data size → faster processing.
Reliable Analytics – Ensures reports/BI dashboards don't show misleading "NULL" values.

Partition

Partition is a key for parallel processing, key for performance and is a key for distributed framework.
Important thing is choosing the right no.of partition and size of the partition. (check spark document).

Repartition

- Function repartition is used to increase or decrease partition in spark.
- Repartition always shuffles (shuffling is one of the costly operations in spark) the data and builds new partition from scratch.
- Repartition results in almost equal sized partitions.
- Due to full shuffle, it is not good for performance in some use cases. But as it creates equal sized partitions, good for performance in some use cases.
- **Use case**: When you need a balanced distribution (good for parallelism before wide ops like join)



Repartition

```
Cmd 17

1  from pyspark.sql.types import IntegerType
2  df = spark.createDataFrame(range(10), IntegerType())
3
4  df.rdd.glom().collect()

▸ (1) Spark Jobs
▸ df: pyspark.sql.dataframe.DataFrame = [value: integer]
Out[11]: [[Row(value=0)],
 [Row(value=1)],
 [Row(value=2)],
 [Row(value=3), Row(value=4)],
 [Row(value=5)],
 [Row(value=6)],
 [Row(value=7)],
 [Row(value=8), Row(value=9)]]
Command took 0.60 seconds -- by audaciousazure@gmail.com at 13/07/202

Cmd 18

1  df1=df.repartition(20)
2  df1.rdd.getNumPartitions()
3
```

If there is no enough data then it will create an empty partition

```
1  df1=df.repartition(20)
2  df1.rdd.getNumPartitions()
3

▸ (1) Spark Jobs
▸ df1: pyspark.sql.dataframe.DataFrame = [value: integer]
Out[12]: 20
Command took 0.27 seconds -- by audaciousazure@gmail.com at

Cmd 19

1  df1.rdd.glom().collect()

▸ (1) Spark Jobs
Out[13]: [[],
 [Row(value=0)],
 [],
 [Row(value=5)],
 [],
 [],
 [Row(value=1)],
 [],
 [Row(value=6)],
 [Row(value=2)],
 [],
 [],
```

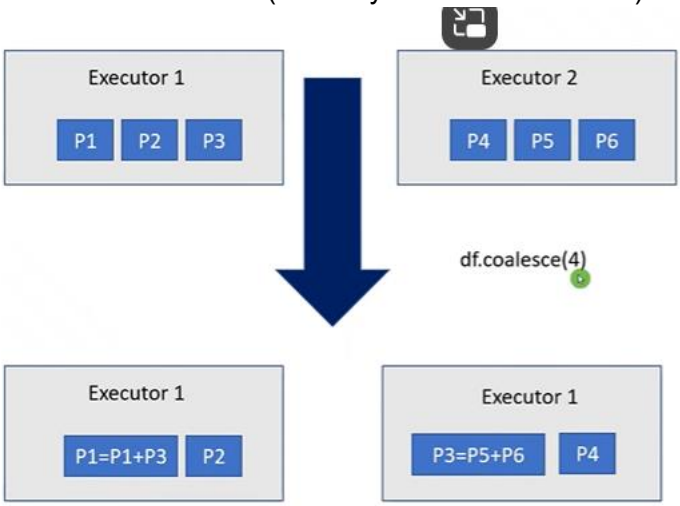
Reduce the partition

```
1  df1=df.repartition(2)
2  df1.rdd.getNumPartitions()

▸ (1) Spark Jobs
▸ df1: pyspark.sql.dataframe.DataFrame = [value: integer]
Out[14]: 2
Command took 0.46 seconds -- by audaciousazure@gmail.com at 13/07/2021

Cmd 21

1  df1.rdd.glom().collect()

▸ (1) Spark Jobs
Out[15]: [[Row(value=0),
  Row(value=1),
  Row(value=2),
  Row(value=3),
  Row(value=5),
  Row(value=6),
  Row(value=7),
  Row(value=8)],
 [Row(value=4), Row(value=9)]]
```

Coalesce

- Coalesce function only reduces the number of partitions.
- coalesce function doesn't require a full shuffle.

- coalesce combines few partitions or shuffles data only from few partitions thus avoiding full shuffles.
- Due to partitions merge, it produces uneven size of partitions
- Since full shuffle is avoided, coalesce is more performant than repartition.
- Avoids shuffle (unless you set shuffle=True)



Coalesce

Use case: When reducing partitions after a wide op (e.g., before writing to disk)

Filter

- Single & multiple condition
- Start with
- Ends with
- Contains
- Like
- Null values
- Isin

```
Filter(df.columns != 50)
Filter(f.col('column1') > 50) & (f.col(column2') > 50))
Filter(df.column.isNull())
Filter(df.column.isNotNull())
Filter(df.column.contains(""))
Filter(df.column.startswith(""))
Filter(df.column.endswith(""))
```

<u>Select Vs WithColumn</u>

Sample Dataframe

```
employee_data = [(111,"Stephen","King",2000),
    (222,"Philipp","Larkin",8000),
    (333,"John","Smith",6000)
  ]
employee_schema = ["Id","FirstName","LastName","salary"]

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

display(df)
```

Cmd 2

Add New Columns with Multiple WithColumn

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

dfWithColumn = df.withColumn("Name",concat(col("FirstName"),lit(" "),col("LastName"))) \
                 .withColumn("BonusPercent",lit(10)) \
                 .withColumn("TotalSalary",col("salary")*col("BonusPercent")) \
                 .withColumn("DateCreated",current_timestamp())

display(dfWithColumn)
```

For adding four new columns spark will create 4 new dataframes internally, it hits the performance.

Avoid this we can go with **Select()**

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

dfSelect = df.select("*"
                     ,concat(col("FirstName"),lit(" "),col("LastName")).alias("Name")
                     ,lit(10).alias("BonusPercent")
                     ,(col("salary")*lit(10)).alias("TotalSalary")
                     ,current_timestamp().alias("DateCreated"))
display(dfSelect)

        # SELECT VS WITHCOLUMN

dfWithColumn = df.withColumn("Name",concat(col("FirstName"),lit(" "),col("LastName"))) \
                 .withColumn("BonusPercent",lit(10)) \
                 .withColumn("TotalSalary",col("salary")*col("BonusPercent")) \
                 .withColumn("DateCreated",current_timestamp())
display(dfWithColumn)
```

withColumns

```
df=df.withColumns({'Age':lit(10),'Year':lit(2023)})
```

If you're on Databricks → withColumns is better when you need to add/replace multiple columns.

If you're on open-source Spark → select is better (because there's no withColumns).

Avoid chaining withColumn in loops — it's the worst for performance.