PySpark end to end features and functionalities

Section-1

Why learning Spark is important?

Many of the world's largest and fastest-growing companies, like **Netflix, Yahoo, and eBay**, use Spark to handle massive amounts of data. These companies process **petabytes** of data on thousands of computers at once.

Why is Spark Important?

- **Super Fast**: Spark can be **100 times faster than Hadoop** because it processes data in memory instead of reading and writing from a hard drive.

Section-3 (Spark Installation and Set Up Standalone)

PyCharm Installation:

Prerequisites:

- Java
- Python
- Spark

Go to https://www.jetbrains.com/pycharm/download/?section=windows to install PyCharm

Pycharm Basics:

- If we want to rename the variables then we can use the Refractor option (Right click => Refractor => Rename) but it does not impact the local variables and vice versa.
- Ctrl+F: to find any variable.
- Alt + Shift +C: to get all the recent changes with time.
- Ctrl + Shift +E: to get all the recent locations.
- Ctrl+ Alt+ T: in how many ways can we use the particular variable.

sys.argv is a list in Python, which contains the command-line arguments passed to the script. The first element, sys.argv[0], is the name of the script itself. Subsequent elements are the arguments passed to the script.

PyCharm Run time Arguments

```
import sys
print("Hello " + sys.argv[0] + ' ' + sys.argv[1] + ' ' +
sys.argv[2])
```

To give the arguments go to the ellipsis and then in the spark parameters add the parameters

OUTPUT:

```
C:\PySpark_Installed\demo\.venv\Scripts\python.exe
C:\PySpark_Installed\demo\test.py spark developers
Hello C:\PySpark_Installed\demo\test.py spark developers

Process finished with exit code 0
```

Integrate Python and PySpark

Goto File=> Settings => Add Content Root (Add python and py4j) then write the code:

```
from pyspark.sql import SparkSession
spark=
SparkSession.builder.master('local').appName('Test').getOrCrea
te()
print("Spark Object is created")

rdd=spark.sparkContext.parallelize([1,2,3])
print(rdd.first())
```

OUTPUT:

```
C:\PySpark_Installed\PythonProject\.venv\Scripts\python.exe
C:\PySpark_Installed\PythonProject\test.py
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/03/05 11:52:13 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
Spark Object is created
1

Process finished with exit code 0
```

Explanation of the code:

`spark = SparkSession.builder.master('local').appName('Test').getOrCreate()`

- **`SparkSession.builder`:** This is a builder pattern used to configure and create a SparkSession.
- **`.master('local')`:** Specifies the master URL for the Spark application. Here, `'local'` means the application will run locally on your machine.
- **`.appName('Test')`:** Sets the name of the Spark application.
- **`.getOrCreate()`:** Creates a new SparkSession if one does not already exist, or returns the existing one. This ensures that only one SparkSession is active per JVM.

`rdd = spark.sparkContext.parallelize([1, 2, 3])`

- **`spark.sparkContext`:** The `SparkContext` is the entry point for low-level Spark functionality. It is automatically created when you create a SparkSession.

- **`.parallelize([1, 2, 3])`:** Converts the Python list `[1, 2, 3]` into an RDD. The data is distributed across the cluster (or locally in this case) and can be processed in parallel.

```
print(rdd.first())
```

- **`rdd.first()`:** This is an **action** in Spark. Actions trigger the execution of the RDD transformations and return a result to the driver program. In this case, it returns the first element of the RDD.
- **`print()`:** Prints the result to the console.

NOTE:
Error: SparkSession is not created
Solution:

- **Python**: Use Python 3.8 or 3.9 (Spark 3.x is not fully compatible with Python 3.10+).
- **Java**: Use Java 8, 11, or 17 (Spark 3.5.5 supports these versions).

SparkSession is the entry point to programming with Apache Spark.

RDD (Resilient Distributed Dataset) is the fundamental data structure in Apache Spark. It represents an immutable, distributed collection of objects that can be processed in parallel across a cluster.

Debugging Using PyCharm

Debug Option is on the top right.

Step Over: to execute the current line of code in the debugger and move to the next line without stepping into any functions that might be called on that line.

Step Into: Steps into the method to show what's inside it.

Step Into My Code: It will take to the next debug part.

Step Out: Allows you to immediately jump out of the current function you are currently debugging.

Section-4 (HDFS Course)

HDFS stands for **Hadoop Distributed File System**. It is a special type of file system designed to store **huge amounts of data** efficiently. HDFS (Hadoop Distributed File System) follows a **Master-Slave Architecture** to manage and store big data efficiently.

Why Was HDFS Created?

In today's world, companies like **Facebook, Google, Uber, and Microsoft** generate **massive amounts of data** every second. Storing and processing this data is challenging because:

1. **Storage Problem** – Traditional databases cannot handle large-scale data.
2. **Processing Problem** – Even if we store the data, analyzing and processing it takes too much time.

How Does HDFS Solve the Storage Problem?

HDFS solves this by using a **distributed approach**, where data is **split into multiple smaller parts** and stored across multiple machines (**nodes**).

Vertical Scaling vs. Horizontal Scaling

- **Vertical Scaling** – Adding more RAM/CPU to the same machine. But there is a limit.
- **Horizontal Scaling** – Adding more machines (nodes) to increase storage and processing power. **HDFS uses this method.**

How Does HDFS Work?

1. Data is divided into chunks and stored on multiple machines (nodes).
2. Each chunk has multiple copies (replication), so if one machine fails, the data is not lost.
3. Commodity Hardware (cheap computers) can be used instead of expensive servers, making it cost-effective.

Advantages of HDFS

- **Stores huge data –** Splits and distributes data across many computers.
- **Cost-effective –** Uses cheap hardware (commodity hardware).
- **Fault-tolerant –** If one machine fails, copies exist on other machines.
- **Secure –** Provides data security features.

How Does Processing Work?

HDFS is only for **storing** data. To **process** big data, we use:

- **MapReduce** (Old method, slower).
- **Apache Spark** (New method, **100x faster** than MapReduce because it uses in-memory processing).

Key Components of HDFS

1. **Name Node (Master Node)**
2. **Data Nodes (Slave Nodes)**
3. **Secondary Name Node (Backup Node)**

1. Name Node (Master)

- The **Name Node** is the **boss (master)** of the system.
- It manages all the **Data Nodes** (workers/slaves).
- It keeps track of **where files are stored** in the Data Nodes.
- It does not store the actual data, only the **metadata** (information about data).
- Only **one active Name Node** exists at a time.

Example: Imagine a **library** where the **librarian (Name Node)** does not store books but **maintains a catalog** of where each book is kept on the shelves (**Data Nodes**).

2. Data Nodes (Slaves/Workers)

- **Data Nodes store the actual files** in the form of blocks.
- They follow the **instructions of the Name Node** to store, delete, or copy data.
- Multiple Data Nodes exist to **spread the data across different machines**.
- They use **commodity hardware** (cheap computers), which can fail but have backups.

Example: Think of **shelves in a library** where books (data) are stored. The librarian (Name Node) directs people where to find books, but the books themselves are on the shelves (Data Nodes).

3. Secondary Name Node (Backup Node)

- If the **Name Node fails**, the whole system will stop working.
- To prevent this, we have a **Secondary Name Node** as a backup.
- It takes **regular snapshots (checkpoints)** of metadata to **restore the system if the Name Node crashes**.

Example: Think of the **library's backup records**. If the librarian loses the catalog, a backup record (Secondary Name Node) helps restore it.

How Does HDFS Work?

1. A **client requests a file** from HDFS.
2. The **Name Node checks the metadata** and tells the client which Data Node has the file.
3. The client **reads or writes** the file from the Data Node.
4. If a Data Node fails, another copy is already available in a different Data Node.

Fault Tolerance & Backup

- **Replication:** HDFS makes multiple copies of each data block (default is 3 copies).
- **Edit Log & Image File:**
    - **Edit Log** – Records recent changes.
    - **Image File** – Keeps track of all changes since the system started.
- **Secondary Name Node** combines these logs every hour to update the system.

What is a Data Block?

- A **data block** is a **small piece** of a large file.
- Instead of storing a huge file as a single unit, HDFS **splits it into smaller blocks**.
- The default block size in **Hadoop 2.x and above = 128 MB**
- In **Hadoop 1.x = 64 MB**
- Block size can be **changed** in the configuration file **HDFS-SITE.XML**

Example: Imagine you have a **big book**. Instead of storing it as one **giant book**, you **divide it into chapters** (blocks) and store them separately.

How Does HDFS Split a File?

Let's say we have a file called **data.txt** of **400 MB**.

- HDFS will divide it into **four blocks**:
    - **Block 1** → 128 MB
    - **Block 2** → 128 MB
    - **Block 3** → 128 MB
    - **Block 4** → 16 MB (Remaining data)

Example: Think of a **movie** stored in multiple CDs instead of a single disc.

What is Replication?

- **Replication means making multiple copies of data** to prevent data loss.
- By **default, HDFS keeps 3 copies** of each block.
- If one **Data Node fails**, another copy is used automatically.

Example: Imagine taking **3 photocopies of an important document**. If you lose one, you can use another copy.

How Replication Works?

Imagine we have **four Data Nodes** storing our blocks:

Block	Node 1	Node 2	Node 3	Node 4
Block A	✅	✅	❌	✅
Block B	✅	✅	✅	❌
Block C	✅	✅	✅	❌
Block D	✅	❌	✅	✅

- If **Node 1 fails**, Block A is **still available** in Node 2 and Node 4.
- HDFS **automatically selects the closest copy** to process the data.

Configuring Replication

- The default **replication factor = 3**
- Can be **changed** in **HDFS-SITE.XML**
- **For important files**, replication can be set to **4 or more**.
- **For less important files**, it can be set to **1 or 2** to save storage.

Example: Think of **bank transaction data** (needs high replication) vs **temporary log files** (low replication).

Rack Awareness in HDFS

When a **data node fails**, we need to get its data from another node where a **replica (copy)** of the data exists. But where should these replicas be stored?

To store them efficiently, **HDFS follows a concept called Rack Awareness**.

What is a Rack?

A **rack** is a collection of **30-40 data nodes** (computers storing data). These racks are physically placed in different locations inside a data center.

Why is Rack Awareness Important?

If we store all the replicas **on the same rack**, and that rack fails, we will **lose all the replicas**. To **avoid data loss**, Hadoop distributes replicas **across different racks**.

How Does HDFS Store Replicas?

1. The **first replica** is stored on **one rack** (e.g., **Rack 1**).
2. The **next two replicas** are stored **on a different rack** (e.g., **Rack 2**).
3. This ensures that **if an entire rack fails, other copies are still available**.

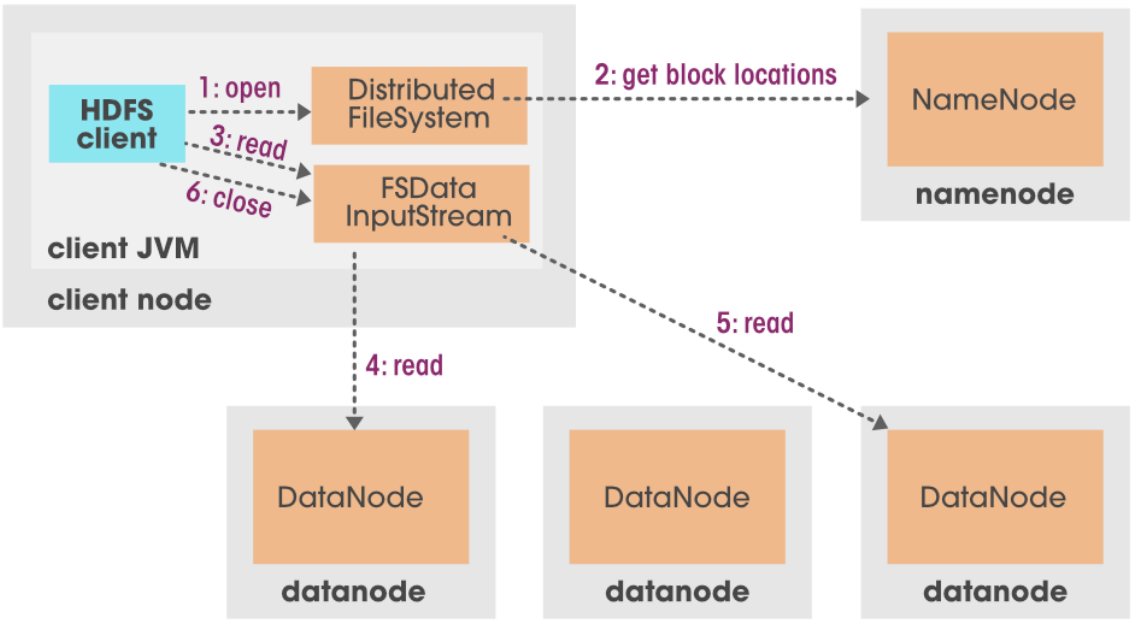
Example:

- Suppose we have **3 racks**, and each rack has **4 data nodes**.
- If **Block A** is stored in **Data Node 1 (Rack 1)**, its replicas **must** be in another rack (e.g., Rack 2 or Rack 3).
- HDFS **avoids placing all three copies in the same rack** to prevent total data loss.

Why Does HDFS Place Two Replicas on the Same Rack?

HDFS places two replicas **on the same rack** to **reduce network delays** and **speed up data processing**. But it ensures that at least **one replica is on a different rack** for safety.

HDFS Reading Mechanism Architecture



Step 1: Client Sends a Read Request

- The **client (user or application)** wants to read a file from HDFS.
- It calls an **"open" method**, which is like knocking on HDFS's door to ask for the file.

Step 2: NameNode Provides Information

- The **NameNode** (the main controller in HDFS) checks:
  - **Is the client allowed to read this file?**
  - **Where is the file stored?** (It provides the **IP addresses of the DataNodes** that have the file's blocks.)
  - It also gives an **authorization token** to allow access.

Step 3: Client Starts Reading the Data

- Now, the **client knows where the file is stored** and starts reading it **block by block**.
- A **special object called FSDataInputStream** helps in reading.
- The **first block** is read from **one DataNode**.

Step 4: Reading Continues Until the File is Complete

- The client **reads one block at a time**.

- After reading **one block**, it **closes the connection** with that DataNode.
- Then, it **connects to the next DataNode** and reads the **next block**.
- This process **repeats** until the entire file is read.

Step 5: Closing the Connection

- Once the file is **completely read**, the client calls the **"close" method**.
- This **removes all temporary objects and tokens** used during the process.

Hadoop Commands:

Hadoop provides a command Line Interface to interact with HDFS.

List all HDFS Commands: hadoop fs or hdfs dfs

Basic usage for any command: hadoop fs -usage ls

Full detailed information for any commands: hadoop fs -help ls

Bring Data from GitHub to local to HDFS

Clone the GitHub repository: git clone “repository link”

To list all the files recursively in the directory: ls -lrt

To install unzip: sudo apt install unzip

To unzip a file: unzip file1.zip

To create directories: hadoop fs -mkdir -p “directory path”

To upload multiple files from the local file system to HDFS: hadoop fs -put datafiles/* “directory path”

Create or remove directories in HDFS

Remove a directory if it is empty: rmdir

Delete files and directories: rm

To create a folder: mkdir

To copy files from HDFS to local files: copyToLocal or get

- **Syntax:** hadoop fs -copyToLocal <HDFS_source> <local_destination>

Recursively list the contents of directories: -R

- **Syntax:** hadoop fs -ls -R practice/retail_db

Display the paths of files and directories only: -C

- **Syntax:** hadoop fs -ls -C practice/retail_db

Reverse the order of the sort: -r

- **Syntax:** hadoop fs -ls -r practice/retail_db

Sort files by size: -S

- **Syntax:** hadoop fs -ls -S practice/retail_db

Sort files by modification time (most recent first): -t

- **Syntax:** hadoop fs -ls -t practice/retail_db

Displays the first few lines of a file in HDFS: -head

- **Syntax:** hadoop fs -head <HDFS_file_path>

Displays the last few lines of a file in HDFS: -tail

- **Syntax:** hadoop fs -tail <HDFS_file_path>

Displays the full content of an HDFS file: -cat

- **Syntax:** hadoop fs -cat <HDFS_file_path>

Prints statistics related to any file/ directory: -stat

- **Syntax:** hadoop fs -stat [format] <HDFS_file_path>
- Formats:

Format Specifier	**Description**
%b	File size in bytes
%F	Type of file (regular file or directory)
%o	File owner
%g	Group name
%r	Replication factor
%y	Last modification date

%n	File name

To change file or directory permissions: chmod

- **Syntax:** hadoop fs -chmod <mode> <HDFS_file_or_directory>

In octal format, permissions are represented using three digits:

- **First digit** → Owner permissions
- **Second digit** → Group permissions
- **Third digit** → Others (everyone else) permissions

In symbolic format, permissions are changed using:

- **u** → User (owner)
- **g** → Group
- **o** → Others
- **a** → All (user, group, others)
- **+** → Add permission
- **-** → Remove permission
- **=** → Set exact permission

Section-6 (Introduction to Spark)

Why was Spark Developed?

Whenever a new technology is created, it is usually to **solve an existing problem**. Similarly, Spark was developed to **overcome the limitations of older systems** like Hadoop MapReduce.

Challenges in Big Data Processing

1. **Scaling Issues**: As data increases, processing time should **not increase drastically**. Traditional databases like **Oracle and SQL Server** couldn't handle massive data efficiently.
2. **Hadoop MapReduce Limitations**:
    - **Complex Management**: Hard to manage and configure.
    - **Slow Processing**: Reads and writes data to **HDFS after every step**, making it **very slow**.
    - **Limited Functionality**: Only supports **Map and Reduce** operations, missing many useful functions like **join, filter, union**.
    - **Only Supports Java**: Could not support **Python, R**, which are widely used in **Data Science & Machine Learning**.
    - **Batch Processing Only**: It cannot handle **real-time streaming or interactive SQL queries**.

How Spark is Different from Hadoop MapReduce

- Instead of **reading/writing** data from HDFS **after each step**, **Spark processes data in-memory** (RAM).
- Intermediate results **are stored in memory**, reducing **disk I/O operations**, making it **much faster** than Hadoop.

How Spark Works

1. Spark **reads data from HDFS**.
2. It **performs all operations in memory** (RAM) instead of writing back to disk.
3. The final output is **written to HDFS only once**, making processing **100x faster** than Hadoop.

Development of Spark

- **Before Spark**, different tools were developed to solve different problems:
    - **Hive** → For SQL-like queries

    - **Storm** → For real-time streaming
    - **Impala** → For faster queries
    - **Mahout** → For Machine Learning

This created **complexity** as each tool had **its own APIs and configurations**.

The Birth of Spark

- In **2009**, researchers at **UC Berkeley** wanted to create a **single unified system** for all these needs.
- They used **Hadoop MapReduce as a base** and improved it to **create Spark**.
- Spark supports **multiple programming languages** like **Python, R, Java, and SQL**.
- In **2013**, Spark became popular, and its creators **founded Databricks**.
- In **2014**, Apache released **Spark 1.0**, marking the first stable version.

Spark Release History

Version	Release Date
0.5	June 2010
1.0	May 2014
2.0	July 2016
3.1	March 2021

Why Spark is Better?

- **Faster Processing** → Uses **in-memory** computation.
- **Supports Multiple Languages** → Python, R, Java, SQL
- **Handles Real-time Data** → Unlike Hadoop, which is **only for batch processing.**
- **Easy to Use** → Simple APIs for **big data processing.**

What is Spark?

- Apache Spark is a unified analytics engine designed for big data processing.
- It can run on-premises (local servers), data centers, or in the cloud.

- Spark processes data 100 times faster than Hadoop because it stores intermediate results in memory (RAM) instead of reading and writing to disk repeatedly.

The Four Major Components of Spark

1. Spark SQL → For working with structured data (like SQL queries on big data)
2. Spark Streaming → For processing real-time data (live data streams)
3. MLlib → A machine learning library for Spark
4. GraphX → A graph processing library for Spark

How Spark is Used

- **You can run Spark on different cluster managers, such as:**
  - **YARN (Hadoop-based manager)**
  - **Kubernetes**
  - **Mesos**
  - **EC2 (Amazon Cloud)**
  - **Standalone mode (without external cluster managers)**

What are Spark Components?

Apache Spark is designed to work with any cluster manager (a system that manages computing resources). It supports five different cluster managers:
1. Hadoop YARN → Used in Hadoop-based environments
2. Mesos → A general-purpose cluster manager
3. Kubernetes → Used for managing containers
4. Amazon EC2 → Cloud-based Spark cluster
5. Standalone Mode → Runs without external cluster managers

Spark Core (The Foundation of Spark)

At the base of Spark, there is Spark Core, which provides fundamental APIs (functions that allow interaction with Spark). These include:

- **Transformations** → Operations like map, reduce, join, filter to process data
- **Actions** → Commands that execute the transformations and return results
- **RDDs (Resilient Distributed Datasets)** → The basic data structure in Spark that allows parallel data processing

Major Components of Spark

1. Spark SQL & DataFrames

- Allows working with structured data (like tables in a database)
- Supports SQL queries on large datasets
- Used for batch processing (processing large amounts of data at once)

2. Spark Streaming

- Processes real-time streaming data (e.g., live stock prices, social media feeds)
- Works on top of DataFrames

3. MLlib (Machine Learning Library)

- Provides tools for machine learning, such as:
    - Classification
    - Regression
    - Clustering
    - Recommendation systems

4. GraphX

- Used for graph-based computations (e.g., social network analysis, shortest path algorithms)
- Supports parallel graph processing

Section-7 (Introduction to Spark Session)

What is Spark Session?

Spark Session is an **entry point** to work with Spark functionalities like **DataFrames, SQL, Streaming, and Machine Learning**. It was **introduced in Spark 2.x** and is now recommended over older context classes like:

- **Spark Context**
- **SQL Context**
- **Hive Context**

Why Use Spark Session?

Before Spark 2.x, we had to use **different context classes** for different tasks:

- **Spark Context** → Used for working with RDDs
- **SQL Context** → Used for working with structured data (SQL queries, DataFrames)
- **Hive Context** → Used for additional Apache Hive functionalities

Now, **Spark Session combines all these into a single object** (unified access), making it easier to manage and use.

Comparison: Spark 1.x vs. Spark 2.x

Feature	**Spark 1.x (Before 2.0)**	**Spark 2.x (After 2.0)**
Entry Point	SparkContext, SQLContext, HiveContext	SparkSession (Unified)
Ease of Use	Had to manage multiple contexts	Everything is under SparkSession
Recommended?	Not Recommended	Recommended

Understanding Classes, Objects, and Instances in Spark

Think of **SparkSession** as a **blueprint (Class)** for creating **Spark objects (Instances)**.

Example:

- **Class (Blueprint)** → A **house design**
- **Object (Instance of the Class)** → Your **own house** built from the blueprint
- **Multiple objects** (houses) can be built from the same blueprint

Similarly, in Spark:

- **SparkSession** is a class
- **spark** is an object (instance of **SparkSession**)
- This object gives us access to all Spark functionalities

Using Spark in Different Versions

1. **Spark 1.x (Before 2.0) → Using `SparkContext`**

```
sc = SparkContext("local", "MyApp")
```

   - **sc** is an object of **SparkContext**
   - This was the main way to access Spark features in **Spark 1.x**
2. **Spark 2.x (After 2.0) → Using SparkSession**

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

   - **spark** is an object of **SparkSession**
   - This is now the recommended way to use Spark

NOTE:

- We can access the **SparkContext** inside **SparkSession** by:

  sc = spark.sparkContext

Section-8 (RDD Fundamentals)

What is RDD?

RDD stands for **Resilient Distributed Dataset**. Let's break it down:

1. **Resilient** – It can recover lost data if something goes wrong.
2. **Distributed** – Data is divided into small parts and processed across multiple machines.
3. **Dataset** – It holds data.

Key Features of RDD:

1. **Resilient** – If data is lost, Spark can rebuild it using something called **lineage** (a history of transformations).
2. **Distributed** – Data is broken into **partitions** and spread across different machines.
3. **Lazy Evaluation** – Nothing happens until you request an action (e.g., count, collect).
4. **Immutable** – Once created, an RDD cannot be changed.
5. **In-Memory Computation** – Data is processed in RAM instead of slow disk storage.
6. **Works with Structured & Semi-Structured Data** – Can process different types of data.

When to Use RDD?

- When working with **unstructured data** (like text files or media streams).
- When you **don’t care about schema or optimization**.
- When you need **low-level control** over data.

RDD Properties

RDD is defined by 5 main properties:

- List of Parent RDDs (Dependencies)
- An array of partitions that a dataset is divided into.
- A Compute function to do a computation on partitions.
- Optional practitioner that defines how keys are hashed and the pairs partitioned (key value RDDs)
- Optional Preferred locations - Information about the locations of the split block for an HDFS file (if on YARN).

Main Problems in RDD

Problem 1: Cannot optimize

- A partition is going to create an Iterator and it will execute that particular code from that partition and distribute across the clusters.
    - So, the problem in this is **Opaque computation** (i.e., RDD does not know what is the function doing with the data, it just serialize the code, send it over to an executor and execute it).

  Example: If you are trying to join a project then cannot automatically optimize it because of the spark computation.
    - It also contains the **opaque data** (i.e., the data that is stored in the RDDs is also opaque to Spark and Spark cannot do any pruning of data).

  Example:
  Let us see while loading a file of, let's say, 1000 columns. But in the computation, you only require two columns. So added cannot prune the remaining 998 columns. It can still carry it forward till you know you apply a form action. So it still carries forward the whole thousand columns.

Problem 2: Pretty verbose to work with

- It is a very tedious task to understand the code of RDDs. So, we use DataFrames or Spark SQL.

Section-9 (Create RDD)

Ways to create RDD

1. External Data (HDFS, local)

sample.txt

```
Hello,
this is a sample file.
Spark is amazing!
Learning RDDs step by step.
It is fun
```

```python
from pyspark.sql import SparkSession

# Step 1: Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

# Step 2: Read data from a text file (local or HDFS)
rdd = sc.textFile("sample.txt")  # Each line in the file becomes an element in the RDD

# Step 3: Display first few elements
print(rdd.take(5))  # Prints first 5 lines of the file
```

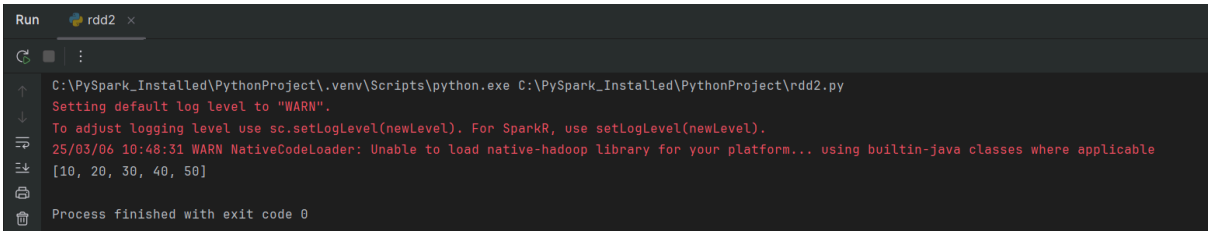
OUTPUT:

```
Run  rdd1
C:\PySpark_Installed\PythonProject\.venv\Scripts\python.exe C:\PySpark_Installed\PythonProject\rdd1.py
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/03/06 10:45:20 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
['Hello,', 'this is a sample file.', 'Spark is amazing!', 'Learning RDDs step by step.', 'It is fun']

Process finished with exit code 0
```

2. Local Data

```python
from pyspark.sql import SparkSession

# Step 1: Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext
# Step 2: Create a Python list
data = [10, 20, 30, 40, 50]

# Step 3: Convert it into an RDD
rdd = sc.parallelize(data)

# Step 4: Display the RDD data
print(rdd.collect())
```

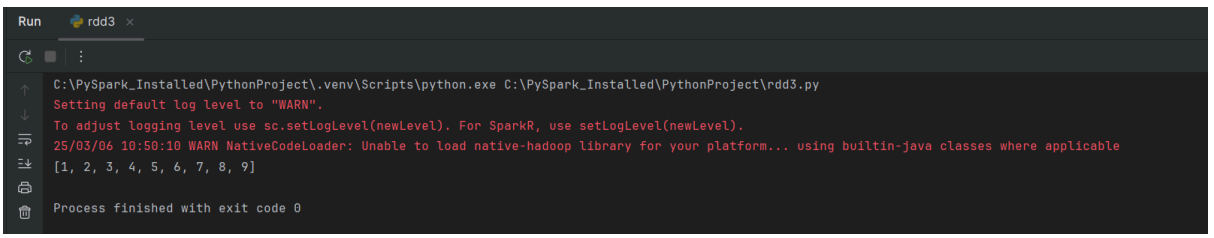
OUTPUT:



3. Python List/ Parallelize Collections

```python
from pyspark.sql import SparkSession

# Step 1: Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

# Step 2: Create a Python list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Step 3: Parallelize the list to create an RDD
rdd = sc.parallelize(numbers)

# Step 4: Display the data
print(rdd.collect())
```

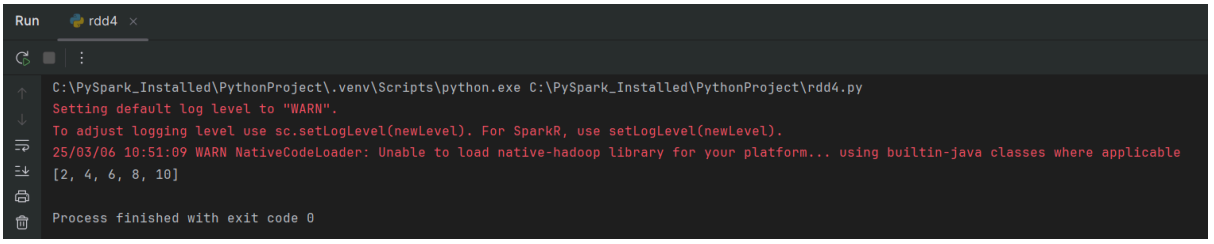
OUTPUT:



4. Other RDDs

```python
from pyspark.sql import SparkSession

# Step 1: Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

# Step 2: Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Step 3: Apply a transformation (filter out even numbers)
even_rdd = rdd.filter(lambda x: x % 2 == 0)

# Step 4: Display the transformed RDD
print(even_rdd.collect())
```

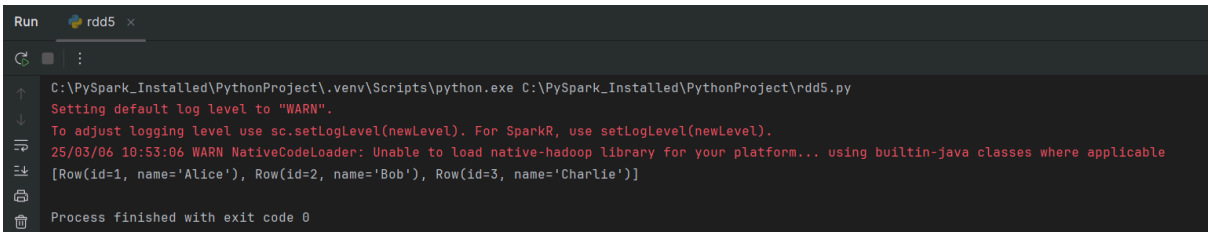
OUTPUT:



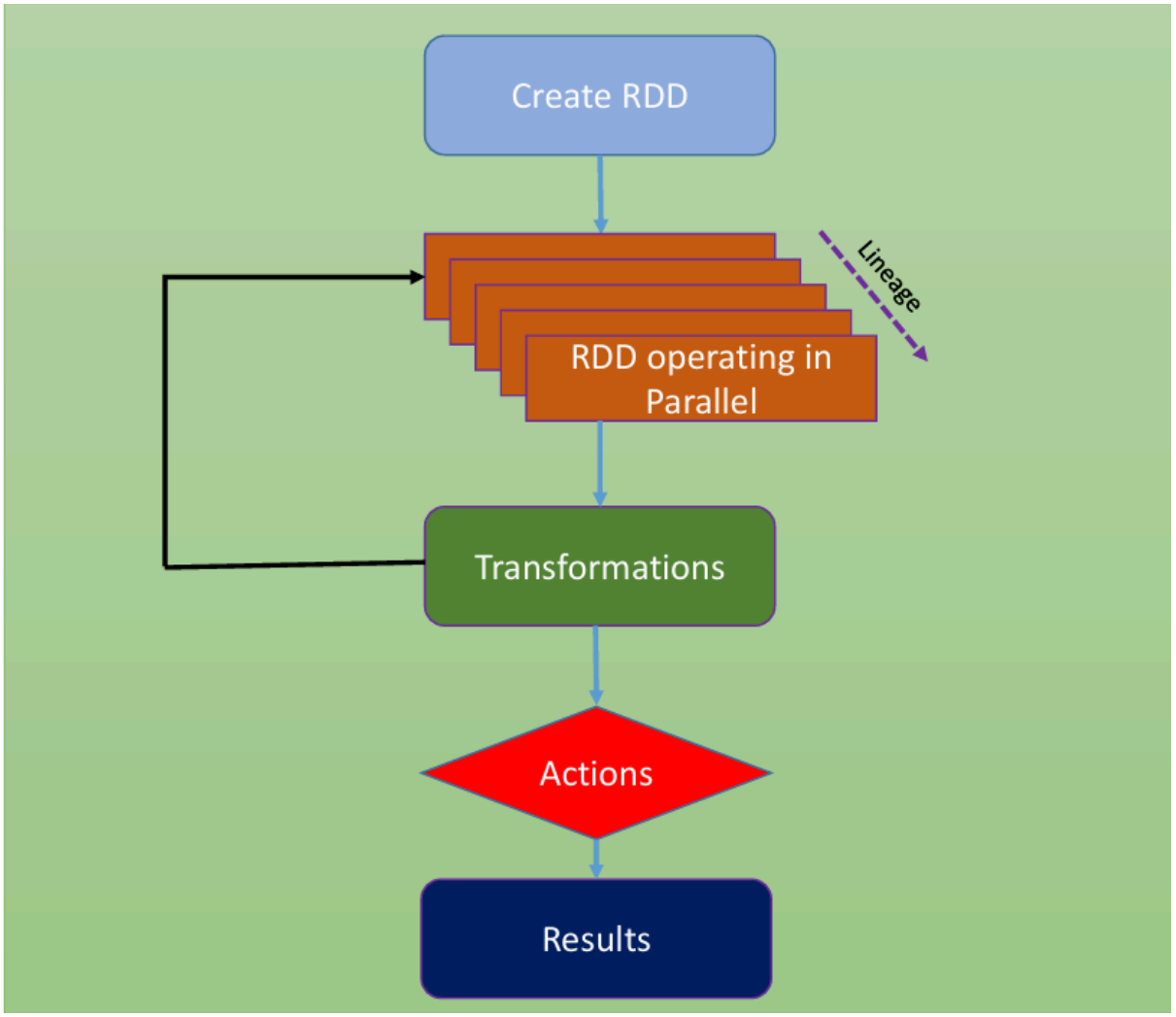
5. Existing DataFrame

```python
from pyspark.sql import Row
from pyspark.sql import SparkSession

# Step 1: Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext
# Step 2: Create a DataFrame
df = spark.createDataFrame(
    [(1, "Alice"), (2, "Bob"), (3, "Charlie")],
    ["id", "name"]
)

# Step 3: Convert DataFrame to RDD
rdd = df.rdd

# Step 4: Display the RDD content
print(rdd.collect())
```

OUTPUT:


Run
rdd5
C:\PySpark_Installed\PythonProject\.venv\Scripts\python.exe C:\PySpark_Installed\PythonProject\rdd5.py
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/03/06 10:53:06 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
[Row(id=1, name='Alice'), Row(id=2, name='Bob'), Row(id=3, name='Charlie')]
Process finished with exit code 0

Section-10 (RDD Operations)



Firstly, we will create an RDD and then we apply a series of transformations. On each transformation an RDD is formed. In this diagram there are 4 to 5 RDDs so there must be 4 to 5 transformations. RDDs store the data. If any data is lost then we can use the linear set to record the data quickly and efficiently.
Transformation is a lazy operation i.e., nothing happens except that spark just keeps the recipe or transformation logics. So, once we apply an action, it would execute all the transformations and gives the results.

Transformations

Row Level	Joining	Key Agg	Sorting	Set	Sampling	Pipe	Partitions
map	join	reduceByKey	sortByKey	union	sample	pipe	Coalesce
flatMap	cogroup	aggregateByKey		intersection			Repartition
filter	cartesian	groupByKey		distinct			repartitionAndSort WithinPartitions
mapValues		countByKey		substract			

Actions			
Display	**Total Agg**	**File Extraction**	**foreach**
take	reduce	saveAsTextFile	foreach
takeSample	count	saveAsSequenceFile	
takeOrdered		saveAsObjectFile	
first			
collect			

Transformations vs. Actions

Transformations (Lazy Operations)

- Transformations **do not execute immediately**. Instead, they create a new RDD.
- They are **only executed when an Action is triggered**.
- Examples: `map()`, `filter()`, `flatMap()`, `join()`, `reduceByKey()`

Actions (Triggers Execution)

- Actions **execute the transformations** and return a result.
- They **collect data from RDDs**.
- Examples: `collect()`, `count()`, `reduce()`, `saveAsTextFile()`

Row-Level Transformations

These operations **modify individual elements of an RDD**.

map()

- Transforms each element into another element.
- **Number of input elements = Number of output elements**.

Example: Extract order IDs from a CSV file

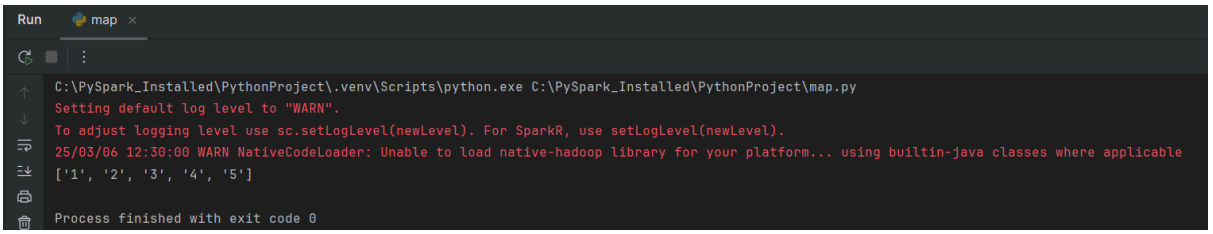
orders.csv

```python
import csv

data = [
    ["1", "2023-01-01 10:30:00", "1001", "CLOSED"],
    ["2", "2023-01-02 11:00:00", "1002", "COMPLETE"],
    ["3", "2023-01-03 12:15:00", "1003", "PENDING"],
    ["4", "2023-01-04 14:45:00", "1004", "CANCELLED"],
    ["5", "2023-01-05 16:20:00", "1005", "CLOSED"]
]

with open("orders.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)

print("orders.csv created successfully!")
```

OUTPUT:


```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

orders = sc.textFile("orders.csv")
order_ids = orders.map(lambda x: x.split(',')[0])
print(order_ids.take(5))  # Shows first 5 order IDs
```

OUTPUT:



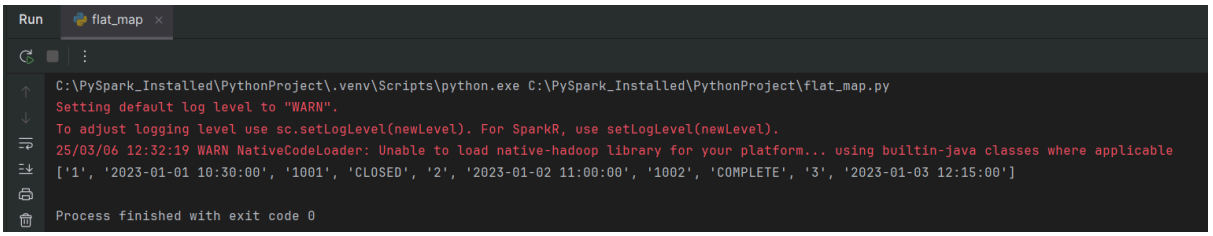
flatMap()

- Similar to `map()`, but **each element can return multiple outputs**.
- The number of output elements **can be more than the number of input elements**.

Example: Word Count

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

lines = sc.textFile("orders.csv")
words = lines.flatMap(lambda x: x.split(','))
print(words.take(10))  # Shows first 10 words
```

OUTPUT:



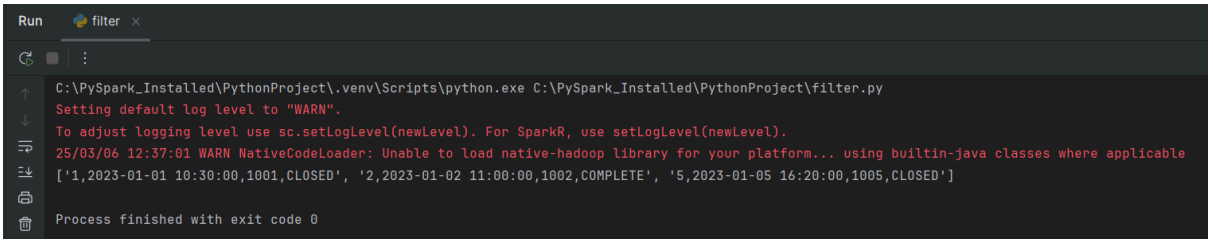
filter()

- Removes elements **that do not meet a condition**.

Example: Get all orders that are "CLOSED" or "COMPLETE"

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

orders = sc.textFile("orders.csv")
filtered_orders = orders.filter(lambda x: x.split(',')[3] in ("CLOSED", "COMPLETE"))
print(filtered_orders.take(5))
```

OUTPUT:



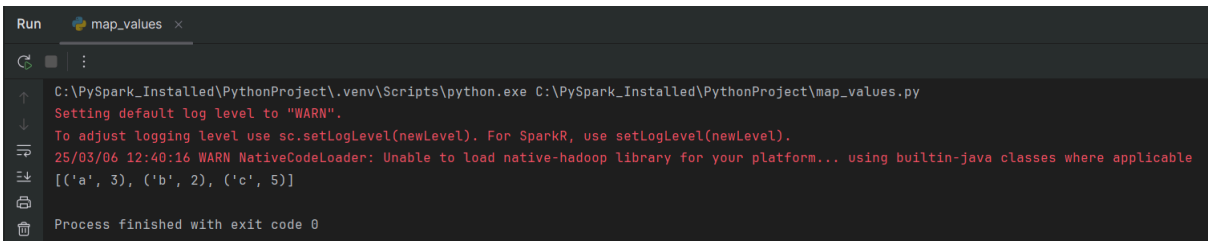
mapValues()

- Changes only the **values** in a pair RDD, **keeping the keys the same**.

Example: Find the length of each list in a Pair RDD

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize([("a", [1,2,3]), ("b", [3,4]), ("c", [1,2,3,4,5])])
length_rdd = rdd.mapValues(len)
print(length_rdd.collect())
```

OUTPUT:



Joins in Spark

Joins help combine two RDDs based on a **common key**.

join()

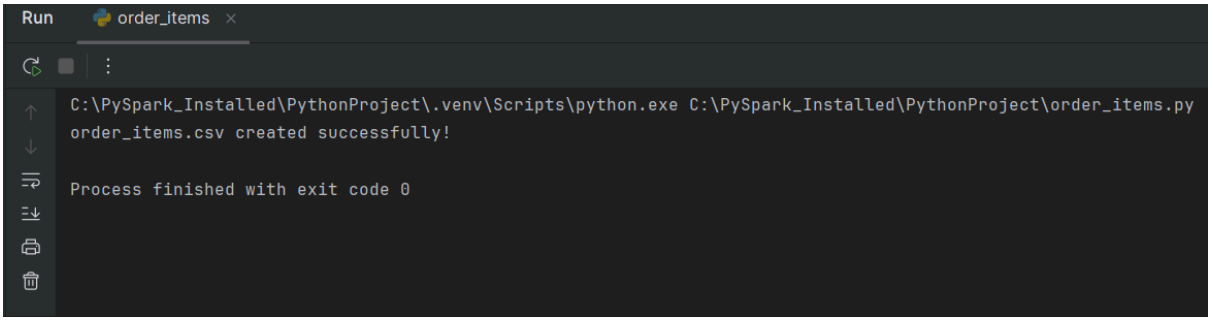
- Combines two RDDs based on **matching keys**.
- Returns **(key, (value1, value2))**.

Example: Join Orders with Order Items

order_items.csv

```python
import csv

data = [
    ["1", "1", "1001", "1", "299.99", "299.99"],
    ["2", "2", "1002", "2", "199.99", "399.98"],
    ["3", "2", "1003", "1", "79.99", "79.99"],
    ["4", "3", "1004", "3", "150.00", "450.00"],
    ["5", "4", "1005", "2", "89.99", "179.98"],
    ["6", "5", "1001", "1", "250.00", "250.00"]
]

with open("order_items.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)

print("order_items.csv created successfully!")
```

OUTPUT:



```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

orders = sc.textFile("orders.csv").map(lambda x: (x.split(',')[0], x.split(',')[2]))
order_items = sc.textFile("order_items.csv").map(lambda x: (x.split(',')[1], x.split(',')[4]))

joined_rdd = orders.join(order_items)
print(joined_rdd.take(5))
```

OUTPUT:

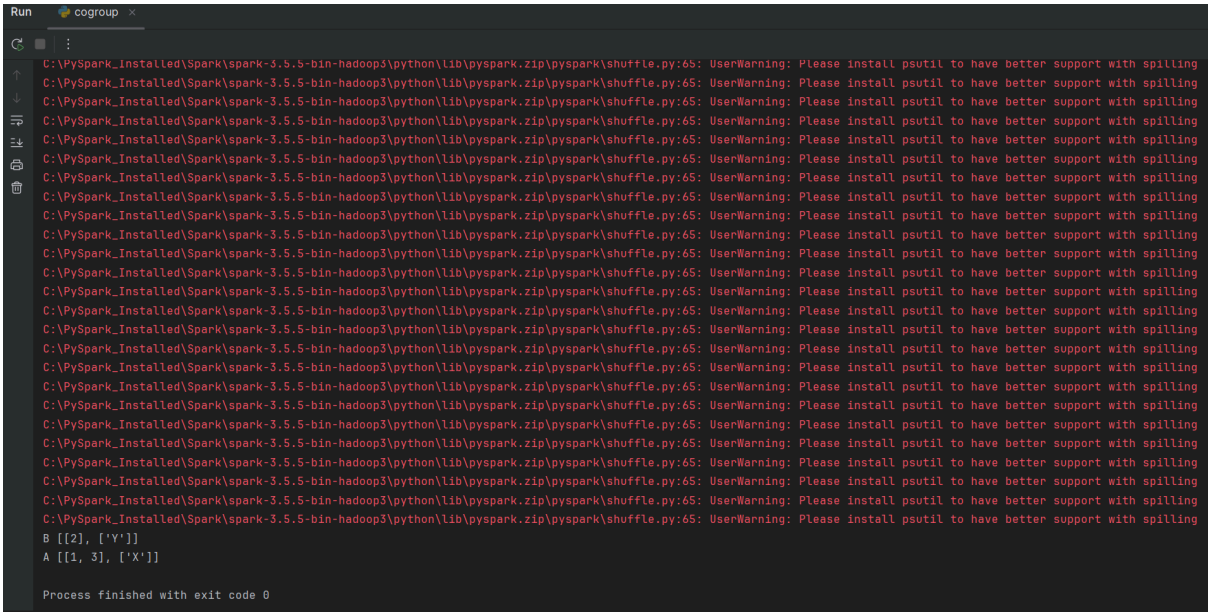
```
C:\PySpark_Installed\PythonProject\.venv\Scripts\python.exe C:\PySpark_Installed\PythonProject\join.py
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/03/06 14:23:32 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\lib\pyspark.zip\pyspark\shuffle.py:65: UserWarning: Please install psutil to have better support with spilling
[('4', ('1004', '89.99')), ('3', ('1003', '150.00')), ('1', ('1001', '299.99')), ('2', ('1002', '199.99')), ('2', ('1002', '79.99'))]

Process finished with exit code 0
```

cogroup()

The **`cogroup()`** transformation is used when **two RDDs** have **the same key** and we want to group their values together.

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd1 = sc.parallelize([("A", 1), ("B", 2), ("A", 3)])
rdd2 = sc.parallelize([("A", "X"), ("B", "Y")])

result = rdd1.cogroup(rdd2)

for key, values in result.collect():
    print(key, list(map(list, values)))
```

OUTPUT:

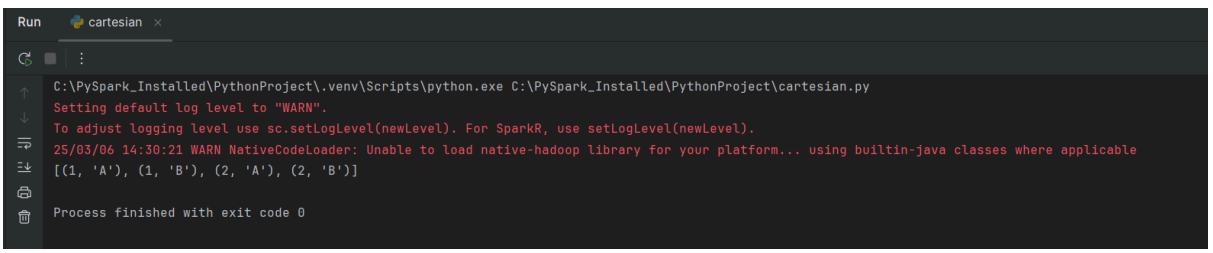


cartesian()

The **`cartesian()`** transformation performs a **cross join** between two RDDs. It creates a **pair of every element from RDD1 with every element from RDD2**.

```
1   from pyspark.sql import SparkSession
2
3   # Create a SparkSession
4   spark = SparkSession.builder.appName("RDDExample").getOrCreate()
5   sc = spark.sparkContext  # Get SparkContext
6
7   rdd1 = sc.parallelize([1, 2])
8   rdd2 = sc.parallelize(["A", "B"])
9
10  result = rdd1.cartesian(rdd2)
11
12  print(result.collect())
```

OUTPUT:



Set Operations on RDDs

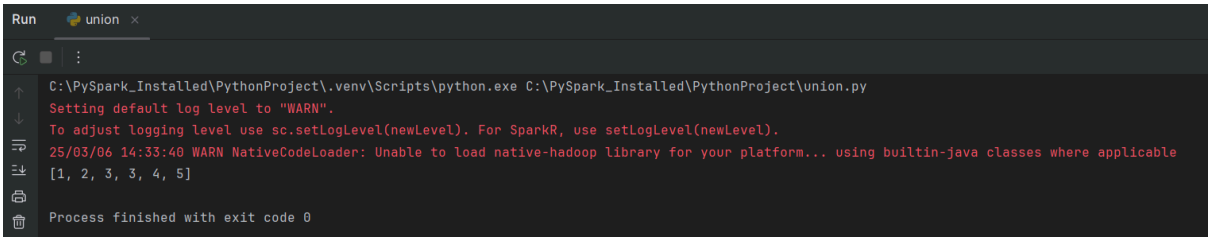
union()

- Combines two RDDs, keeping all elements from both.

Example:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([3, 4, 5])
union_rdd = rdd1.union(rdd2)
print(union_rdd.collect())
```

OUTPUT:



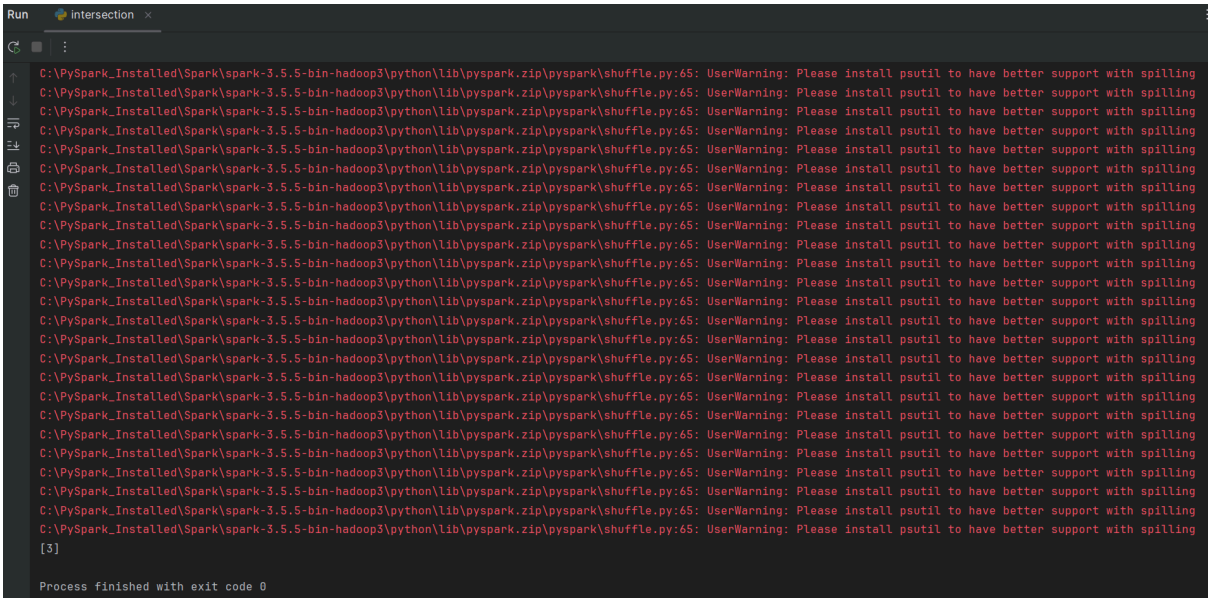
intersection()

- Returns only the **common elements** between two RDDs.

Example:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([3, 4, 5])
intersect_rdd = rdd1.intersection(rdd2)
print(intersect_rdd.collect())
```

OUTPUT:



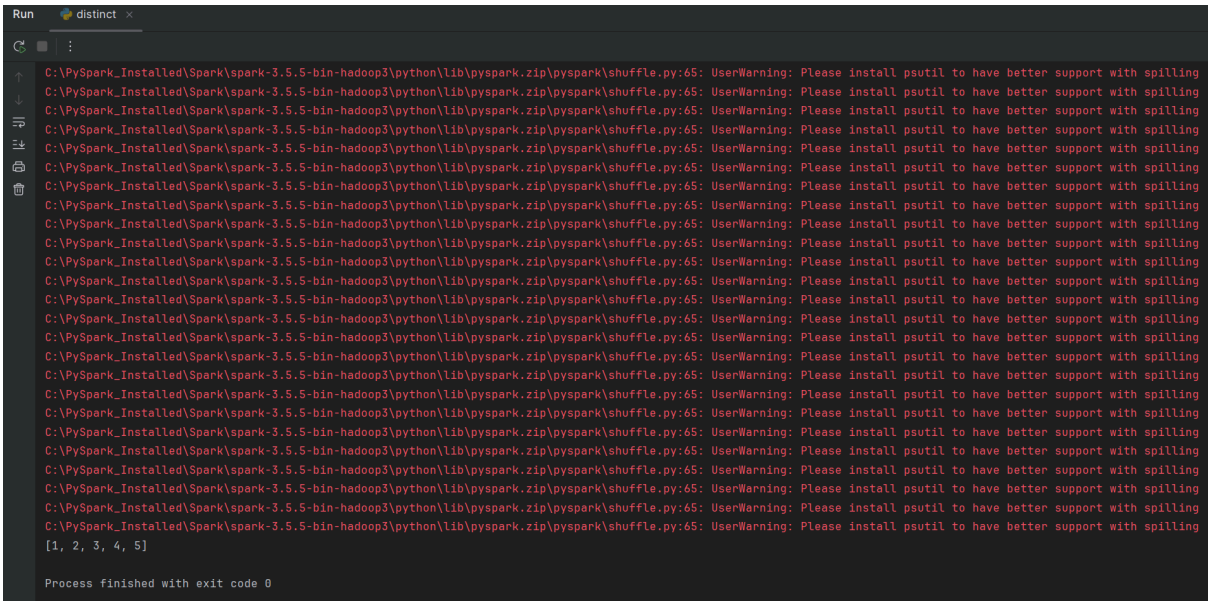
distinct()

- Removes **duplicate elements** from an RDD.

Example:

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([3, 4, 5])
union_rdd = rdd1.union(rdd2)
distinct_rdd = union_rdd.distinct()
print(distinct_rdd.collect())
```

OUTPUT:

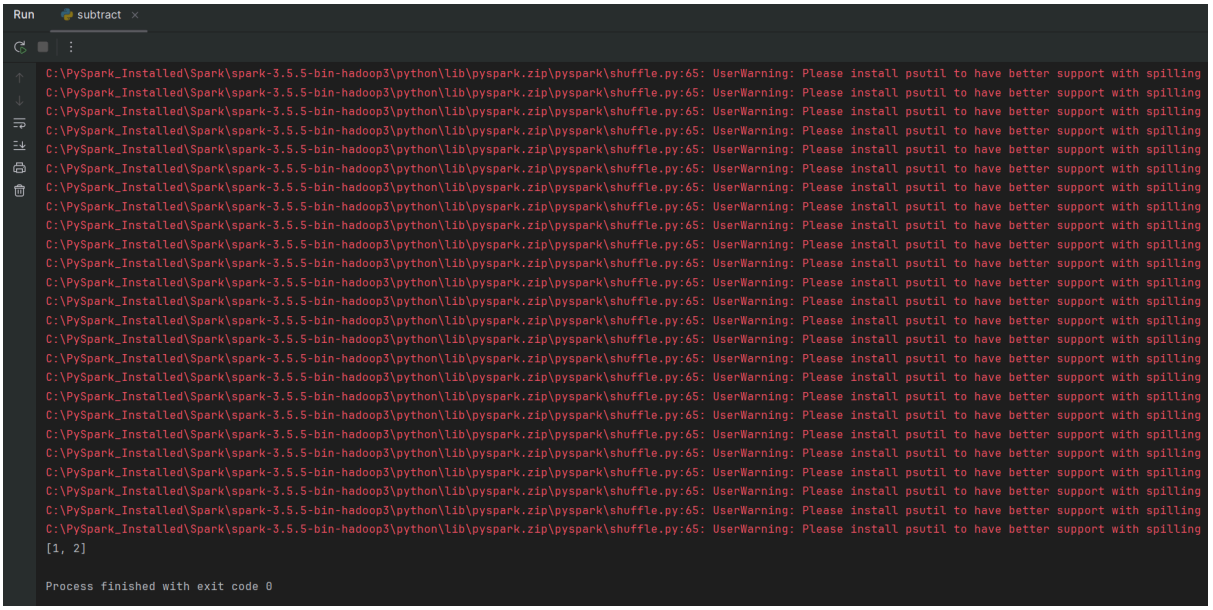


subtract()

- The **`subtract()`** transformation in Spark is used to **remove elements** of one RDD from another.
- It works like the **set difference operation** in mathematics.
- **Only unique elements are considered.** If an element exists in both RDDs, it will be removed.

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd1 = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = sc.parallelize([3, 4, 5, 6, 7])

result = rdd1.subtract(rdd2)

print(result.collect())
```

OUTPUT:



Sorting

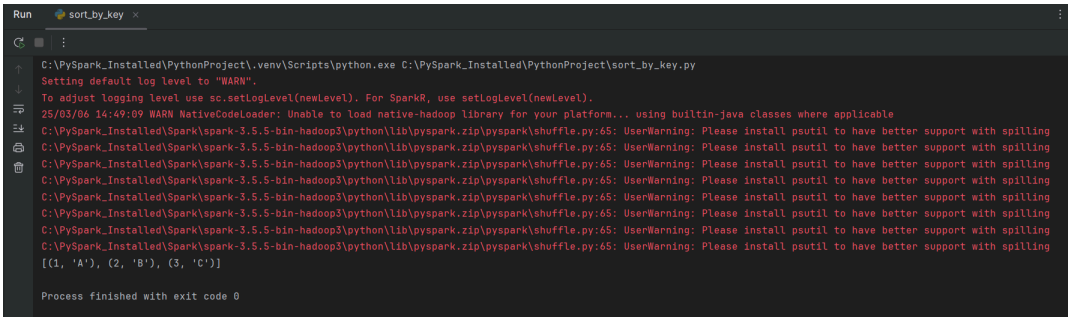
sortByKey()

- Sorts RDD based on **keys** in a pair RDD.

Example:

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize([(3, "C"), (1, "A"), (2, "B")])
sorted_rdd = rdd.sortByKey()
print(sorted_rdd.collect())
```

OUTPUT:



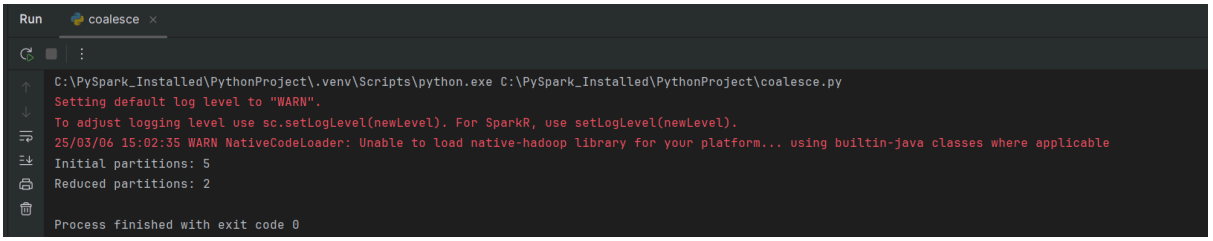
Partitioning

coalesce()

- **Used to decrease the number of partitions** in an RDD.
- **Avoids full shuffle** when **`shuffle=False`**, which makes it **faster** but can create **uneven partitions**.
- If **`shuffle=True`**, it **behaves like `repartition()`** (i.e., performs a full shuffle).
- Typically used **after filtering** to reduce partition count.

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize(range(1, 11), numSlices: 5)  # Creates an RDD with 5 partitions

print("Initial partitions:", rdd.getNumPartitions())

rdd_coalesced = rdd.coalesce(2)  # Reduce partitions to 2

print("Reduced partitions:", rdd_coalesced.getNumPartitions())
```

OUTPUT:

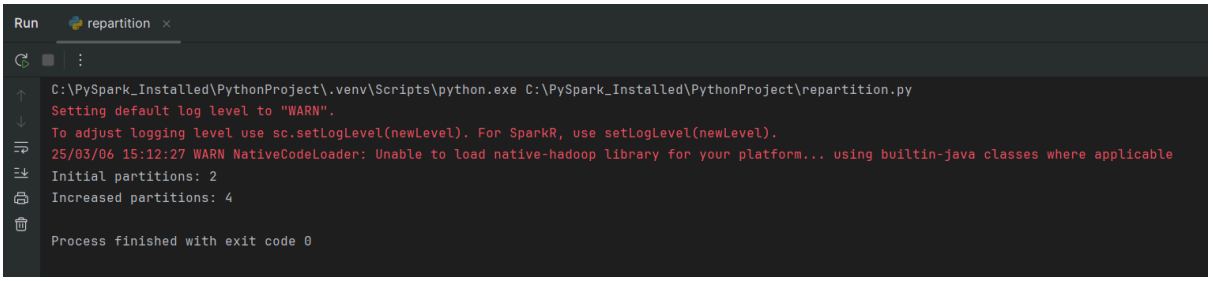


repartition()

- **Used to increase or decrease partitions**.
- **Always performs a full shuffle**, redistributing data evenly.
- More **balanced** compared to `coalesce()` but **slower** due to shuffling.

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize(range(1, 11), numSlices: 2)  # Creates an RDD with 2 partitions

print("Initial partitions:", rdd.getNumPartitions())

rdd_repartitioned = rdd.repartition(4)  # Increase partitions to 4

print("Increased partitions:", rdd_repartitioned.getNumPartitions())
```

OUTPUT:



Repartition	Coalesce
Repartitions do a full shuffle.	Coalesce avoids full shuffle.
Preferably used to increase the number of partitions.	Preferably used to decrease the number of partitions.
Repartition results in roughly equal sized partitions.	Coalesce results in partitions with different size of data.
Coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions	Coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions.

Aggregation Operations

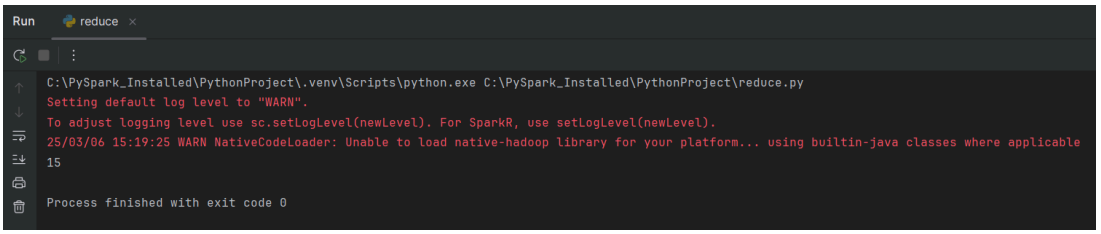
reduce()

- Reduces all elements in an RDD using a **function**.

Example: Find sum of numbers

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize([1, 2, 3, 4, 5])
sum_rdd = rdd.reduce(lambda a, b: a + b)
print(sum_rdd)
```

OUTPUT:



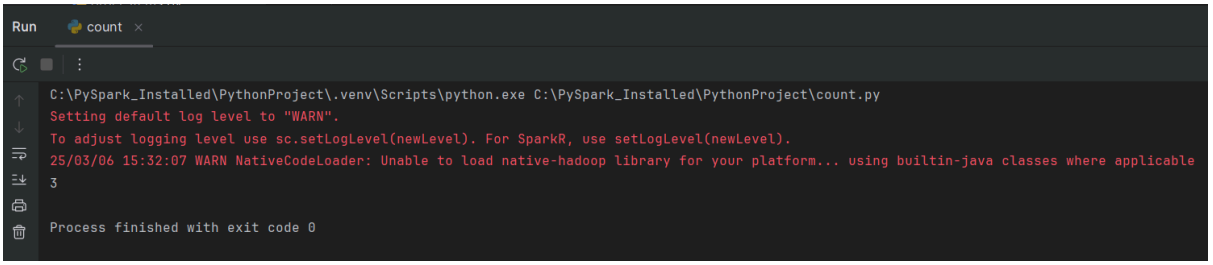
count()

- Returns the **number of elements** in an RDD.

Example:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext
rdd = sc.parallelize(["apple", "banana", "cherry"])
print(rdd.count())
```

OUTPUT:



Actions for Displaying Data

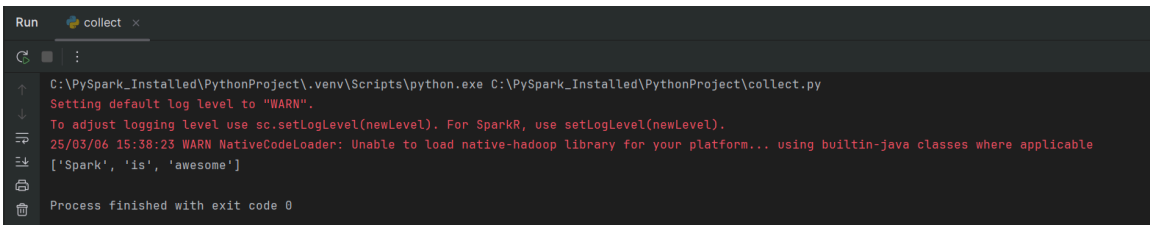
collect()

- Returns **all elements** from an RDD.

Example:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize(["Spark", "is", "awesome"])
print(rdd.collect())
```

OUTPUT:



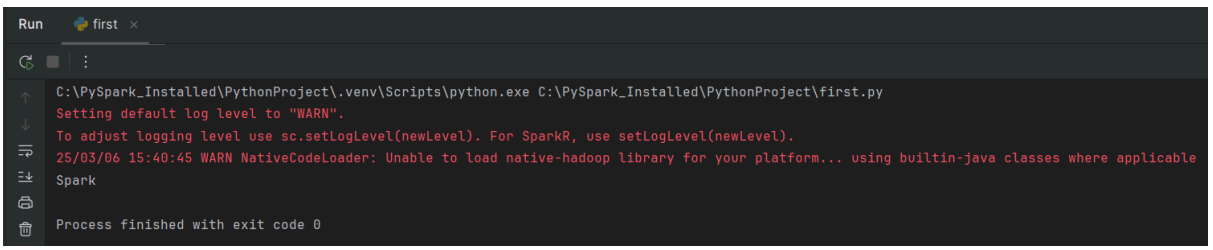
first()

- Returns the **first element**.

Example:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize(["Spark", "is", "awesome"])
print(rdd.first())
```

OUTPUT:



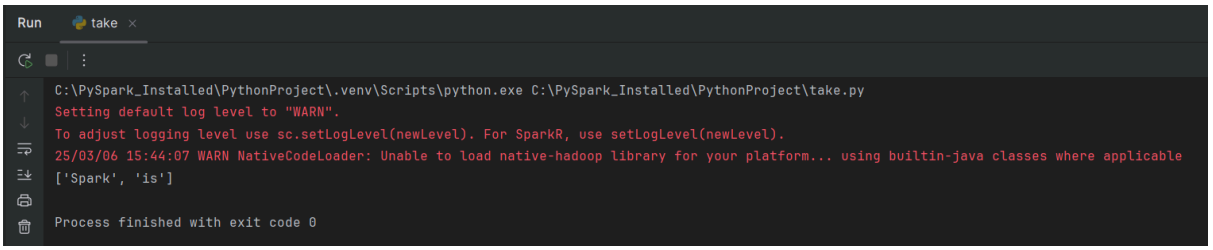
take(n)

- Returns the **first n elements**.

Example:

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("RDDExample").getOrCreate()
sc = spark.sparkContext  # Get SparkContext

rdd = sc.parallelize(["Spark", "is", "awesome"])
print(rdd.take(2))
```

OUTPUT:



Shuffling and Combiner in Spark

What is Shuffling?

Shuffling in Spark means **moving data across different partitions or even nodes** in a cluster. This happens when Spark needs to group data by a key, such as in `groupByKey()`, `reduceByKey()`, or `join()` operations.

Key Points about Shuffling:

- **Creates a new stage** in the processing.
- **Can be slow** because it involves **disk I/O, network I/O, and data serialization/ deserialization**.
- **Operations that cause shuffling:**
  - `groupByKey()`, `reduceByKey()`, `join()`, `distinct()`, etc.
- **Operations that do not cause shuffling:**
  - `count()` and `countByKey()`.
- **Avoid shuffling whenever possible** to improve performance.
- If shuffling is necessary, use a **combiner** to reduce its cost.

What is a Combiner?

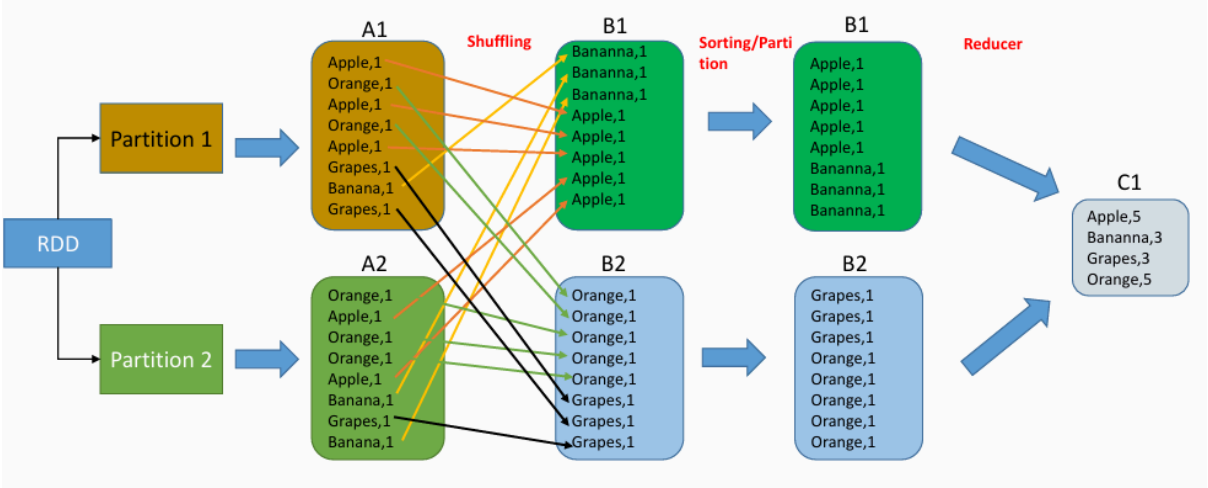
A **combiner** helps to **reduce the amount of data shuffled** across the network. It does this by **performing intermediate computations within each partition** before shuffling.

Key Benefit of a Combiner:

- It **groups values locally in each partition** before sending data across nodes, which minimizes network traffic.
- **`reduceByKey()` and `aggregateByKey()` use combiners**, so they are preferred over `groupByKey()` (which does not use a combiner and causes more shuffling).

How Does Shuffling Work?

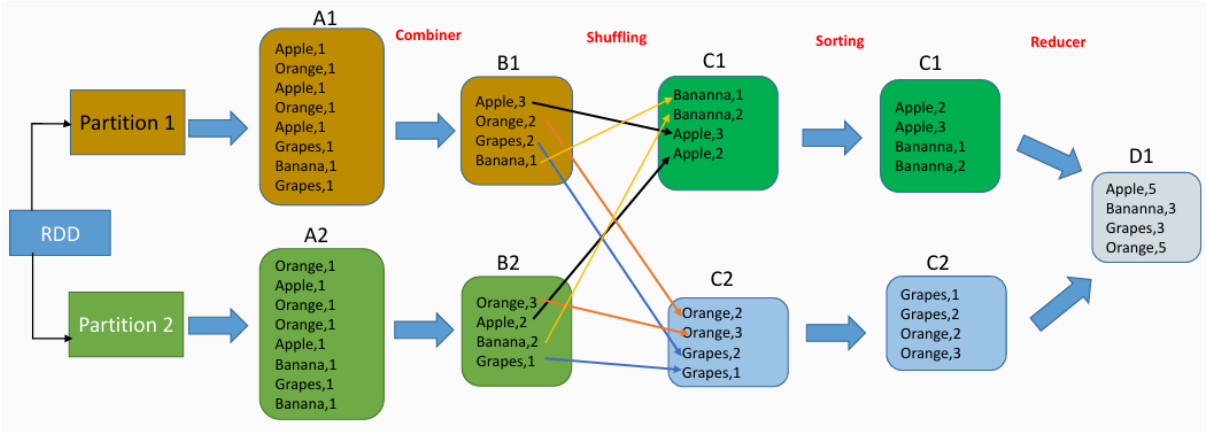
Without a Combiner (Inefficient Shuffling)



- This method **shuffles too much data**, slowing down performance.

With a Combiner (Efficient Shuffling)

Instead of sending all individual values across the network, Spark **aggregates them locally first**



- **Less data transfer = Faster performance!**

How to Detect and Reduce Shuffling?

- Use `.toDebugString` to check if shuffling is happening.
- For **DataFrames**, use `.explain()` to analyze the query plan.
- Reduce the number of partitions using `repartition()` or `coalesce()`.
- Use `reduceByKey()` instead of `groupByKey()` for aggregations.

Aggregation Operations (Keys)

groupByKey()

- Groups values based on a key.
- Returns **(key, list of values)**.
- **Not efficient** because it does **not use a combiner** (increases memory usage and shuffling).
- **Better alternatives**: `reduceByKey()` and `aggregateByKey()`.
- Example: Find total revenue for each product.

reduceByKey()

- Groups values by key and **reduces them using a function** (e.g., sum, max).
- Uses a **combiner**, making it **more efficient** than `groupByKey()`.

aggregateByKey()

- Similar to `reduceByKey()`, but allows the **output type to be different from the input type**.
- Uses three arguments:

- **Zero Value**: Initial value (e.g., `0` for sum, `None` for collections).
- **SeqOp**: Function to accumulate values per partition.
- **CombOp**: Function to combine results across partitions.

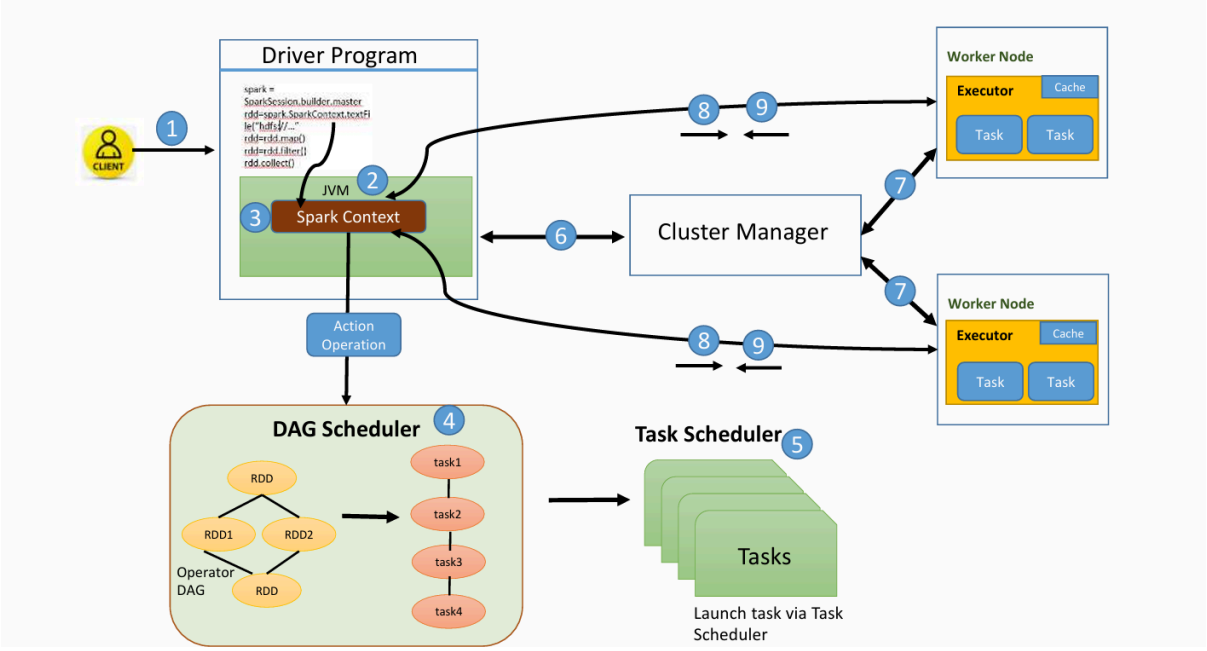
countByKey()

- Counts how many times each key appears.
- **No shuffle**, returns a dictionary.

Section-11 (Spark Cluster Execution Architecture)

Full Architecture

Apache Spark follows a **Master-Slave** architecture, meaning there is a central component (**Driver**) that controls multiple worker components (**Executors**). Here's how it works step by step:



Step-by-Step Execution Flow

1. **Client Submits Code to the Driver**
   - The user writes a Spark program and submits it for execution.
   - The **Driver** is responsible for managing the execution of this program.
2. **JVM (Java Virtual Machine) is Created on the Driver**
   - A JVM is started where the driver program runs.
3. **SparkContext is Created**
   - Inside the driver's JVM, **SparkContext** is created.
   - It is the main entry point that connects Spark with the cluster.
4. **Logical DAG (Directed Acyclic Graph) is Created**
   - Spark internally converts the user's code into a **logical execution plan** called a DAG.
   - **DAG Scheduler** optimizes the tasks (e.g., combining transformations) and converts it into a **physical execution plan** with multiple stages.
   - Each stage consists of **tasks** (small units of work).
5. **Task Scheduler Sends Tasks to Cluster Manager**
   - The **Task Scheduler** assigns the tasks and submits them to the **Cluster Manager**.
6. **Driver Communicates with Cluster Manager**
   - The driver requests resources (CPU, memory, etc.) from the **Cluster Manager** to execute tasks.

7. **Cluster Manager Allocates Resources**
    - It assigns **Worker Nodes** and **Executors** to run the tasks.
8. **Driver Sends Code & Dependencies to Executors**
    - The driver sends the **application code and required files** (JAR or Python files) to the executors.
9. **Executors Start Execution**
    - The executors execute the tasks assigned to them.
    - They **register** with the driver so that the driver knows their status.
    - Intermediate data/results can be **cached** in worker nodes for efficiency.
10. **Driver Monitors Execution**
    - The driver keeps track of executor progress.
    - It schedules future tasks based on **data location** to optimize performance.

11. **Results are Returned to the Driver**

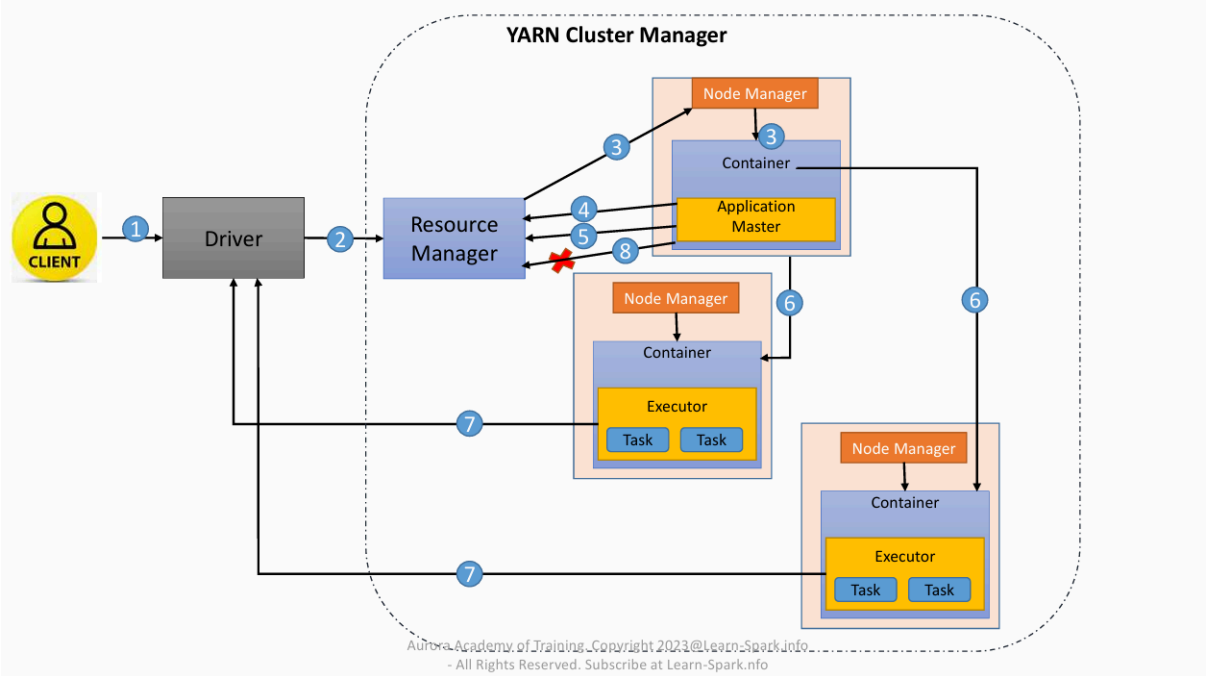
    - Once execution is completed, the final result is sent back to the driver.

Cluster Manager Types in Spark

Spark can work with different cluster managers:

1. **Standalone Mode** → Simple built-in cluster manager.
2. **Apache Mesos** → Can also manage Hadoop and other workloads.
3. **Hadoop YARN** → Used in Hadoop-based clusters.
4. **Kubernetes** → Manages containerized applications efficiently.

YARN as Spark Cluster Manager



YARN (Yet Another Resource Negotiator) is a **cluster manager** used in Hadoop 2.x. It helps in **resource allocation** (CPU, memory) and **job scheduling** when running Spark applications.

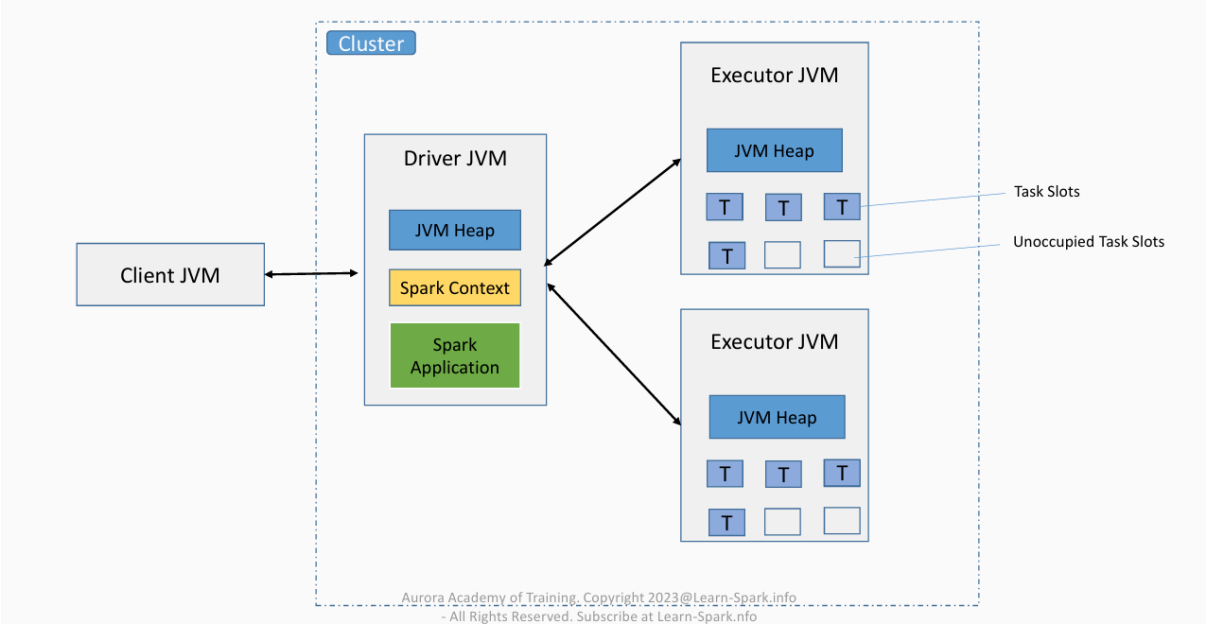
Main Components of YARN

1. **Resource Manager (RM) → "The Boss" (Runs on Master Node)**
    - Manages resources across the cluster.
    - Has **two parts**:
        - **Scheduler** → Assigns resources based on job needs.
        - **Application Manager** → Handles running **Application Masters** and restarts them if needed.
2. **Node Manager (NM) → "The Worker Manager" (Runs on Worker Nodes)**
    - Runs on **each worker node** in the cluster.
    - Launches and **monitors containers** (CPU, RAM, Memory) assigned by the Resource Manager.
    - Ensures tasks run properly on each node.
3. **Application Master (AM) → "The Job Coordinator"**
    - Assigned **per job** by the Resource Manager.
    - **Negotiates resources** from RM and works with NM to monitor and execute tasks.
4. **Containers → "The Workspaces"**
    - A container is a **bundle of resources** (CPU, RAM, etc.) on a node.
    - RM **allocates** and NM **monitors** them for job execution.

Step-by-Step Execution Flow in Spark on YARN

1. **User submits a Spark job** → The **Driver** starts and creates **SparkContext**.
2. **Driver contacts YARN (Resource Manager)** to request resources.
3. **Resource Manager assigns a Node Manager**, which then launches an **ApplicationMaster** for that job.
4. **ApplicationMaster registers with Resource Manager** and requests **more containers** to run tasks.
5. **Resource Manager assigns containers**, and **Node Managers launch executors** inside them.
6. **Executors start running tasks**, with the **Driver coordinating everything**.
7. **Once tasks are finished, the ApplicationMaster unregisters** from the Resource Manager.

JVM Processes



When we **submit a Spark job**, several **JVM (Java Virtual Machine) processes** are created across the cluster to handle the execution. Let's break it down step by step.

Step 1: JVM on the Local Server (Driver JVM)

- When a user submits a **Spark job**, a **JVM process is created** on the local server.
- This JVM is called the **Driver JVM**, and it is responsible for managing the job.

Inside the **Driver JVM**, you will find:

1. **JVM Heap** → This is a portion of memory allocated to the Driver.
2. **Spark Context** → The main entry point for Spark, created when a job is submitted.
3. **Spark Application Details** → Stores information about the running application.

Only one Driver is created per job in the cluster.

Step 2: JVMs on Executors

- Spark does not execute tasks on the Driver itself. Instead, it assigns work to **Executors**, which are worker nodes in the cluster.
- For each **Executor**, a separate **JVM process** is created.

For example:
If the cluster has **2 Executors**, then **2 Executor JVMs** will be created.
If the cluster has **5 Executors**, then **5 Executor JVMs** will be created.

Step 3: Inside the Executor JVM

Each **Executor JVM** contains:

1. **JVM Heap** → Memory allocated for the executor.
2. **Task Slots** → These are like small "workspaces" inside the executor where tasks run.
    - **Filled Boxes** = Task slots that are currently in use.
    - **Empty Boxes** = Task slots that are available for new tasks.

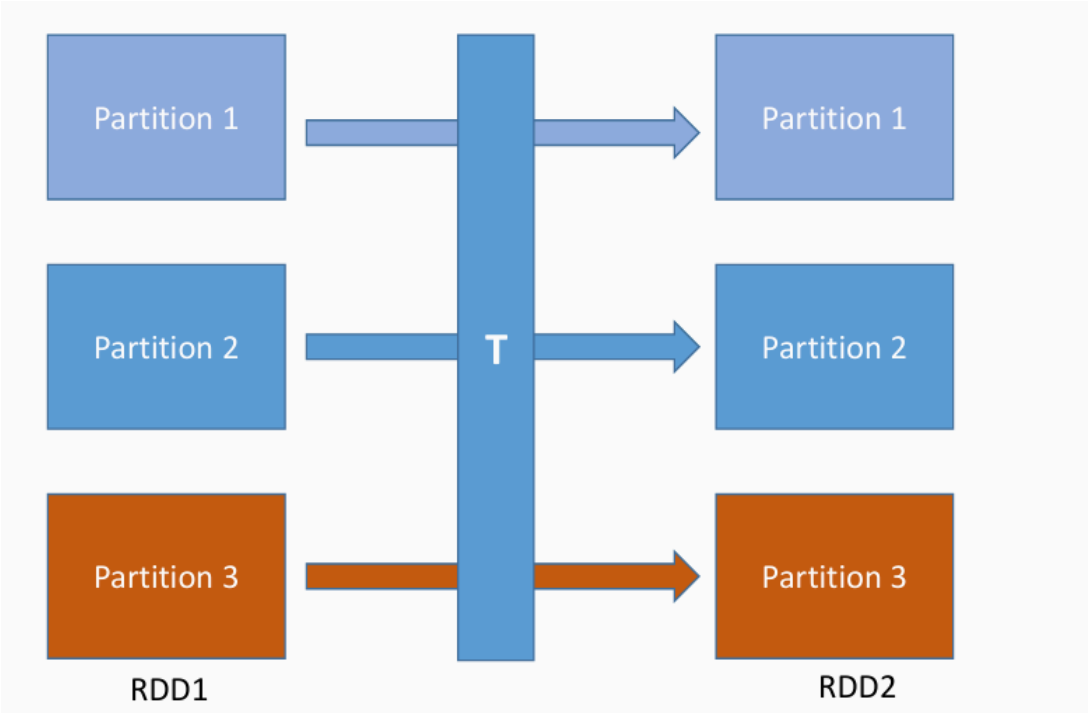
Terms used in Spark Execution Framework

- **Application** → Your **full Spark program**
- **Job** → A **big computation** triggered by an action
- **Stage** → **Mini steps** inside a job
- **Task** → The **smallest unit of work**
- **Application JAR** → Packaged **code file**
- **Driver** → **Controls execution** and assigns work
- **Cluster Manager** → **Allocates resources** to Spark
- **Deploy Mode** → Defines **where the driver runs**
- **Worker Node** → **Computers that do the work**
- **Executor** → **Runs tasks inside worker nodes**
- **Cache** → Stores **frequently used data** for speed

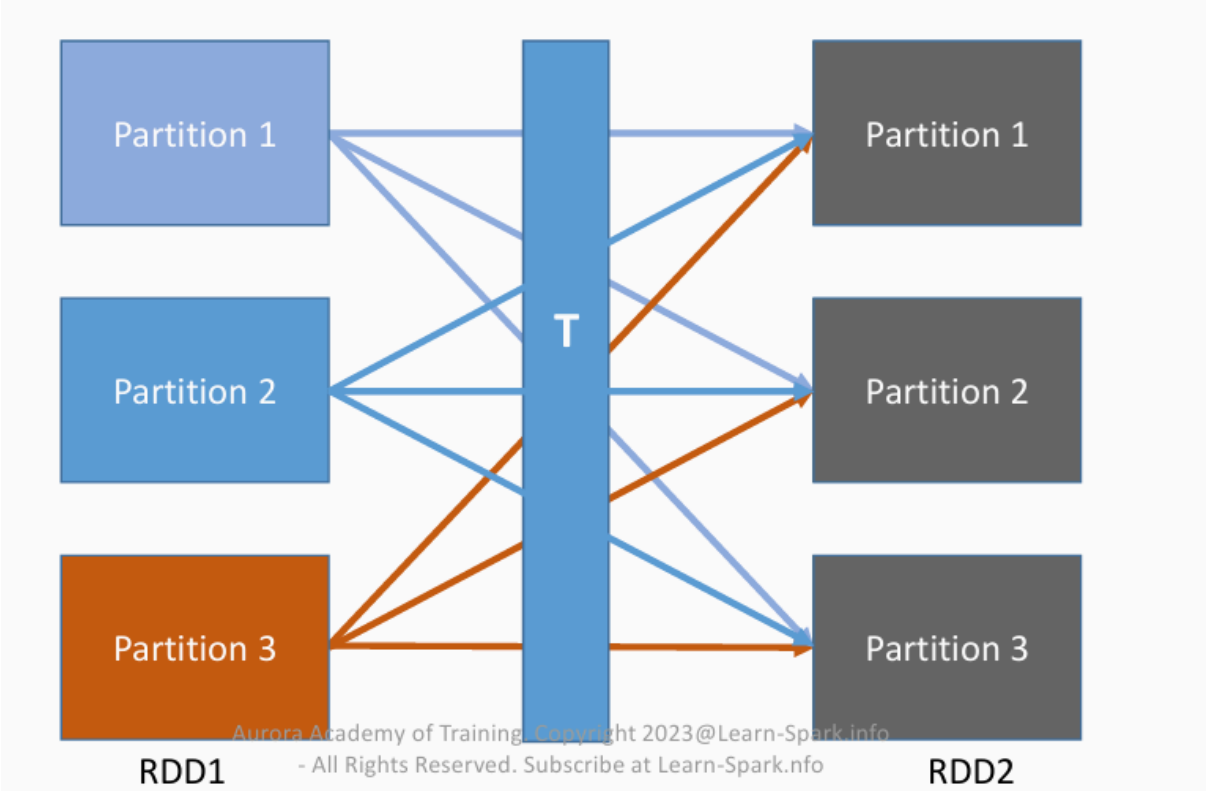
Narrow and Wide Transformations

Feature	Narrow Transformations	Wide Transformations
Speed	Fast	Slow
Data Shuffle	No	Yes
Stages	1 stage	Multiple Stages
Examples	map(), filter()	groupByKey(), join(), distinct()

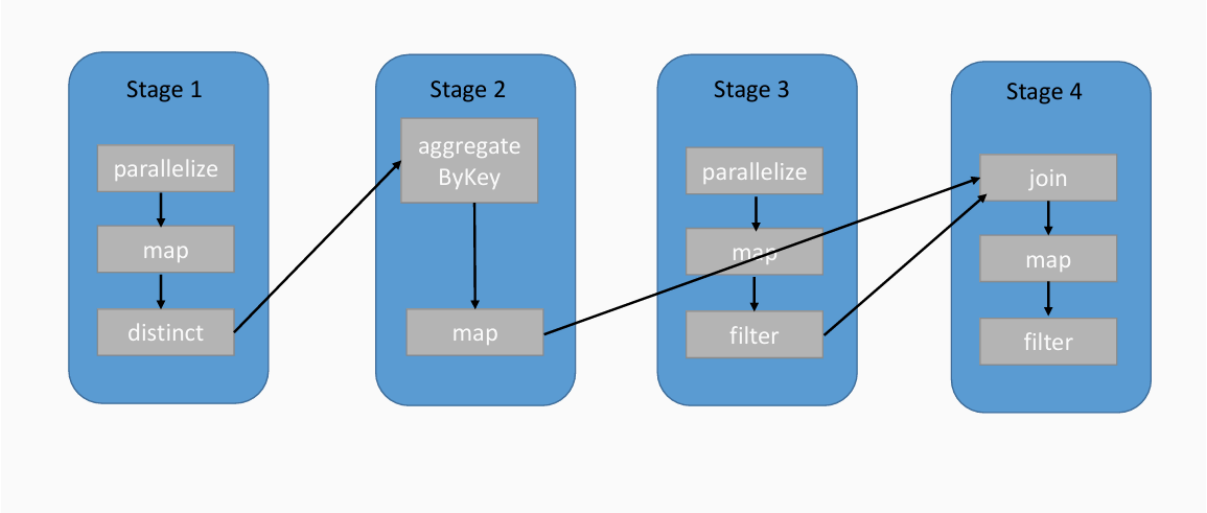
Narrow Transformations:



Wide Transformations:



DAG



A **DAG (Directed Acyclic Graph)** is a type of graph used by Spark to structure and optimize the execution of a job.

- **Directed** → The graph has a **one-way flow** (from start to end).
- **Acyclic** → No **loops** or going back to a previous step.
- **Graph** → Represents different **stages** and **tasks** as nodes connected by edges.

How DAG is Created and Used in Spark

Step 1: User Submits a Job

- A **client** (user) submits a Spark application.

Step 2: Driver Creates SparkContext

- The **Driver program** takes the application and initializes a **SparkContext** to manage the execution.

Step 3: Spark Identifies Transformations & Actions

- **Transformations** (e.g., `map()`, `filter()`) and **Actions** (e.g., `collect()`, `save()`) are detected.

Step 4: Logical Execution Plan (DAG) is Created

- Spark **arranges** all transformations into a **logical order**, forming the **DAG**.

Step 5: DAG is Sent to DAG Scheduler

- If an **Action** (like `collect()`) is found, the DAG is **submitted to the DAG Scheduler**.
- If no **Action** is found, execution **stops here**.

Step 6: DAG Scheduler Converts Logical Plan to Physical Plan

- The **DAG Scheduler** breaks the job into **Stages**.
- **Narrow transformations (map, filter)** are merged into **one stage**.
- A **new stage** is created when a **wide transformation (groupByKey, join, etc.)** is found.

Step 7: Tasks are Sent to Task Scheduler

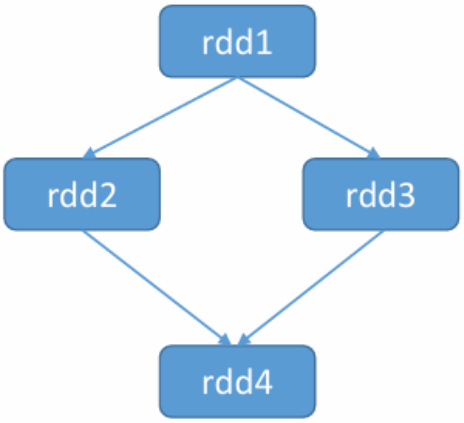
- The **DAG Scheduler** groups tasks and sends them to the **Task Scheduler**.
- The **Task Scheduler** submits them to the **Cluster Manager** (YARN, Mesos, Kubernetes, etc.) for execution.

Step 8: Execution and Completion

- The **Cluster Manager** assigns tasks to **executors** on worker nodes.
- The **executors run the tasks**, process the data, and return results.

RDD Lineage

RDD Lineage is the **history** or **relationship chain** of an RDD (Resilient Distributed Dataset) in Spark. It shows **how an RDD was created** by keeping track of its parent RDD(s).



RDD Lineage Graph

How Does RDD Lineage Work?

Every RDD **remembers** which RDD it came from and **what transformation** was applied to it.

Why is RDD Lineage Important?

- Helps Spark **recompute** lost RDDs if a failure happens.
- Supports **fault tolerance** (If data is lost, Spark can trace back and rebuild it).
- Improves **performance** by avoiding unnecessary recomputation.

Viewing RDD Lineage

- You can print the lineage using the `toDebugString()` API
    - Helps in **debugging** by showing the complete **RDD transformation history**.
    - Useful for **optimizing performance** by checking how Spark processes the data.
    - Shows **how RDDs are connected** (RDD lineage graph).

DAG Scheduler to Task Scheduler

Once the **DAG Scheduler** creates the **execution plan** (stages and tasks), it sends these tasks to the **Task Scheduler**, which assigns them to executors for execution.

Step-by-Step Explanation:

Step 1: DAG Scheduler breaks the job into stages

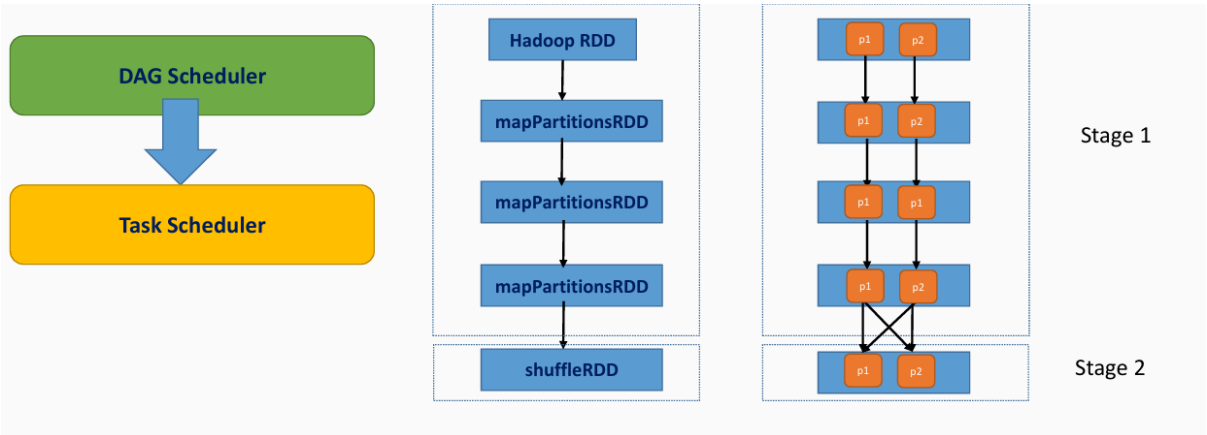
- Spark first creates a **DAG (Directed Acyclic Graph)** of tasks.
- Then, the **DAG Scheduler** divides the job into **multiple stages** based on transformations.
- Example: If the job has a `map()` and then a `reduceByKey()`, Spark will create **two stages** (one for `map()`, one for `reduceByKey()`).

Step 2: DAG Scheduler submits tasks to the Task Scheduler

- Each **stage** contains multiple **tasks**.
- The number of **tasks** depends on the number of **partitions** in the data.

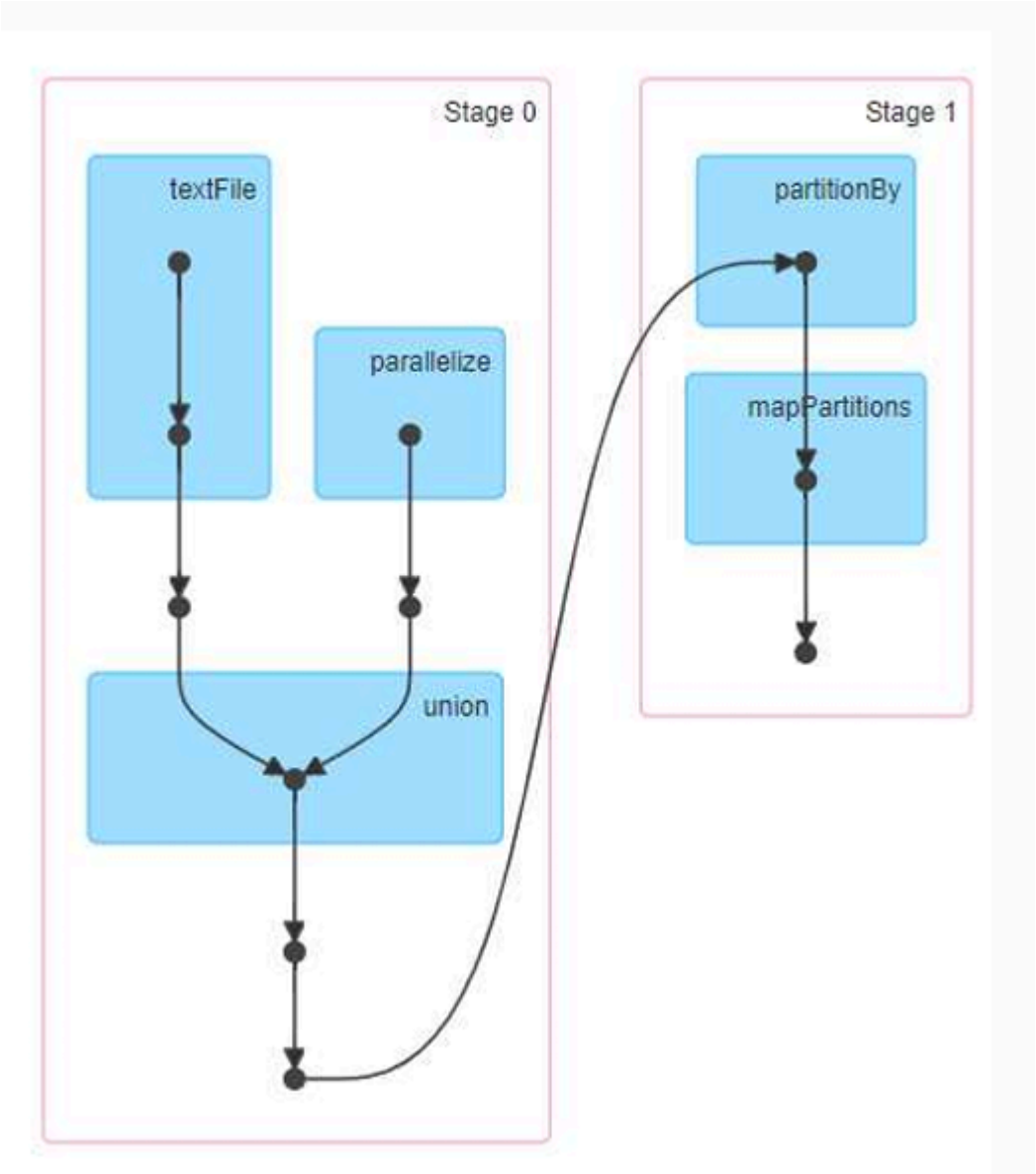
Step 3: Task Scheduler distributes tasks to executors

- The **Task Scheduler** takes the tasks and assigns them to **available worker nodes** (executors).
- Tasks are run **in parallel**, depending on how many **cores** are available.



Parallel Execution of Stages

- The stages which are not dependent on each other may be submitted to the cluster for execution in parallel.



Section-12 (RDD Persistence)

One of the most important features of Apache Spark is its ability to store data in memory for faster processing. This is called persisting (or caching).

What Happens When We Persist (Cache) an RDD?

- When we persist an **RDD** (Resilient Distributed Dataset), each node stores the parts of the data it processes **in memory** and reuses them for future operations.
- We can mark an RDD for storage using **persist()** or **cache()** methods.
- The first time the RDD is computed, it is stored in memory for reuse.
- Spark's cache is **fault-tolerant**, meaning if a part of the data is lost, Spark will **recompute it** using the original transformations.
- Different **storage levels** decide where and how the data is stored.

Storage Levels in Spark

Spark provides multiple storage options for persisting data:

1. **Memory (RAM)** – Data is stored in RAM for the fastest access.
2. **Disk** – Data is stored on disk when memory is not enough.
3. **Off-Heap Memory** – Memory outside the **JVM** (Java Virtual Machine) but still accessible for Spark.
4. **Serialized vs. Deserialized** – Data can be stored in a compressed format (**serialized**) or in its original form (**deserialized**) for faster access.

When to Use Different Storage Levels?

Scenario	Best Storage Level	Reason
Data fits in memory	`MEMORY_ONLY`	Fastest processing
Large dataset, memory is not enough	`MEMORY_AND_DISK`	Uses disk when memory is full
Need faster recovery from failures	`MEMORY_AND_DISK_2`	Data is replicated
Need to optimize CPU usage	`SERIALIZED` storage	Saves memory, but increases CPU load

Trade-offs

- More memory usage = Faster processing
- Less memory usage (disk storage) = Slower processing
- Replicated storage = More fault tolerance but uses more memory

Section-13 (Spark Shared Variables)

- Shared variables are the variables that are required to be used by functions and methods in parallel.
- Shared variables can be used in parallel operations.
- Spark provides two types of shared variables:
    - Broadcast
    - Accumulator

Broadcast Variables:

- In **Spark**, when you need to share a large dataset (like a large array or a big file) **with all the worker nodes**, you can use a **Broadcast Variable**.
- **Instead of sending the data again and again with every task, Spark will send it only once per node** and keep it cached (saved) on that node.
- This saves **time, memory, and network bandwidth**.

Key Points About Broadcast Variables:

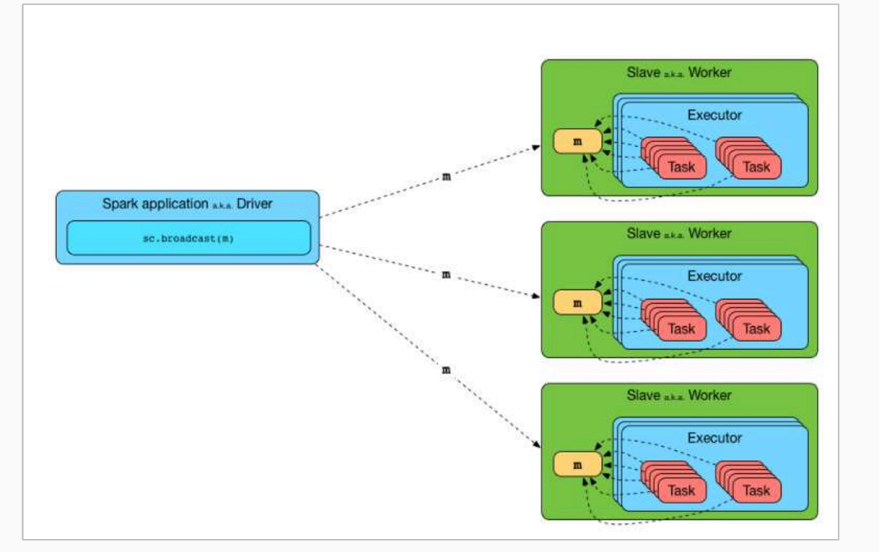
1. **Immutable (cannot be changed)** → Once you send the data to the nodes, you **cannot modify it**.
2. **Cached on each node** → Spark saves the data **once per node** instead of sending it again and again.
3. **Memory Efficient** → It reduces **network traffic** by distributing the data only once per node instead of per task.

When Should You Use Broadcast Variables?

- Suppose you have a **huge dataset (like an array of 1 million records)** that all your tasks need to use.
- If you do **not** use a broadcast variable, **Spark will send the data again and again** with every task (100 times in your example).
- But **if you use a broadcast variable**, the data will be sent **only once per node**, making it faster and more efficient.

What Should You Avoid?

- **Do not modify the value** of the broadcast variable once it is sent to the nodes.
- If you change the value, **some nodes will have old data and some will have new data**, causing **wrong results**.
- So always keep **broadcast variables read-only**.




```
>>> days={"sun":"Sunday", "mon":"Monday", "tue":"Tuesday"}
>>> bcDays=spark.sparkContext.broadcast (days)
>>> bcDays
<pyspark.broadcast.Broadcast object at 0x00000258FB140C70>
>>> bcDays.value
{'sun': 'Sunday', 'mon': 'Monday', 'tue': 'Tuesday'}
>>> bcDays.value['sun']
'Sunday'
```

Accumulator Variables:

- It is a shared variable to perform sum and counter operations.
- These variables are shared by all executors to update and add information through associative or commutative operations.
    - Commutative -> f(x, y) = f(y, x)
      Ex : sum(5 ,7) = sum(7, 5) ✅
    - Associative -> f(f(x, y), z) = f(f(x, z), y) = f(f(y, z), x)
      Ex : sum(multiply(5,6),7) = sum(multiply(6,7),5) ❌
      sum(sum(5,6),7) = sum(sum(6,7),5) ✅

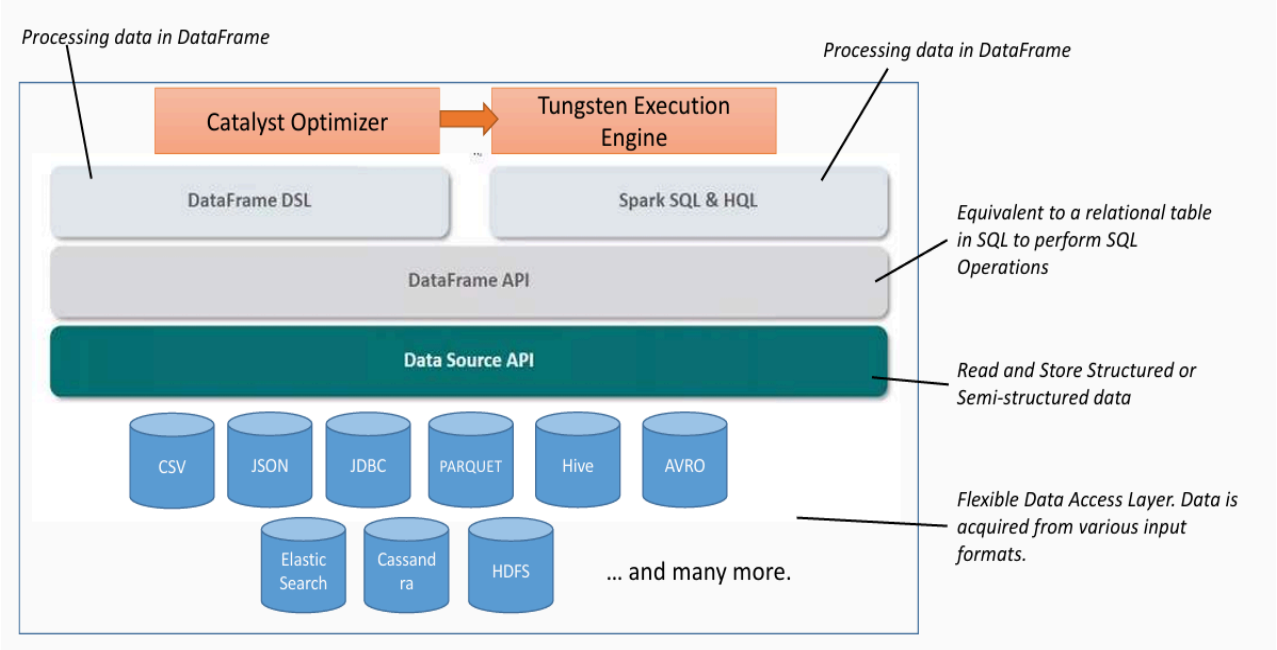
```
>>> counter=0
>>> def f1(x):
...     global counter
...     counter+=1
...
>>> f1(10)
>>> counter
1
>>> f1(10)
>>> counter
2
```

```
>>> rdd=spark.sparkContext.parallelize([1,2,3])
>>> rdd.foreach(f1)
>>> counter
2
```

The counter Variable will not added or changed, because when spark ships this code to every executor the variables become local to that executor. So the variable is updated for that executor but do not send it back to the driver. To avoid this problems, we need ac accumulator. All the updates to accumulator variable in every executor is send it back to the driver.

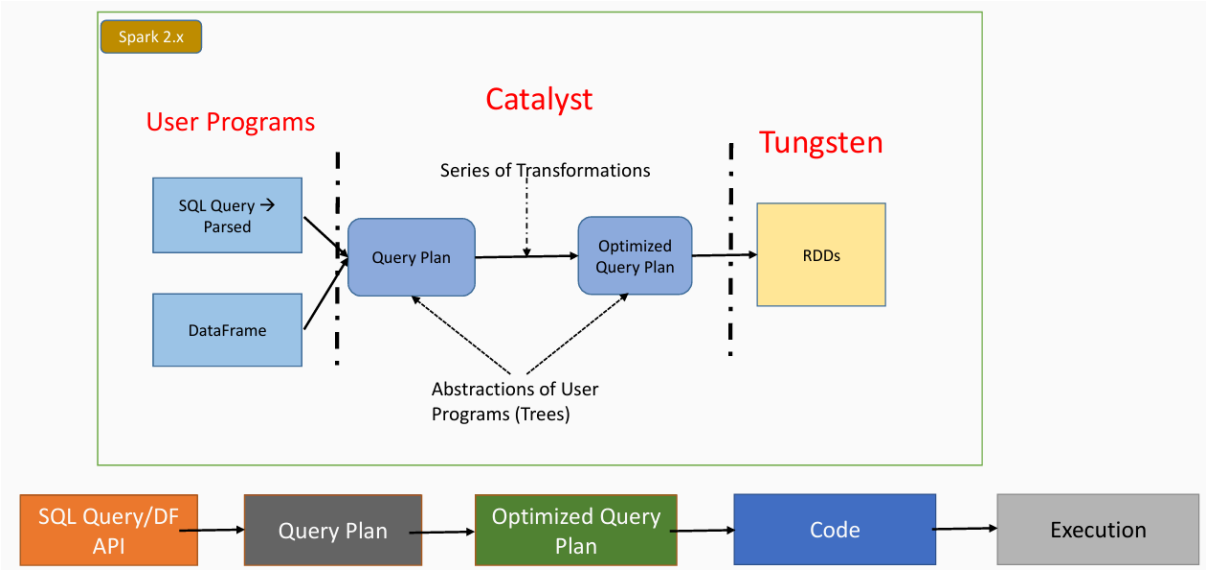
```
>>> counter1 = spark.sparkContext.accumulator(0)
>>> counter1
Accumulator<id=0, value=0>
>>> def f2(x):
...     global counter1
...     counter1.add(1)
...
>>> rdd.foreach(f2)
>>> counter1
Accumulator<id=0, value=3>
>>> counter1.value
3
>>> accum=sc.accumulator(0)
>>> accum.value
0
>>> rdd=spark.sparkContext.parallelize([1,2,3,4,5])
>>> rdd.foreach(lambda x: accum.add(x))
>>> accum.value
15
```

Section-14 (Spark SQL Architecture)



Layer	What it does
Flexible Data Access Layer	Reads data from CSV, JSON, Databases, etc.
Data Source API Layer	Converts data into DataFrame.
DataFrame API	Allows you to manipulate data using Python code.
DataFrame DSL / Spark SQL	Allows you to use SQL queries.
Catalyst Optimizer	Optimizes your query automatically.
Tungsten Engine	Executes the optimized query on multiple nodes.

How Catalyst and Tungsten Works



What Happens When You Run a SQL Query in Spark?

Imagine a **SQL Query** like this:

```
SELECT name, age FROM people WHERE age > 30;
```

What happens in the background in Spark?

- **SQL Query → Becomes a Query Plan → Gets Optimized → Runs Fast Using Tungsten Engine.**
- **But how?**
- Let's break it down in a very simple way.

Step 1: User Program (Write SQL Query)

What is a User Program?

- This is simply the **SQL Query** or **DataFrame API code** that you write in Spark.
- Example:

```
SELECT name, age FROM people WHERE age > 30;
```

OR

```
df.filter(df.age > 30).select(df.name, df.age).show()
```

So in simple terms:

- **User Program = SQL Query or Python Code**

Step 2: Catalyst Optimizer Converts Query into Query Plan (Tree)

- When we write the SQL query, **Spark doesn't run it directly.**
- **Instead, Spark first converts the SQL query into a Query Plan.**
- **What is a Query Plan?**
    - A **Query Plan** is like a **recipe** that Spark creates based on your SQL query.
    - It tells **what steps** Spark should follow to **get your result.**
    - This **Query Plan** is like a **Tree Structure**.

Example:

- Suppose the query was:

```
SELECT name, age FROM people WHERE age > 30;
```

- Spark will create a Query Plan like:

```
Query Plan (Tree Structure)
     Filter (age > 30)
         |
       Select (name, age)
         |
       Read from CSV File
```

- **This Query Plan is the first step:** It contains the steps to read the data, filter it, and show results.

But, this plan is not optimized yet.

Step 3: Catalyst Optimizer (Optimize the Query Plan)

- It **looks at the query plan** and tries to **optimize it** to make it run faster and cheaper.
- It applies a **series of transformations** to modify the plan.

Example (How It Optimizes)

- Imagine the original query was:

```
SELECT name, age FROM people WHERE age > 30;
```

- The initial query plan was:

```
Filter (age > 30)
   |
Select (name, age)
```

```
   |
Read from CSV File
```

- Catalyst Optimizer will do smart things like:
    - **Push Filter Down** → Instead of reading the entire file, it will **filter data first**, reducing the data size.
    - **Column Pruning** → Instead of reading **all columns**, it will only read **name** and **age**.
    - **Combine Filters** → If there were multiple filters, it will combine them to reduce computation.
- **After Optimization:**

```
Read only 'name' and 'age' from CSV
   |
Filter (age > 30)
   |
Show result
```

- Now the query will run **10x faster** because Spark will:
    - Read only **2 columns** instead of 10 columns.
    - Filter the data first, reducing computation.

Step 4: Optimized Query Plan is Sent to Tungsten Execution Engine

- Tungsten is like a **powerful engine** that takes your query and converts it into **optimized machine code (Java bytecode)**.
- Then it executes this code **on a cluster (multiple nodes)**.

Step 5: Tungsten Execution Engine Does Magic

- Tungsten Engine does **3 amazing things**:

1. In-Memory Processing (Super Fast)

- It keeps your data **in memory (RAM)** instead of writing it to disk.
- This makes the processing **100x faster**.

2. Code Generation (Super Efficient)

- Tungsten will automatically generate **Java bytecode** to process your data.
- Instead of processing each row **one-by-one**, it will process **multiple rows in one go**.

Example:

- Without Tungsten → Process 1 row → Move to next → Slow
- With Tungsten → Process 1000 rows in one go → Super Fast

3. Binary Processing (Save Memory)

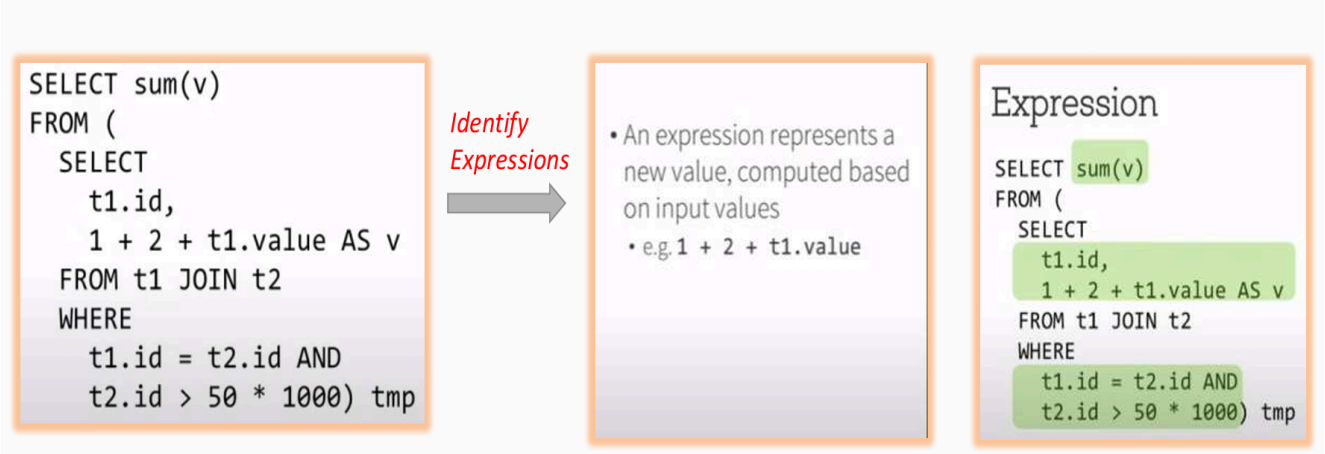
- Instead of treating data as **text**, Tungsten converts it to **binary (0s and 1s)**.
- Binary processing is extremely fast and uses very little memory.

This is why **Spark is super fast** compared to other tools.

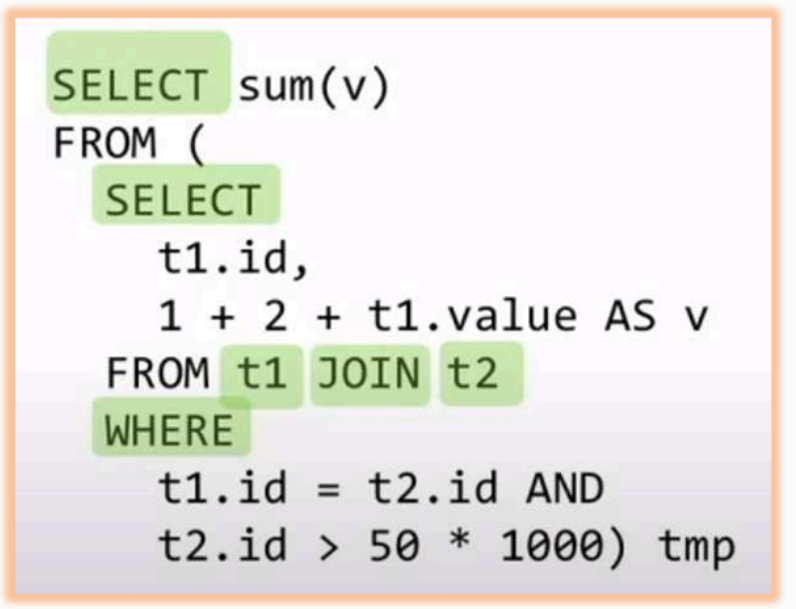
Step 6: Results Are Shown

- Finally, Tungsten Engine will **execute the query in a distributed cluster** and show the result.

Trees: Abstraction of Users Program

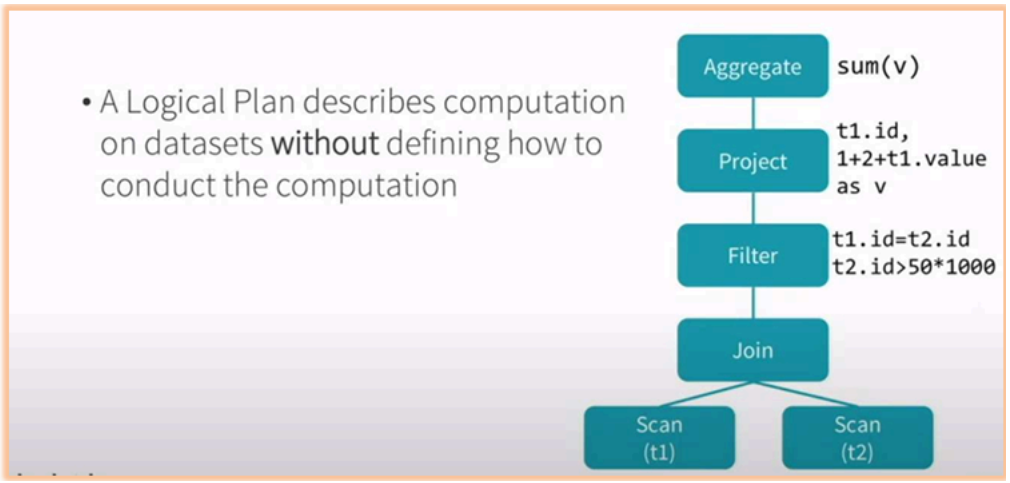


Query Plan: describes data operation like aggregates, joins, filters etc. and these operations essentially generate a new dataset based on a input dataset.

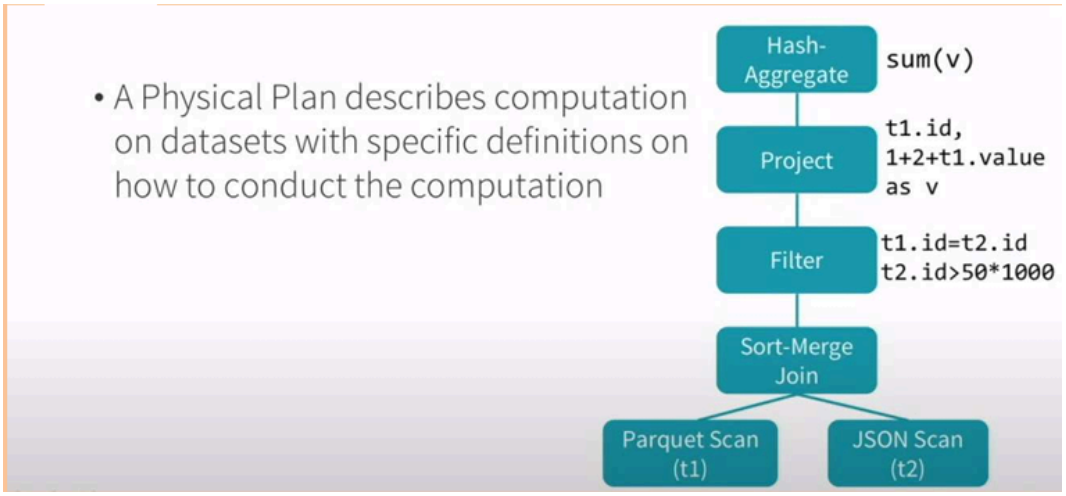


Two types:

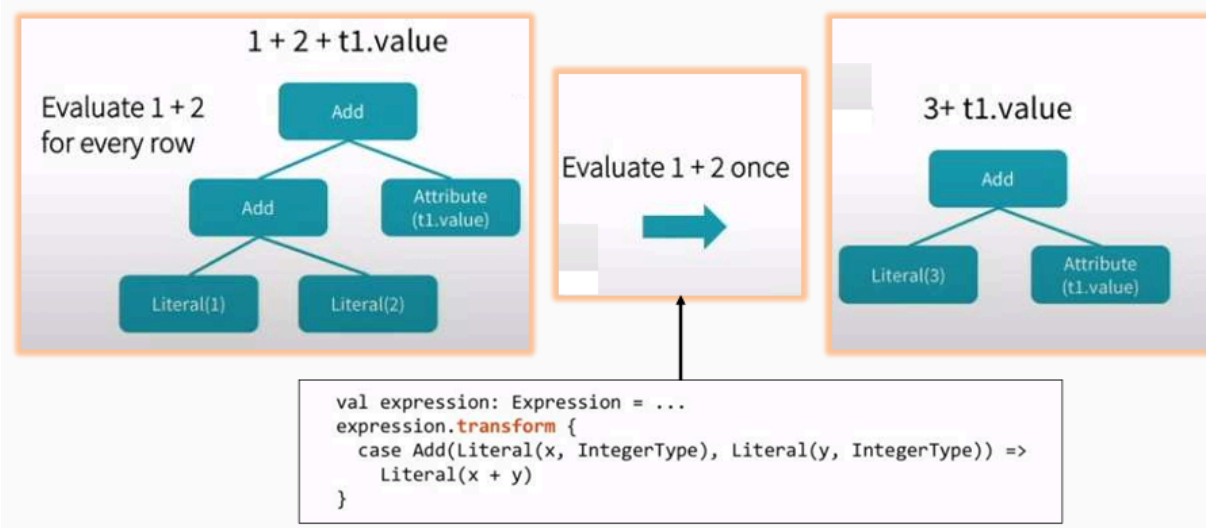
1. Logical Query Plan-



2. Physical Query Plan-



Transform:



A **transform function** is simply a function that applies a **rule** to make your query more efficient.

Example:
In the query:

```
SELECT 1+2+t1.value
```

This means **for every row**, Spark will calculate **1+2** again and again (maybe billions of times).
But logically, **1+2** is always **3**

So Spark says:

- Why calculate **1+2** a billion times?
- Why not calculate it once during compilation and use the result?

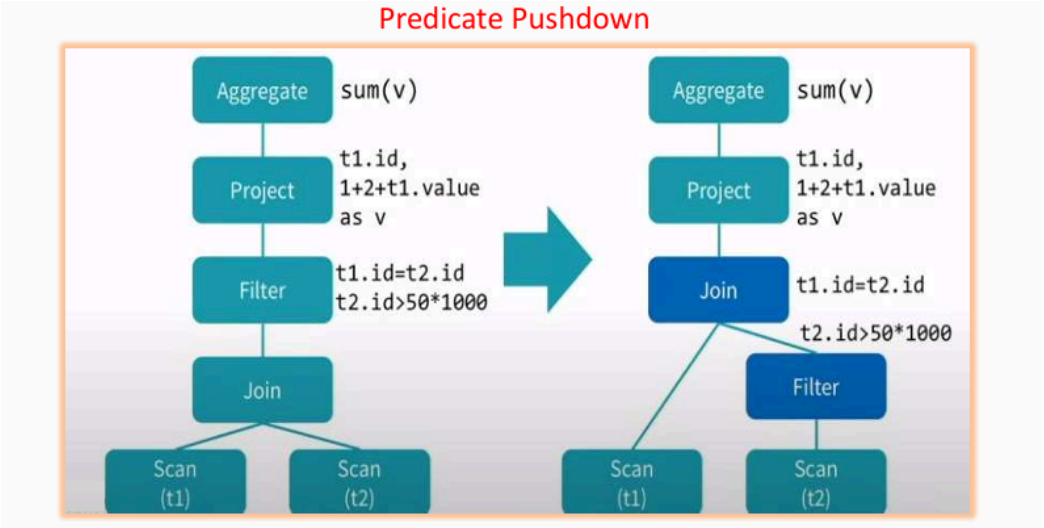
This is where a **Transform Function** comes into play.

- It will **evaluate 1+2 = 3 only once** during compile time.
- So your final query becomes:

```
SELECT 3+t1.value
```

This is a huge performance improvement.

There are 3 optimizations:



What is happening here?

- First, you are **joining** two tables.
- After joining, you are applying a **filter** (`T2.id > 50000`).

But logically:

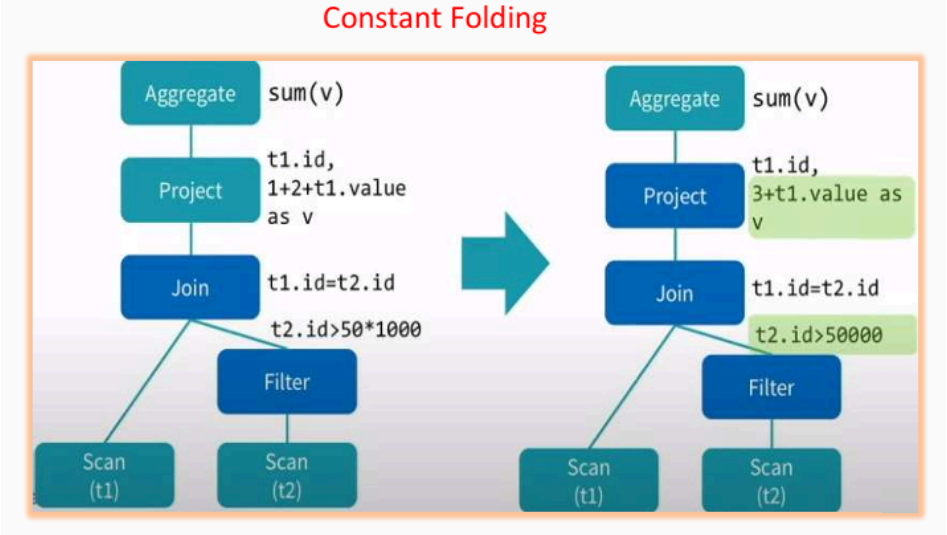
- Why should Spark **join all rows** and then **filter**?

- Instead, **why not filter the data first and then join?**

Spark realizes this and says:

- It will **push the filter to the data source level**.
- This means before joining, it will only bring data where `T2.id > 50000`.
- This will reduce the size of data and make the join faster.

This optimization is called **Predicate Pushdown**.



Constant Folding is a specific transform function that deals with **fixed values (constants)**.
Example:

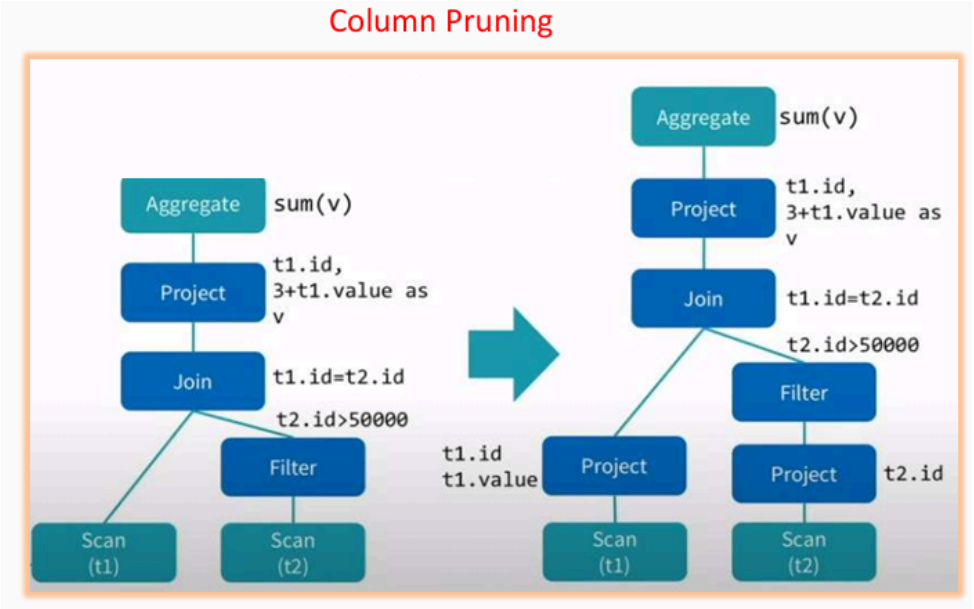
```
SELECT 1+2+t1.value
```

- Spark will **calculate 1+2 = 3** at compile time.
- So the query becomes:

```
SELECT 3+t1.value
```

This saves a lot of time because Spark doesn't calculate **1+2** for each row.

This optimization is called Constant Folding.



Column Pruning means:

- **Only bring the columns you need, not everything**.

Example:

```
SELECT t1.value, t2.id

FROM T1

JOIN T2 ON T1.id = T2.id;
```

- **T1** has 100 columns.
- **T2** has 100 columns.
- But you only need **T1.value** and **T2.id**.
- By default, Spark **loads all columns** from T1 and T2.
- This wastes a lot of time and memory.

Spark uses a **transform function** called **Column Pruning**. It will only **load the columns you need** instead of loading everything.

- So Spark will only fetch:
  - **T1.value**
  - **T2.id**
- This significantly improves performance.

What Is The Volcano Iterator Model in Spark 1.6?

- In **Spark 1.6** (before Spark 2.0), when a query is written like this:

Example Query:

```
SELECT COUNT(*)

FROM orders

WHERE order_cust_id = 1000;
```

Spark had to process your query step-by-step like this:

1. **Scan** → Read data from the table (`orders`).
2. **Filter** → Keep only rows where `order_cust_id = 1000`.
3. **Project** → Select required columns (if needed).
4. **Aggregate** → Count the number of remaining rows.

What Is An Operator in Spark?

- In Spark, every step of the query (like Scan, Filter, Project, Aggregate) is called an **Operator**.
- Each **Operator** does only one job:
    - **Scan Operator** → Read data from the table.
    - **Filter Operator** → Filter rows based on condition.
    - **Project Operator** → Select required columns.
    - **Aggregate Operator** → Count the rows.

How Did Operators Communicate With Each Other in Spark 1.6?

- **This is where the problem starts.**
- Spark used something called:

Iterator Interface (Volcano Model)

Imagine **4 workers** in a factory:

1. **Worker 1 (Scan)** → Reads all products from the shelf.
2. **Worker 2 (Filter)** → Only keeps products with **price > 1000**.
3. **Worker 3 (Project)** → Selects only the **product name**.
4. **Worker 4 (Aggregate)** → Counts how many products are left.

Now the rule is:

- Worker 1 **cannot directly talk** to Worker 3 or 4.
- Worker 1 can only **give the product to Worker 2**.
- Worker 2 can only **give the product to Worker 3**.
- Worker 3 can only **give the product to Worker 4**.

How do they pass products to each other?

- By calling a function called **next()**.
- **next()** → Means: "Give me the next record."

What Is Happening in Spark 1.6 (Step-by-Step)?

Step 1: Scan Operator

- It **reads the data** and sends it **to the Filter Operator**.

```
for each row in orders:
    pass the row to the next operator (Filter)
```

Step 2: Filter Operator

- The **Filter Operator** says:

```
for each row:
    if order_cust_id == 1000:
        pass the row to the next operator (Project)
```

The **Filter Operator** has no idea:

- **Where the data came from** .
- **Who will use the data next**.

It only **calls the next() function** to pass data.

Step 3: Project Operator

The **Project Operator** says:

```
for each row:
  select only the columns we need
  pass the row to the next operator (Aggregate)
```

- Again, it doesn't know where the data came from.

Step 4: Aggregate Operator

- The **Aggregate Operator** finally says:

```
for each row:
   count++
```

What Is The Problem in Volcano Model?

- The **biggest problem** was the use of **next() function**.
- Every time data passed from:
    - **Scan → Filter → Project → Aggregate**,
    - Spark had to make a **function call**.

Why Is Function Call Bad?

Function call means:

- The operator **suspends its work**.
- **Saves data in memory (RAM)**.
- Calls the next operator using **next()** function.
- **Then comes back** and processes the next row.

This is **VERY SLOW**.

Why Was This Model Slow?

Problem 1: Too Many Function Calls

Every time Spark read one record, it called:

- **Scan → next() → Filter → next() → Project → next() → Aggregate**.
- If you had **millions of records**, you had **millions of function calls**.
- This was extremely slow.

Problem 2: Extensive Memory Access

- Every time Spark passed data to another operator, it wrote the data in **memory (RAM)**.
- This looked like:

```
Read → Write to memory → Read → Write to memory → Read → Write to memory
```

- This **memory read/write** made Spark very slow.

Problem 3: No Modern CPU Features

Modern CPUs have fast features like:

- **Pipelining** → Process multiple data in parallel.
- **SIMD** → Process multiple rows in one go.
- **Prefetching** → Load data in advance.

But because of **function calls**, Spark could not use these features.

Why Did Spark Move To Tungsten Engine (Spark 2.0)?

Spark engineers realized:

- **Why are we making so many function calls?**
- **Why not combine all steps into one function?**

So in **Spark 2.0**, they introduced:

Whole Stage Code Generation (WSCG)

This means:

- Combine **Scan + Filter + Project + Aggregate** into **one big function**.
- No more function calls.
- No more memory writes.
- Full CPU power usage.

Feature	Volcano Model	Tungsten Engine
Function Calls	Millions of calls	No function calls
Memory Access	Write to Memory	Use CPU Registers
Speed	Very Slow	Super Fast
CPU Usage	No modern CPU usage	Full CPU Optimization

Benchmark

```
>>> spark.conf.get("spark.sql.codegen.wholeStage")
'true'
>>> spark.conf.get("spark.sql.codegen.wholeStage",False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\pyspark\sql\conf.py", line 57, in get
    self._checkType(default, "default")
  File "C:\PySpark_Installed\Spark\spark-3.5.5-bin-hadoop3\python\pyspark\sql\conf.py", line 68, in _checkType
    raise TypeError(
TypeError: expected default 'False' to be a string (was 'bool')
>>> spark.conf.set("spark.sql.codegen.wholeStage",False)
>>> spark.conf.get("spark.sql.codegen.wholeStage")
'false'
>>> from time import time
>>> time()
1741602980.8941953
>>> from pyspark.sql.functions import sum
>>> def benchmark(version):
...     start=time()
...     spark.range(1000 * 1000 * 1000).select(sum("id")).show()
...     end=time()
...     elapsed=end-start
...     print(elapsed)
...
>>> benchmark("spark 1.6")
```

```
>>> benchmark("spark 1.6")
+------------------+
|           sum(id)|
+------------------+
|499999999500000000|
+------------------+

34.40936326980591
>>> spark.conf.set("spark.sql.codegen.wholeStage",True)
>>> spark.conf.get("spark.sql.codegen.wholeStage")
'true'
>>> def benchmark(version):
...     start=time()
...     spark.range(1000 * 1000 * 1000).select(sum("id")).show()
...     end=time()
...     elapsed=end-start
...     print(elapsed)
...
>>> benchmark("Spark 2.0")
+------------------+
|           sum(id)|
+------------------+
|499999999500000000|
+------------------+

1.9180397987365723
```

What Is `explain()` In PySpark?

- The `explain()` function in PySpark is used to **show how Spark is processing your query internally**.
- It helps you to understand **how Spark optimizes your query** behind the scenes.
- It is mostly used for **debugging and performance tuning**.

What Does `explain(extended=False)` Mean?

The `explain()` function has an **optional parameter** called **`extended`**.

- **extended = False (default)** → Shows **only the Physical Plan (actual execution)**.
- **extended = True** → Shows **all four plans (from start to end)**.

What Are These Four Plans?

When you run a query in Spark, Spark **does not execute it directly**.
Instead, it converts your query step-by-step into a **more optimized version**.

Here's what happens:

Plan Name	What It Means
Parsed Logical Plan	Spark reads your query and checks if the query is valid (like SQL syntax).
Analyzed Logical Plan	Spark verifies if the columns, tables, etc., exist or not.
Optimized Logical Plan	Spark tries to **optimize** your query to make it faster.
Physical Plan	Spark finally converts your query into **execution steps**.

```
>>> spark.conf.get("spark.sql.codegen.wholeStage")
'true'
>>> spark.range(1000).filter("id > 100").selectExpr("sum(id)")
DataFrame[sum(id): bigint]
>>> spark.range(1000).filter("id > 100").selectExpr("sum(id)").show()
+-------+
|sum(id)|
+-------+
| 494450|
+-------+

>>> spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[sum(id#48L)])
   +- Exchange SinglePartition, ENSURE_REQUIREMENTS, [plan_id=95]
      +- HashAggregate(keys=[], functions=[partial_sum(id#48L)])
         +- Filter (id#48L > 100)
            +- Range (0, 1000, step=1, splits=8)
```

```
>>> spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain(extended=True)
== Parsed Logical Plan ==
'Project [unresolvedalias('sum('id), Some(org.apache.spark.sql.Column$$Lambda$1506/0x000001368fd5e040@6f63acdf))]
+- Filter (id#57L > cast(100 as bigint))
   +- Range (0, 1000, step=1, splits=Some(8))

== Analyzed Logical Plan ==
sum(id): bigint
Aggregate [sum(id#57L) AS sum(id)#62L]
+- Filter (id#57L > cast(100 as bigint))
   +- Range (0, 1000, step=1, splits=Some(8))

== Optimized Logical Plan ==
Aggregate [sum(id#57L) AS sum(id)#62L]
+- Filter (id#57L > 100)
   +- Range (0, 1000, step=1, splits=Some(8))

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[sum(id#57L)], output=[sum(id)#62L])
   +- Exchange SinglePartition, ENSURE_REQUIREMENTS, [plan_id=112]
      +- HashAggregate(keys=[], functions=[partial_sum(id#57L)], output=[sum#65L])
         +- Filter (id#57L > 100)
            +- Range (0, 1000, step=1, splits=8)
```

Section-15 (SparkSession Features)

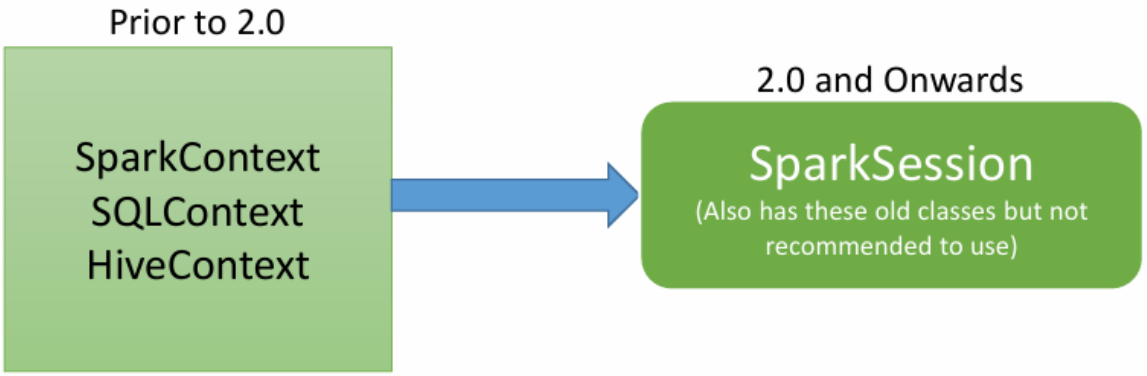
In **Spark (a big data processing tool)**, if you want to work with data, you need a starting point to interact with Spark. This starting point is called an **Entry Point**.

Before Spark 2.0

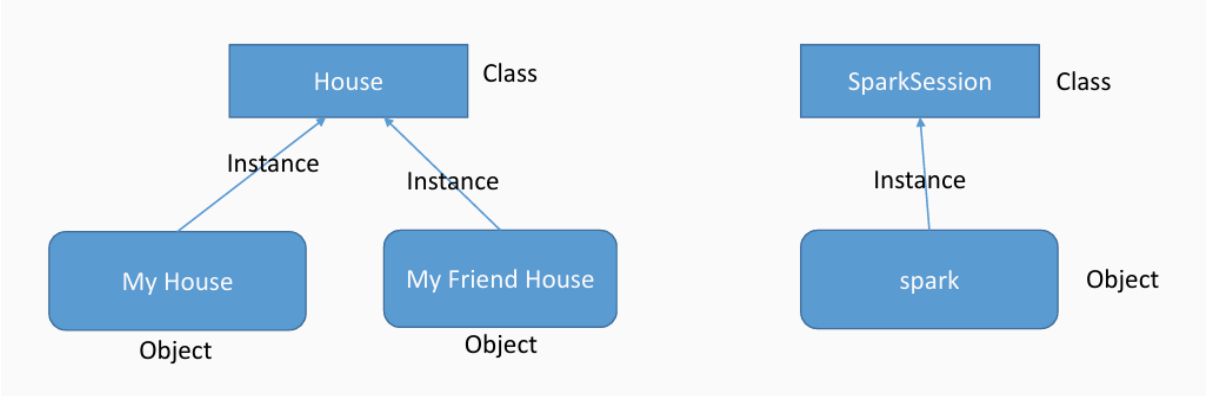
- **SparkContext** was the entry point to work with **RDD (Resilient Distributed Dataset)**, **Accumulators**, and **Broadcast Variables**.
- If you wanted to work with structured data (like tables), you needed **SQLContext**.
- If you wanted to work with Hive (a data warehouse system), you needed **HiveContext**.
- This means you had to create **different objects** for different tasks, which was a bit confusing.

After Spark 2.0

- **SparkSession** was introduced to simplify everything.
- Now, you don't need **SparkContext**, **SQLContext**, or **HiveContext** separately.
- **SparkSession** can do everything that these three used to do.
- It combines all functionalities (working with RDD, DataFrame, SQL, Hive, etc.) into **one single object** called **SparkSession**.



We can create a class to create one or more objects.



Spark Session Commonly used Functions:

- **version:** Returns Spark version where your application is running.

```
>>> spark.sparkContext.version
'3.5.5'
```

- **range(start, end=None, step=1, numPartitions=None):** This creates a DataFrame with a range of values.

```
>>> df=spark.range(1,10,2)
>>> type(df)
<class 'pyspark.sql.dataframe.DataFrame'>
>>> df.show()
+---+
| id|
+---+
|  1|
|  3|
|  5|
|  7|
|  9|
+---+
```

- **createDataFrame() :** This creates a DataFrame from a collection(list, dict), RDD or Python Pandas.

```
>>> lst=[('Robert',35),('James',25)]
>>> df=spark.createDataFrame(data=lst)
>>> df.show()
+------+---+
|    _1| _2|
+------+---+
|Robert| 35|
| James| 25|
+------+---+
```

- **sql() :** Returns a dataframe representing the result of a given query.
    - TempView: It is valid for only that particular session.

```
>>> lst1=[('Robert',35),('James',25)]
>>> df_emp=spark.createDataFrame(data=lst1,schema=['EmpName','Age'])
>>> lst2=[('Robert',101),('James',102)]
>>> lst2=[('Robert',101),('James',102)]
>>> df_dept=spark.createDataFrame(data=lst2,schema=['EmpName','DeptNo'])
>>> df_emp.createOrReplaceTempView("emp")
>>> spark.sql(""" select * from emp where empname='Robert' """).show()
+-------+---+
|EmpName|Age|
+-------+---+
| Robert| 35|
+-------+---+
```

- **table():** Returns the specified table as dataframe.

```
>>> spark.sql(""" select * from emp """).show()
+-------+---+
|EmpName|Age|
+-------+---+
| Robert| 35|
|  James| 25|
+-------+---+

>>> df=spark.table("emp")
>>> df.show()
+-------+---+
|EmpName|Age|
+-------+---+
| Robert| 35|
|  James| 25|
+-------+---+
```

- **sparkContext:** Returns sparkContext
- **conf():** Runtime configuration (get and set).
  Ex- spark.conf.set('spark.sql.shuffle.partitions',300) # configures the number of partitions that are used when shuffling data for joins or aggregations

```
>>> spark.conf.get('spark.sql.shuffle.partitions')
'200'
```

- **stop() :** Stop the underlying SparkContext.

Section-16 (DataFrame Fundamentals)

DataFrame is a dataset organized into named columns/ rows.

DataFrame Sources:

- Structured Data Files (CSV, JSON, AVRO, PARQUET etc.)
- Hive
- Cassandra
- Python Data frame
- RDBMS Databases
- RDDs

DataFrame Features:

1. DataFrame is distributed.
    a. Like RDD, DataFrame is also distributed.
    b. Supports HA (High availability) and FT (Fault Tolerant).

```
>>> lst=[("Robert",31),("Alicia",25),("Deja",19),("Manoj",31)]
>>> df=spark.createDataFrame(data=lst, schema=["Name","Age"])
>>> df.show()
+------+---+
|  Name|Age|
+------+---+
|Robert| 31|
|Alicia| 25|
|  Deja| 19|
| Manoj| 31|
+------+---+

>>> df.rdd.getNumPartitions()
8
>>> df.rdd.glom().collect()
[[], [Row(Name='Robert', Age=31)], [], [Row(Name='Alicia', Age=25)], [], [Row(Name='Deja', Age=19)], [], [Row(Name='Mano
j', Age=31)]]
```

2. Each transformation is a lazy operation. Evaluation is not started until an action s triggered.
3. Immutability: DataFrames are considered to be immutable storage.
4. Used across the Spark Ecosystem: DataFrame is a unified API across all libraries in Spark.
5. Polyglot: Supports multiple languages - Scala, Python, Java, R
6. Works on huge collection of dataset, feasible to work with a wide file.
7. Supports both structured and semi-structured data (JSON, XML etc).

DataFrame Organization of Data

A DataFrame has 3 levels to organize and process its data: Schema, Storage and API.

- Schema:
    - DataFrame is implemented as a dataset of rows.
    - Each column is named and typed.
- Storage:
    - Storage is distributed and data is stored in partitions.
    - Storage in Memory and Disc or off-heap or any of these three combinations.
- API:
    - Used to process the data.

Section-17 (Data Types)

Numeric Types	
IntegerType()	4-byte signed integer numbers
FloatType()	4-byte single-precision floating point numbers
DoubleType()	8-byte double-precision floating point numbers

String Types	
StringType()	Character String Values
VarcharType(length)	Variant of StringType with Length limitation
CharType(length)	Variant of VarcharType with Fixed Length

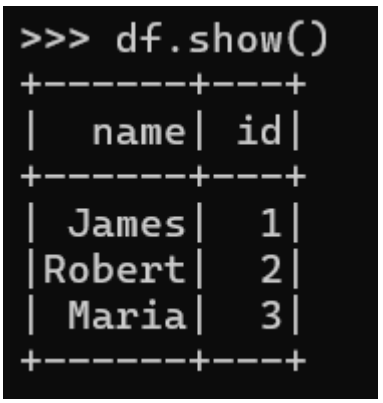
Boolean Types	
BooleanType ()	Boolean Values (True or False. Also can have Null Values)

Binary Type	
BinaryType ()	Byte Sequence Values

Date Types	
TimestampType()	year, month, day, hour, minute, second, time zone
DateType ()	year, month, day

Complex Type
ArrayType (elementType,containsNull)
MapType (keyType, valueType,valueContainsNull)
StructType (fields)

```
>>> data=[("Robert",1),("James",2)]
>>> data
[('Robert', 1), ('James', 2)]
>>> schema=['Name','Age']
>>> df=spark.createDataFrame(data=data, schema=schema)
>>> df.printSchema()
root
 |-- Name: string (nullable = true)
 |-- Age: long (nullable = true)

>>> df.show()
+------+---+
|  Name|Age|
+------+---+
|Robert|  1|
| James|  2|
+------+---+

>>> from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DataType
>>> schema=StructType([StructField("name",StringType(),True),StructField("id", IntegerType(),True),])
>>> data=[("James",1),("Robert",2),("Maria",3)]
>>> df=spark.createDataFrame(data=data,schema=schema)
>>> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- id: integer (nullable = true)
```


>>> df.show()
+------+---+
| name| id|
+------+---+
| James| 1|
|Robert| 2|
| Maria| 3|
+------+---+

Section-18 (DataFrame Rows)

```
>>> from pyspark.sql import Row
>>> row=Row(name='Robert',Age=35)
>>> row
Row(name='Robert', Age=35)
>>> type(row)
<class 'pyspark.sql.types.Row'>
>>> row.name
'Robert'
```

```
>>> row.Age
35
>>> 'name' in row
True
>>> 'name1' in row
False
>>> 'Robert' in row.name
True
>>> schema=['Name','Age']
>>> data=[Row(Name="Alicia",Age=11),Row(Name="Robert",Age=35)]
>>> rdd=sc.parallelize(data)
>>> for i in rdd.take(2):print(i.Name)
...
Alicia
Robert
```

```
>>> for i in rdd.take(2):print(i.Name + ' ' +str(i.Age))
...
Alicia 11
Robert 35
>>> df=spark.createDataFrame(data=data)
>>> df.printSchema()
root
 |-- Name: string (nullable = true)
 |-- Age: long (nullable = true)

>>> df=spark.createDataFrame(data=data,schema=['Name1','Age1'])
>>> df.printSchema()
root
 |-- Name1: string (nullable = true)
 |-- Age1: long (nullable = true)
```

```
>>> Person=Row("Name","Age")
>>> p1=Person("Alicia",31)
>>> p2=Person("RObert",35)
>>> print(p1.Name+' '+p2.Name)
Alicia RObert
>>> data=[Person("Alicia",31),Person("Robert",35)]
>>> rdd=sc.parallelize(data)
>>> for i in rdd.take(2):print(i.Age)
...
31
35
>>> df=spark.createDataFrame(data=data)
```

```
>>> df.printSchema()
root
 |-- Name: string (nullable = true)
 |-- Age: long (nullable = true)

>>> df.show()
+------+---+
|  Name|Age|
+------+---+
|Alicia| 31|
|Robert| 35|
+------+---+
```

```
>>> person=Row(name="Alice",age=11,username="Alice")
>>> person.count("Alice")
2
>>> person.index(11)
1
>>> person.asDict()
{'name': 'Alice', 'age': 11, 'username': 'Alice'}
```

Section-19 (DataFrame Columns)

```
>>> from pyspark.sql.types import StructType, StructField, IntegerType, StringType, TimestampType
>>> from datetime import datetime
>>> spark = SparkSession.builder.appName("RetailDB").getOrCreate()
25/03/16 14:10:53 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take effect.
>>> spark = SparkSession.builder.appName("RetailDB").getOrCreate();
>>> schema = StructType([
...     StructField("order_id", IntegerType(), False),
...     StructField("order_date", TimestampType(), False),
...     StructField("order_cust_id", IntegerType(), False),
...     StructField("order_status", StringType(), False)
... ])
>>> data = [
...     (1, datetime(2023, 1, 1, 10, 0, 0), 101, "COMPLETE"),
...     (2, datetime(2023, 1, 2, 11, 30, 0), 102, "PENDING"),
...     (3, datetime(2023, 1, 3, 14, 0, 0), 103, "CLOSED"),
...     (4, datetime(2023, 1, 4, 16, 45, 0), 104, "PENDING_PAYMENT"),
...     (5, datetime(2023, 1, 5, 18, 20, 0), 105, "COMPLETE")
... ]
>>> orders_df = spark.createDataFrame(data, schema=schema)
>>> orders_df.show()
+--------+-------------------+-------------+---------------+
|order_id|         order_date|order_cust_id|   order_status|
+--------+-------------------+-------------+---------------+
|       1|2023-01-01 10:00:00|          101|       COMPLETE|
|       2|2023-01-02 11:30:00|          102|        PENDING|
|       3|2023-01-03 14:00:00|          103|         CLOSED|
|       4|2023-01-04 16:45:00|          104|PENDING_PAYMENT|
|       5|2023-01-05 18:20:00|          105|       COMPLETE|
+--------+-------------------+-------------+---------------+
```

```
>>> orders_df.select(orders_df.order_id).show()
+--------+
|order_id|
+--------+
|       1|
|       2|
|       3|
|       4|
|       5|
+--------+
```

```
>>> from pyspark.sql.functions import col
>>> orders_df.select(col("*")).show()
+--------+-------------------+-------------+---------------+
|order_id|         order_date|order_cust_id|   order_status|
+--------+-------------------+-------------+---------------+
|       1|2023-01-01 10:00:00|          101|       COMPLETE|
|       2|2023-01-02 11:30:00|          102|        PENDING|
|       3|2023-01-03 14:00:00|          103|         CLOSED|
|       4|2023-01-04 16:45:00|          104|PENDING_PAYMENT|
|       5|2023-01-05 18:20:00|          105|       COMPLETE|
+--------+-------------------+-------------+---------------+
```

```
>>> orders_df.select(orders_df.order_id.alias("orderID")).show()
+-------+
|orderID|
+-------+
|      1|
|      2|
|      3|
|      4|
|      5|
+-------+
```

```
>>> orders_df.dtypes
[('order_id', 'int'), ('order_date', 'timestamp'), ('order_cust_id', 'int'), ('order_status', 'string')]
>>> ord=orders_df.select(orders_df.order_id.cast("string"))
>>> ord.dtypes
[('order_id', 'string')]
```

```
>>> orders_df.where(orders_df.order_id.between(1,2)).show()
+--------+-------------------+-------------+------------+
|order_id|         order_date|order_cust_id|order_status|
+--------+-------------------+-------------+------------+
|       1|2023-01-01 10:00:00|          101|    COMPLETE|
|       2|2023-01-02 11:30:00|          102|     PENDING|
+--------+-------------------+-------------+------------+
```

```
>>> orders_df.where(orders_df.order_status.contains('CLOSED')).show()
+--------+-------------------+-------------+------------+
|order_id|         order_date|order_cust_id|order_status|
+--------+-------------------+-------------+------------+
|       3|2023-01-03 14:00:00|          103|      CLOSED|
+--------+-------------------+-------------+------------+
```

```
>>> orders_df.where(orders_df.order_status.like('%L%')).show()
+--------+-------------------+-------------+------------+
|order_id|         order_date|order_cust_id|order_status|
+--------+-------------------+-------------+------------+
|       1|2023-01-01 10:00:00|          101|    COMPLETE|
|       3|2023-01-03 14:00:00|          103|      CLOSED|
|       5|2023-01-05 18:20:00|          105|    COMPLETE|
+--------+-------------------+-------------+------------+
```

```
>>> orders_df.where(orders_df.order_status.isin('CLOSED','PENDING')).select(orders_df.order_status).distinct().show()
+------------+
|order_status|
+------------+
|     PENDING|
|      CLOSED|
+------------+
```

```
>>> from pyspark.sql import Row
>>> df1=spark.createDataFrame([Row(id=1, value='foo'), Row(id=2, value=None) ])
>>> df1.printSchema()
root
 |-- id: long (nullable = true)
 |-- value: string (nullable = true)

>>> df1.show()
+---+-----+
| id|value|
+---+-----+
|  1|  foo|
|  2| NULL|
+---+-----+

>>> df1.select(df1.value=='foo').show()
+-------------+
|(value = foo)|
+-------------+
|         true|
|         NULL|
+-------------+

>>> df1.select(df1.value=='foo', df1.value.eqNullSafe('foo')).show()
+-------------+---------------+
|(value = foo)|(value <=> foo)|
+-------------+---------------+
|         true|           true|
|         NULL|          false|
+-------------+---------------+
```

```
>>> df1.select(df1.value=='foo', df1.value.isNull()).show()
+-------------+---------------+
|(value = foo)|(value IS NULL)|
+-------------+---------------+
|         true|          false|
|         NULL|           true|
+-------------+---------------+

>>> df1.select(df1.value=='foo', df1.value.isNotNull()).show()
+-------------+------------------+
|(value = foo)|(value IS NOT NULL)|
+-------------+------------------+
|         true|              true|
|         NULL|             false|
+-------------+------------------+
```

```
>>> orders_df.select(orders_df.order_date.substr(1,4)=='2013').show()
+--------------------------------+
|(substring(order_date, 1, 4) = 2013)|
+--------------------------------+
|                           false|
|                           false|
|                           false|
|                           false|
|                           false|
+--------------------------------+
```

```
>>> df1=spark.createDataFrame([Row(r=Row(a1=1, a2="b"))])
>>> df1=spark.createDataFrame([Row(r=Row(a1=1, a2="b")), Row(r=Row(a1=2, a2="bb"))])
>>> df1.printSchema()
root
 |-- r: struct (nullable = true)
 |    |-- a1: long (nullable = true)
 |    |-- a2: string (nullable = true)

>>> df1.show()
+-------+
|      r|
+-------+
| {1, b}|
|{2, bb}|
+-------+
```

```
>>> df1.select(df1.r.a1).show()
+----+
|r.a1|
+----+
|   1|
|   2|
+----+
```

```
>>> df=spark.createDataFrame([([1,2], {"key": "value"})], ["lst", "dict"])
>>> df.show()
+------+--------------+
|   lst|          dict|
+------+--------------+
|[1, 2]|{key -> value}|
+------+--------------+

>>> df.printSchema()
root
 |-- lst: array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- dict: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)

>>> df.select(df.lst[1]).show()
+------+
|lst[1]|
+------+
|     2|
+------+
```

Section-20 (DataFrame ETL)

```
>>> data=[('Robert',35,40,40), ('Robert',35,40,40), ('Ram',31,33,29), ('Ram',31,33,91)]
>>> emp=spark.createDataFrame(data=data,schema=['name','score1','score2','score3'])
>>> emp.show()
+------+------+------+------+
|  name|score1|score2|score3|
+------+------+------+------+
|Robert|    35|    40|    40|
|Robert|    35|    40|    40|
|   Ram|    31|    33|    29|
|   Ram|    31|    33|    91|
+------+------+------+------+

>>> orders_df.select(orders_df.order_id,'order_id',"order_id",(orders_df.order_id+10).alias('order10')).show()
+--------+--------+--------+-------+
|order_id|order_id|order_id|order10|
+--------+--------+--------+-------+
|       1|       1|       1|     11|
|       2|       2|       2|     12|
|       3|       3|       3|     13|
|       4|       4|       4|     14|
|       5|       5|       5|     15|
+--------+--------+--------+-------+
```

```
>>> from pyspark.sql.functions import lower
>>> orders_df.select(orders_df.order_status,lower(orders_df.order_status)).show()
+---------------+--------------------+
|   order_status|lower(order_status)|
+---------------+--------------------+
|       COMPLETE|            complete|
|        PENDING|             pending|
|         CLOSED|              closed|
|PENDING_PAYMENT|     pending_payment|
|       COMPLETE|            complete|
+---------------+--------------------+
```

```
>>> df=spark.range(1)
>>> df.show()
+---+
| id|
+---+
|  0|
+---+

>>> from pyspark.sql.functions import stack
>>> df.selectExpr("stack(3,1,2,3,4,5,6)").show()
+----+----+
|col0|col1|
+----+----+
|   1|   2|
|   3|   4|
|   5|   6|
+----+----+
```

```
>>> emp.dropDuplicates().show()
+------+------+------+------+
|  name|score1|score2|score3|
+------+------+------+------+
|Robert|    35|    40|    40|
|   Ram|    31|    33|    29|
|   Ram|    31|    33|    91|
+------+------+------+------+
```

```
>>> orders_df.where((orders_df.order_id>2) & (orders_df.order_id<5)).show()
+--------+-------------------+-------------+---------------+
|order_id|         order_date|order_cust_id|   order_status|
+--------+-------------------+-------------+---------------+
|       3|2023-01-03 14:00:00|          103|         CLOSED|
|       4|2023-01-04 16:45:00|          104|PENDING_PAYMENT|
+--------+-------------------+-------------+---------------+
```

```
>>> orders_df.sort(orders_df.order_date,orders_df.order_cust_id,ascending=[0,1]).show()
+--------+-------------------+-------------+---------------+
|order_id|         order_date|order_cust_id|   order_status|
+--------+-------------------+-------------+---------------+
|       5|2023-01-05 18:20:00|          105|       COMPLETE|
|       4|2023-01-04 16:45:00|          104|PENDING_PAYMENT|
|       3|2023-01-03 14:00:00|          103|         CLOSED|
|       2|2023-01-02 11:30:00|          102|        PENDING|
|       1|2023-01-01 10:00:00|          101|       COMPLETE|
+--------+-------------------+-------------+---------------+
```

```
>>> data=[('a',1),('d',4),('c',3),('b',2),('e',5)]
>>> df=spark.createDataFrame(data=data,schema='col1 string, col2 int')
>>> df.show()
+----+----+
|col1|col2|
+----+----+
|   a|   1|
|   d|   4|
|   c|   3|
|   b|   2|
|   e|   5|
+----+----+

>>> df.rdd.getNumPartitions()
8
>>> df.rdd.glom().collect()
[[], [Row(col1='a', col2=1)], [], [Row(col1='d', col2=4)], [Row(col1='c', col2=3)], [], [Row(col1='b', col2=2)], [Row(co
l1='e', col2=5)]]
>>> df.sort(df.col1.asc()).show()
+----+----+
|col1|col2|
+----+----+
|   a|   1|
|   b|   2|
|   c|   3|
|   d|   4|
|   e|   5|
+----+----+
```

```
>>> df.sortWithinPartitions(df.col1.asc()).show()
+----+----+
|col1|col2|
+----+----+
|   a|   1|
|   d|   4|
|   c|   3|
|   b|   2|
|   e|   5|
+----+----+
```

```
>>> df1=spark.range(10)
>>> df1.show()
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
+---+

>>> df1=spark.range(5,15)
>>> df1=spark.range(10)
```

```
>>> df2=spark.range(5,15)
>>> df1.show()
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
+---+

>>> df2.show()
+---+
| id|
+---+
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
| 11|
| 12|
| 13|
| 14|
+---+
```

```
>>> df1.union(df2).show()
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
| 11|
| 12|
| 13|
| 14|
+---+
```

```
>>> df=spark.createDataFrame(data=[('a',1),('a',1),('b',2)],schema=('col1 string, col2 int'))
>>> df1=spark.createDataFrame(data=[('a',1),('a',1),('c',2)],schema=('col1 string, col2 int'))
>>> df.intersect(df1).show()
+----+----+
|col1|col2|
+----+----+
|   a|   1|
+----+----+

>>> df.intersectAll(df1).show()
+----+----+
|col1|col2|
+----+----+
|   a|   1|
|   a|   1|
+----+----+
```

```
>>> df1=spark.range(1,10)
>>> df2=spark.range(5,10)
>>> df1.exceptAll(df2).show()
+---+
| id|
+---+
|  1|
|  2|
|  3|
|  4|
+---+
```

```
>>> df1=spark.createDataFrame(data=[(1,'Robert'),(2,'Ria'),(3,'James')],schema='empid int,empname string')
>>> df2=spark.createDataFrame(data=[(2,'USA'),(2,'India')],schema='empid int,country string')
>>> df1.show()
+-----+-------+
|empid|empname|
+-----+-------+
|    1| Robert|
|    2|    Ria|
|    3|  James|
+-----+-------+

>>> df2.show()
+-----+-------+
|empid|country|
+-----+-------+
|    2|    USA|
|    2|  India|
+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid).show()
+-----+-------+-----+-------+
|empid|empname|empid|country|
+-----+-------+-----+-------+
|    2|    Ria|    2|    USA|
|    2|    Ria|    2|  India|
+-----+-------+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid).select(df1.empid,df2.country).show()
+-----+-------+
|empid|country|
+-----+-------+
|    2|    USA|
|    2|  India|
+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid,'inner').select(df1.empid,df2.country).show()
+-----+-------+
|empid|country|
+-----+-------+
|    2|    USA|
|    2|  India|
+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid,'left').select(df1.empid,df2.country).show()
+-----+-------+
|empid|country|
+-----+-------+
|    1|   NULL|
|    2|  India|
|    2|    USA|
|    3|   NULL|
+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid,'right').select(df1.empid,df2.country).show()
+-----+-------+
|empid|country|
+-----+-------+
|    2|    USA|
|    2|  India|
+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid,'full').select(df1.empid,df2.country).show()
+-----+-------+
|empid|country|
+-----+-------+
|    1|   NULL|
|    2|    USA|
|    2|  India|
|    3|   NULL|
+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid,'cross').show()
+-----+-------+-----+-------+
|empid|empname|empid|country|
+-----+-------+-----+-------+
|    2|    Ria|    2|    USA|
|    2|    Ria|    2|  India|
+-----+-------+-----+-------+
```

```
>>> df1.join(df2,df1.empid==df2.empid,'leftanti').show()
+-----+-------+
|empid|empname|
+-----+-------+
|    1| Robert|
|    3|  James|
+-----+-------+
```

```
>>> df1.crossJoin(df2).show()
+-----+-------+-----+-------+
|empid|empname|empid|country|
+-----+-------+-----+-------+
|    1| Robert|    2|    USA|
|    1| Robert|    2|  India|
|    2|    Ria|    2|    USA|
|    2|    Ria|    2|  India|
|    3|  James|    2|    USA|
|    3|  James|    2|  India|
+-----+-------+-----+-------+
```

```
>>> from pyspark.sql.types import StructType, StructField, StringType, IntegerType
>>> spark = SparkSession.builder.appName("EmployeeData").getOrCreate();
25/03/17 10:26:48 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take effect.
>>> spark = SparkSession.builder.appName("EmployeeData").getOrCreate();
>>> schema = StructType([
...     StructField("empname", StringType(), True),
...     StructField("dept", StringType(), True),
...     StructField("state", StringType(), True),
...     StructField("salary", IntegerType(), True),
...     StructField("age", IntegerType(), True)
... ])
>>> data = [
...     ("Alice", "HR", "California", 70000, 30),
...     ("Bob", "IT", "Texas", 90000, 35),
...     ("Charlie", "Finance", "New York", 85000, 40),
...     ("David", "IT", "California", 95000, 28),
...     ("Emma", "Marketing", "Florida", 60000, 32),
...     ("Frank", "HR", "Texas", 75000, 45),
...     ("Grace", "Finance", "Illinois", 80000, 38),
...     ("Hank", "IT", "New York", 92000, 33),
...     ("Ivy", "Marketing", "Illinois", 62000, 29),
...     ("Jack", "Finance", "Florida", 78000, 41)
... ]
>>> df = spark.createDataFrame(data, schema=schema)
>>> df.show()
+-------+---------+----------+------+---+
|empname|     dept|     state|salary|age|
+-------+---------+----------+------+---+
|  Alice|       HR|California| 70000| 30|
|    Bob|       IT|     Texas| 90000| 35|
|Charlie|  Finance|  New York| 85000| 40|
|  David|       IT|California| 95000| 28|
|   Emma|Marketing|   Florida| 60000| 32|
|  Frank|       HR|     Texas| 75000| 45|
|  Grace|  Finance|  Illinois| 80000| 38|
|   Hank|       IT|  New York| 92000| 33|
|    Ivy|Marketing|  Illinois| 62000| 29|
|   Jack|  Finance|   Florida| 78000| 41|
+-------+---------+----------+------+---+
```

```
>>> df.groupBy(df.dept).avg("salary").show()
+---------+------------------+
|     dept|       avg(salary)|
+---------+------------------+
|       HR|           72500.0|
|       IT|92333.33333333333|
|  Finance|           81000.0|
|Marketing|           61000.0|
+---------+------------------+
```

```
>>> df.groupBy(df.dept,df.state).min("salary","age").show()
+---------+----------+-----------+--------+
|     dept|     state|min(salary)|min(age)|
+---------+----------+-----------+--------+
|       HR|California|      70000|      30|
|       IT|     Texas|      90000|      35|
|  Finance|  New York|      85000|      40|
|       IT|California|      95000|      28|
|Marketing|   Florida|      60000|      32|
|       HR|     Texas|      75000|      45|
|  Finance|  Illinois|      80000|      38|
|       IT|  New York|      92000|      33|
|Marketing|  Illinois|      62000|      29|
|  Finance|   Florida|      78000|      41|
+---------+----------+-----------+--------+
```

```
>>> from pyspark.sql.functions import *
>>> df.groupBy(df.dept).agg(min("salary"),max("salary"),avg("salary")).show()
+---------+-----------+-----------+-----------------+
|     dept|min(salary)|max(salary)|      avg(salary)|
+---------+-----------+-----------+-----------------+
|       HR|      70000|      75000|          72500.0|
|       IT|      90000|      95000|92333.33333333333|
|  Finance|      78000|      85000|          81000.0|
|Marketing|      60000|      62000|          61000.0|
+---------+-----------+-----------+-----------------+
```

```
>>> df.where(df.state=='New York').groupBy(df.dept).agg(min("salary"),max("salary"),avg("salary")).show()
+-------+-----------+-----------+-----------+
|   dept|min(salary)|max(salary)|avg(salary)|
+-------+-----------+-----------+-----------+
|Finance|      85000|      85000|    85000.0|
|     IT|      92000|      92000|    92000.0|
+-------+-----------+-----------+-----------+
```

```
>>> from pyspark.sql.window import *
>>> from pyspark.sql import window
>>> spec=Window.partitionBy(df.dept)
>>> Window.partitionBy(df.dept)
<pyspark.sql.window.WindowSpec object at 0x0000026E39825E50>
>>>  data = (("James","Sales","NY",9000,34),
```

```
>>> data=[("James","Sales","NY",9000,34),("Alicia","Sales","NY",8600,56),("Robert","Sales","CA",8100,56),
...      ("Lisa","Finance","CA",9000,24), ("Deja","Finance","CA",9900,40), ("Sugie","Finance","NY",8300,36),
...      ("Reid","Marketing","NY",9100,50)]
>>> schema=["empname","dept","state","salary","age"]
>>> df=spark.createDataFrame(data=data,schema=schema)
>>> df.show()
+-------+---------+-----+------+---+
|empname|     dept|state|salary|age|
+-------+---------+-----+------+---+
|  James|    Sales|   NY|  9000| 34|
| Alicia|    Sales|   NY|  8600| 56|
| Robert|    Sales|   CA|  8100| 56|
|   Lisa|  Finance|   CA|  9000| 24|
|   Deja|  Finance|   CA|  9900| 40|
|  Sugie|  Finance|   NY|  8300| 36|
|   Reid|Marketing|   NY|  9100| 50|
+-------+---------+-----+------+---+
```

```
>>> from pyspark.sql.window import *
>>> from pyspark.sql.functions import *
>>> spec=Window.partitionBy("dept").orderBy(df.salary.desc())
>>> spec
<pyspark.sql.window.WindowSpec object at 0x0000026E36CA0370>
>>> df.select(df.dept,df.salary).show()
+---------+------+
|     dept|salary|
+---------+------+
|    Sales|  9000|
|    Sales|  8600|
|    Sales|  8100|
|  Finance|  9000|
|  Finance|  9900|
|  Finance|  8300|
|Marketing|  9100|
+---------+------+
```

```
>>> spec=Window.partitionBy("dept").orderBy("salary")
>>> df.select(df.dept,df.salary).withColumn("lag_prev_sal",lag("salary",1,0).over(spec)).show()
+---------+------+------------+
|     dept|salary|lag_prev_sal|
+---------+------+------------+
|  Finance|  8300|           0|
|  Finance|  9000|        8300|
|  Finance|  9900|        9000|
|Marketing|  9100|           0|
|    Sales|  8100|           0|
|    Sales|  8600|        8100|
|    Sales|  9000|        8600|
+---------+------+------------+
```

```
>>> df.select(df.dept,df.salary).withColumn("lag_prev_sal",lag("salary",1,0).over(spec)).withColumn("lead_prev_sal",lead
("salary",2,9999).over(spec)).show()
+---------+------+------------+------------+
|     dept|salary|lag_prev_sal|lead_prev_sal|
+---------+------+------------+------------+
|  Finance|  8300|           0|        9900|
|  Finance|  9000|        8300|        9999|
|  Finance|  9900|        9000|        9999|
|Marketing|  9100|           0|        9999|
|    Sales|  8100|           0|        9000|
|    Sales|  8600|        8100|        9999|
|    Sales|  9000|        8600|        9999|
+---------+------+------------+------------+
```

```
>>> spec=Window.partitionBy("dept")
>>> df.select(df.dept,df.salary)
DataFrame[dept: string, salary: bigint]
>>> df.select(df.dept,df.salary).withColumn("sum_sal_per_dept",sum("salary").over(spec)).show()
+---------+------+----------------+
|     dept|salary|sum_sal_per_dept|
+---------+------+----------------+
|  Finance|  9000|           27200|
|  Finance|  9900|           27200|
|  Finance|  8300|           27200|
|Marketing|  9100|            9100|
|    Sales|  9000|           25700|
|    Sales|  8600|           25700|
|    Sales|  8100|           25700|
+---------+------+----------------+
```

```
>>> spec=Window.partitionBy("dept").orderBy(df.salary.desc())
>>> df.select(df.dept,df.salary).withColumn("Highest_Sal",first("salary").over(spec)).show()
+---------+------+-----------+
|     dept|salary|Highest_Sal|
+---------+------+-----------+
|  Finance|  9900|       9900|
|  Finance|  9000|       9900|
|  Finance|  8300|       9900|
|Marketing|  9100|       9100|
|    Sales|  9000|       9000|
|    Sales|  8600|       9000|
|    Sales|  8100|       9000|
+---------+------+-----------+
```

```
>>> dataset=spark.range(100)
>>> dataset.show()
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
| 11|
| 12|
| 13|
| 14|
| 15|
| 16|
| 17|
| 18|
| 19|
+---+
only showing top 20 rows
```

```
>>> dataset.count()
100
```

```
>>> dataset.sample(fraction=0.2).show()
+---+
| id|
+---+
|  5|
|  9|
| 14|
| 18|
| 23|
| 27|
| 32|
| 35|
| 36|
| 39|
| 51|
| 65|
| 73|
| 78|
| 87|
| 88|
| 96|
+---+
```

```
>>> dataset.sample(fraction=0.2,withReplacement=True).show()
+---+
| id|
+---+
|  4|
| 13|
| 13|
| 17|
| 26|
| 32|
| 35|
| 36|
| 37|
| 38|
| 38|
| 44|
| 46|
| 48|
| 57|
| 57|
| 59|
| 61|
| 76|
| 81|
+---+
only showing top 20 rows
```

```
>>> dataset=spark.range(100).select((col("id")%3).alias("key"))
>>> dataset.show()
+---+
|key|
+---+
|  0|
|  1|
|  2|
|  0|
|  1|
|  2|
|  0|
|  1|
|  2|
|  0|
|  1|
|  2|
|  0|
|  1|
|  2|
|  0|
|  1|
|  2|
|  0|
|  1|
+---+
only showing top 20 rows
```

```
>>> dataset.distinct().show()
+---+
|key|
+---+
|  0|
|  1|
|  2|
+---+
```

```
>>> sampled=dataset.sampleBy("key",fractions={0:0.1,1:0.2},seed=10)
>>> sampled.show()
+---+
|key|
+---+
|  0|
|  1|
|  1|
|  1|
|  1|
|  1|
|  1|
+---+
```

```
>>> df.select(first(df.salary)).show()
+--------------+
|first(salary)|
+--------------+
|          9000|
+--------------+
```

```
>>> df.select(greatest(df.salary,df.age)).show()
+---------------------+
|greatest(salary, age)|
+---------------------+
|                 9000|
|                 8600|
|                 8100|
|                 9000|
|                 9900|
|                 8300|
|                 9100|
+---------------------+
```

```
>>> df.select(skewness(df.salary)).show()
+-------------------+
|  skewness(salary)|
+-------------------+
|0.44909895428511117|
+-------------------+
```

```
>>> from pyspark.sql.functions import sha1, col
>>> df.select(df.age,sha1(df.age.cast('string'))).show(truncate=False)
+---+----------------------------------------+
|age|sha1(CAST(age AS STRING))               |
+---+----------------------------------------+
|30 |22d200f8670dbdb3e253a90eee5098477c95c23d|
|35 |972a67c48192728a34979d9a35164c1295401b71|
|40 |af3e133428b9e25c55bc59fe534248e6a0c0f17b|
|28 |0a57cb53ba59c46fc4b692527a38a87c78d84028|
|32 |cb4e5208b4cd87268b208e49452ed6e89a68e0b8|
|45 |fb644351560d8296fe6da332236b1f8d61b2828a|
|38 |5b384ce32d8cdef02bc3a139d4cac0a22bb029e8|
|33 |b6692ea5df920cad691c20319a6fffd7a4a766b8|
|29 |7719a1c782a1ba91c031a682a0a2f8658209adbf|
|41 |761f22b2c1593d0bb87e0b606f990ba4974706de|
+---+----------------------------------------+
```

```
>>> from pyspark.sql.functions import *
>>> df.select(df.age,sha2(df.age.cast('string'),0).alias('encr_age')).show(truncate=False)
+---+----------------------------------------------------------------+
|age|encr_age                                                        |
+---+----------------------------------------------------------------+
|30 |624b60c58c9d8bfb6ff1886c2fd605d2adeb6ea4da576068201b6c6958ce93f4|
|35 |9f14025af0065b30e47e23ebb3b491d39ae8ed17d33739e5ff3827ffb3634953|
|40 |d59eced1ded07f84c145592f65bdf854358e009c5cd705f5215bf18697fed103|
|28 |59e19706d51d39f66711c2653cd7eb1291c94d9b55eb14bda74ce4dc636d015a|
|32 |e29c9c180c6279b0b02abd6a1801c7c04082cf486ec027aa13515e4f3884bb6b|
|45 |811786ad1ae74adfdd20dd0372abaaebc6246e343aebd01da0bfc4c02bf0106c|
|38 |aea92132c4cbeb263e6ac2bf6c183b5d81737f179f21efdc5863739672f0f470|
|33 |c6f3ac57944a531490cd39902d0f777715fd005efac9a30622d5f5205e7f6894|
|29 |35135aaa6cc23891b40cb3f378c53a17a1127210ce60e125ccf03efcfdaec458|
|41 |3d914f9348c9cc0ff8a79716700b9fcd4d2f3e711608004eb8f138bcba7f14d9|
+---+----------------------------------------------------------------+
```

```
>>> df.select(df.age,hash(df.age)).show(truncate=False)
+---+-----------+
|age|hash(age)  |
+---+-----------+
|30 |1796998381 |
|35 |-1061350905|
|40 |1402250653 |
|28 |-1721654386|
|32 |1879002950 |
|45 |564777726  |
|38 |-622913472 |
|33 |401463163  |
|29 |-359179259 |
|41 |-1080649125|
+---+-----------+
```

```
>>> from pyspark.sql.functions import *
>>> ord.select(split(ord.order_date,'-').alias('order_date_list')).show(truncate=False)
+---------------+
|order_date_list|
+---------------+
|[2024, 03, 01] |
|[2024, 03, 02] |
|[2024, 03, 03] |
|[2024, 03, 04] |
|[2024, 03, 05] |
+---------------+
```

```
>>> df=spark.createDataFrame([('ab12cd23fe27kl',)],['s',])
>>> df.show()
+--------------+
|             s|
+--------------+
|ab12cd23fe27kl|
+--------------+
```

```
>>> ord.withColumn('length_status',length(ord.order_status)).show()
+--------+----------+-----------------+------------+-------------+
|order_id|order_date|order_customer_id|order_status|length_status|
+--------+----------+-----------------+------------+-------------+
|       1|2024-03-01|              101|     Shipped|            7|
|       2|2024-03-02|              102|     Pending|            7|
|       3|2024-03-03|              103|   Delivered|            9|
|       4|2024-03-04|              104|   Cancelled|            9|
|       5|2024-03-05|              105|  Processing|           10|
+--------+----------+-----------------+------------+-------------+
```

```
>>> ord.withColumn('initcap',initcap(ord.order_status)).show()
+--------+----------+-----------------+------------+----------+
|order_id|order_date|order_customer_id|order_status|   initcap|
+--------+----------+-----------------+------------+----------+
|       1|2024-03-01|              101|     Shipped|   Shipped|
|       2|2024-03-02|              102|     Pending|   Pending|
|       3|2024-03-03|              103|   Delivered| Delivered|
|       4|2024-03-04|              104|   Cancelled| Cancelled|
|       5|2024-03-05|              105|  Processing|Processing|
+--------+----------+-----------------+------------+----------+
```

```
>>> ord.withColumn('pading',lpad(ord.order_id,10,'0')).show()
+--------+----------+-----------------+------------+----------+
|order_id|order_date|order_customer_id|order_status|    pading|
+--------+----------+-----------------+------------+----------+
|       1|2024-03-01|              101|     Shipped|0000000001|
|       2|2024-03-02|              102|     Pending|0000000002|
|       3|2024-03-03|              103|   Delivered|0000000003|
|       4|2024-03-04|              104|   Cancelled|0000000004|
|       5|2024-03-05|              105|  Processing|0000000005|
+--------+----------+-----------------+------------+----------+
```

```
>>> df1=ord.withColumn('pading',lpad(ord.order_id,10,'0'))
>>> df1.printSchema()
root
 |-- order_id: integer (nullable = false)
 |-- order_date: date (nullable = false)
 |-- order_customer_id: integer (nullable = false)
 |-- order_status: string (nullable = false)
 |-- pading: string (nullable = false)

>>> df1=ord.withColumn('pading',lpad(ord.order_id,10,'#'))
>>> ord.withColumn('pading',lpad(ord.order_id,10,'#')).show()
+--------+----------+-----------------+------------+----------+
|order_id|order_date|order_customer_id|order_status|    pading|
+--------+----------+-----------------+------------+----------+
|       1|2024-03-01|              101|     Shipped|#########1|
|       2|2024-03-02|              102|     Pending|#########2|
|       3|2024-03-03|              103|   Delivered|#########3|
|       4|2024-03-04|              104|   Cancelled|#########4|
|       5|2024-03-05|              105|  Processing|#########5|
+--------+----------+-----------------+------------+----------+
```

```
>>> ord.withColumn('reversed_status',reverse(ord.order_status)).show()
+--------+----------+-----------------+------------+---------------+
|order_id|order_date|order_customer_id|order_status|reversed_status|
+--------+----------+-----------------+------------+---------------+
|       1|2024-03-01|              101|     Shipped|        deppihS|
|       2|2024-03-02|              102|     Pending|        gnidneP|
|       3|2024-03-03|              103|   Delivered|      derevileD|
|       4|2024-03-04|              104|   Cancelled|      dellecnaC|
|       5|2024-03-05|              105|  Processing|     gnissecorP|
+--------+----------+-----------------+------------+---------------+
```

```
>>> ord.withColumn('reversed_status',repeat(ord.order_status,3)).show(truncate=False)
+--------+----------+-----------------+------------+-----------------------------+
|order_id|order_date|order_customer_id|order_status|reversed_status              |
+--------+----------+-----------------+------------+-----------------------------+
|1       |2024-03-01|101              |Shipped     |ShippedShippedShipped        |
|2       |2024-03-02|102              |Pending     |PendingPendingPending        |
|3       |2024-03-03|103              |Delivered   |DeliveredDeliveredDelivered  |
|4       |2024-03-04|104              |Cancelled   |CancelledCancelledCancelled  |
|5       |2024-03-05|105              |Processing  |ProcessingProcessingProcessing|
+--------+----------+-----------------+------------+-----------------------------+
```

```
>>> ord.withColumn('IDStatus',concat(ord.order_id,ord.order_status)).show()
+--------+----------+-----------------+------------+-----------+
|order_id|order_date|order_customer_id|order_status|   IDStatus|
+--------+----------+-----------------+------------+-----------+
|       1|2024-03-01|              101|     Shipped|   1Shipped|
|       2|2024-03-02|              102|     Pending|   2Pending|
|       3|2024-03-03|              103|   Delivered| 3Delivered|
|       4|2024-03-04|              104|   Cancelled| 4Cancelled|
|       5|2024-03-05|              105|  Processing|5Processing|
+--------+----------+-----------------+------------+-----------+
```

```
>>> ord.withColumn('IDStatus',concat_ws(' ',ord.order_id,ord.order_status)).show()
+--------+----------+----------------+------------+------------+
|order_id|order_date|order_customer_id|order_status|    IDStatus|
+--------+----------+----------------+------------+------------+
|       1|2024-03-01|             101|     Shipped|   1 Shipped|
|       2|2024-03-02|             102|     Pending|   2 Pending|
|       3|2024-03-03|             103|   Delivered| 3 Delivered|
|       4|2024-03-04|             104|   Cancelled| 4 Cancelled|
|       5|2024-03-05|             105|  Processing|5 Processing|
+--------+----------+----------------+------------+------------+
```

```
>>> ord.withColumn('order_year',substring(ord.order_date,1,4)).show()
+--------+----------+----------------+------------+----------+
|order_id|order_date|order_customer_id|order_status|order_year|
+--------+----------+----------------+------------+----------+
|       1|2024-03-01|             101|     Shipped|      2024|
|       2|2024-03-02|             102|     Pending|      2024|
|       3|2024-03-03|             103|   Delivered|      2024|
|       4|2024-03-04|             104|   Cancelled|      2024|
|       5|2024-03-05|             105|  Processing|      2024|
+--------+----------+----------------+------------+----------+
```

```
>>> ord.withColumn('dummy',substring_index(ord.order_date,'-',1)).show()
+--------+----------+----------------+------------+-----+
|order_id|order_date|order_customer_id|order_status|dummy|
+--------+----------+----------------+------------+-----+
|       1|2024-03-01|             101|     Shipped| 2024|
|       2|2024-03-02|             102|     Pending| 2024|
|       3|2024-03-03|             103|   Delivered| 2024|
|       4|2024-03-04|             104|   Cancelled| 2024|
|       5|2024-03-05|             105|  Processing| 2024|
+--------+----------+----------------+------------+-----+
```

```
>>> ord.withColumn('instr',instr(ord.order_status,'LO')).show()
+--------+----------+----------------+------------+-----+
|order_id|order_date|order_customer_id|order_status|instr|
+--------+----------+----------------+------------+-----+
|       1|2024-03-01|             101|     Shipped|    0|
|       2|2024-03-02|             102|     Pending|    0|
|       3|2024-03-03|             103|   Delivered|    0|
|       4|2024-03-04|             104|   Cancelled|    0|
|       5|2024-03-05|             105|  Processing|    0|
+--------+----------+----------------+------------+-----+
```

```
>>> ord.withColumn('locate',locate('00',ord.order_date,2)).show()
+--------+----------+----------------+------------+------+
|order_id|order_date|order_customer_id|order_status|locate|
+--------+----------+----------------+------------+------+
|       1|2024-03-01|             101|     Shipped|     0|
|       2|2024-03-02|             102|     Pending|     0|
|       3|2024-03-03|             103|   Delivered|     0|
|       4|2024-03-04|             104|   Cancelled|     0|
|       5|2024-03-05|             105|  Processing|     0|
+--------+----------+----------------+------------+------+
```

```
>>> df=spark.createDataFrame([('translate',)],['col'])
>>> df.show()
+---------+
|      col|
+---------+
|translate|
+---------+
```

```
>>> df.withColumn('ex',translate(df.col,'rnit',"123")).show()
+---------+-------+
|      col|     ex|
+---------+-------+
|translate|1a2slae|
+---------+-------+
```

```
>>> df=spark.createDataFrame([("SPARK_SQL","CORE")],("x","y"))
>>> df.show()
+---------+----+
|        x|   y|
+---------+----+
|SPARK_SQL|CORE|
+---------+----+

>>> df.select(overlay("x","y",7).alias("overlayed")).show()
+----------+
| overlayed|
+----------+
|SPARK_CORE|
+----------+
```

```
>>> df=spark.createDataFrame(data=[('11ss1 ab',)],schema=['str'])
>>> df.show()
+--------+
|     str|
+--------+
|11ss1 ab|
+--------+
```

```
>>> from pyspark.sql.functions import *
>>> df.select(df.str,regexp_extract(df.str,'(\d)',1)).show()
+--------+----------------------------+
|     str|regexp_extract(str, (\d), 1)|
+--------+----------------------------+
|11ss1 ab|                           1|
+--------+----------------------------+

>>> df.select(df.str,regexp_extract(df.str,'(\d+)',1)).show()
+--------+-----------------------------+
|     str|regexp_extract(str, (\d+), 1)|
+--------+-----------------------------+
|11ss1 ab|                           11|
+--------+-----------------------------+

>>> df.select(df.str,regexp_extract(df.str,'(\d+)(\w+)',1)).show()
+--------+-------------------------------------+
|     str|regexp_extract(str, (\d+)(\w+), 1)|
+--------+-------------------------------------+
|11ss1 ab|                                 11|
+--------+-------------------------------------+
```

```
>>> df.select(df.str,regexp_extract(df.str,'(\d+)(\w+)(\s)',3)).show()
+--------+----------------------------------------+
|     str|regexp_extract(str, (\d+)(\w+)(\s), 3)|
+--------+----------------------------------------+
|11ss1 ab|                                        |
+--------+----------------------------------------+
```

```
>>> df.select(df.str,regexp_extract(df.str,'(\d+)(\w+)(\s)([a-z]+)',4)).show()
+--------+-------------------------------------------------+
|     str|regexp_extract(str, (\d+)(\w+)(\s)([a-z]+), 4)|
+--------+-------------------------------------------------+
|11ss1 ab|                                               ab|
+--------+-------------------------------------------------+
```

```
>>> df.select(df.str,regexp_replace(df.str,'(\d+)','xx')).show()
+--------+-----------------------------------+
|     str|regexp_replace(str, (\d+), xx, 1)|
+--------+-----------------------------------+
|11ss1 ab|                          xxssxx ab|
+--------+-----------------------------------+
```

```
>>> df=spark.createDataFrame(addr,["id","addr","city"])
>>> df.show(truncate=False)
+---+--------------------+--------+
|id |addr                |city    |
+---+--------------------+--------+
|1  |2625 Indian School Rd|Phoenix |
|2  |1234 Thomas St      |Glendale|
+---+--------------------+--------+

>>> df.withColumn('new_addr',when(df.addr.endswith('Rd'),regexp_replace(df.addr,'Rd','Road')) \
...     .when(df.addr.endswith('St'),regexp_replace(df.addr,'St','Street')) \
...     .otherwise(df.addr)) \
...     .show(truncate=False)
+---+--------------------+--------+---------------------+
|id |addr                |city    |new_addr             |
+---+--------------------+--------+---------------------+
|1  |2625 Indian School Rd|Phoenix |2625 Indian School Road|
|2  |1234 Thomas St      |Glendale|1234 Thomas Street   |
+---+--------------------+--------+---------------------+
```

```
>>> df=spark.createDataFrame(data=[('11ss1 ab',)],schema=['str'])
>>> df.show()
+--------+
|     str|
+--------+
|11ss1 ab|
+--------+

>>> df.select(df.str,df.str.rlike('(\d)')).show()
+--------+---------------+
|     str|RLIKE(str, (\d))|
+--------+---------------+
|11ss1 ab|           true|
+--------+---------------+

>>> df.select(df.str,df.str.rlike('(\s)')).show()
+--------+---------------+
|     str|RLIKE(str, (\s))|
+--------+---------------+
|11ss1 ab|           true|
+--------+---------------+

>>> df.select(df.str,df.str.rlike('(\s\s)')).show()
+--------+-----------------+
|     str|RLIKE(str, (\s\s))|
+--------+-----------------+
|11ss1 ab|            false|
+--------+-----------------+
```

```
>>> df.select(current_date()).show()
+--------------+
|current_date()|
+--------------+
|    2025-03-17|
+--------------+
```

```
>>> df.select(current_timestamp()).show()
+--------------------+
| current_timestamp()|
+--------------------+
|2025-03-17 15:53:...|
+--------------------+
```

```
>>> df.select(current_timestamp()).show(truncate=False)
+--------------------------+
|current_timestamp()       |
+--------------------------+
|2025-03-17 15:54:10.704396|
+--------------------------+
```

```
>>> df.select(df.order_date,dayofyear(df.order_date)).distinct().show()
+----------+--------------------+
|order_date|dayofyear(order_date)|
+----------+--------------------+
|2024-03-01|                  61|
|2024-03-02|                  62|
|2024-03-03|                  63|
|2024-03-04|                  64|
|2024-03-05|                  65|
+----------+--------------------+
```

```
>>> df.select(df.order_date,weekofyear(df.order_date)).distinct().show()
+----------+---------------------+
|order_date|weekofyear(order_date)|
+----------+---------------------+
|2024-03-01|                    9|
|2024-03-02|                    9|
|2024-03-03|                    9|
|2024-03-04|                   10|
|2024-03-05|                   10|
+----------+---------------------+
```

```
>>> df.select(df.order_date,second(df.order_date)).distinct().show()
+----------+-----------------+
|order_date|second(order_date)|
+----------+-----------------+
|2024-03-01|                0|
|2024-03-02|                0|
|2024-03-03|                0|
|2024-03-04|                0|
|2024-03-05|                0|
+----------+-----------------+
```

```
>>> ord.select(months_between("new_order_date","order_date")).show()
+-----------------------------------------------+
|months_between(new_order_date, order_date, true)|
+-----------------------------------------------+
|                                     1.61290323|
|                                     1.61290323|
|                                     1.61290323|
|                                     1.61290323|
|                                     1.61290323|
+-----------------------------------------------+
```

```
>>> ord.select(months_between("new_order_date","order_date",roundOff=False)).show()
+------------------------------------------------+
|months_between(new_order_date, order_date, false)|
+------------------------------------------------+
|                              1.6129032258064515|
|                              1.6129032258064515|
|                              1.6129032258064515|
|                              1.6129032258064515|
|                              1.6129032258064515|
+------------------------------------------------+
```

```
>>> df.select(df.order_date,add_months(df.order_date,5)).show()
+----------+------------------------+
|order_date|add_months(order_date, 5)|
+----------+------------------------+
|2024-03-01|              2024-08-01|
|2024-03-02|              2024-08-02|
|2024-03-03|              2024-08-03|
|2024-03-04|              2024-08-04|
|2024-03-05|              2024-08-05|
+----------+------------------------+
```

```
>>> ord.select(datediff("new_order_date","order_date")).show()
+-----------------------------------+
|datediff(new_order_date, order_date)|
+-----------------------------------+
|                                 50|
|                                 50|
|                                 50|
|                                 50|
|                                 50|
+-----------------------------------+
```

```
>>> df.withColumn('order_year',date_trunc('mm',df.order_date)).show()
+--------+----------+-----------------+------------+-------------------+
|order_id|order_date|order_customer_id|order_status|         order_year|
+--------+----------+-----------------+------------+-------------------+
|       1|2024-03-01|              101|     Shipped|2024-03-01 00:00:00|
|       2|2024-03-02|              102|     Pending|2024-03-01 00:00:00|
|       3|2024-03-03|              103|   Delivered|2024-03-01 00:00:00|
|       4|2024-03-04|              104|   Cancelled|2024-03-01 00:00:00|
|       5|2024-03-05|              105|  Processing|2024-03-01 00:00:00|
+--------+----------+-----------------+------------+-------------------+
```

```
>>> df.withColumn('new_order_date',date_format(df.order_date,'yyyy/MM/dd')).show()
+--------+----------+-----------------+------------+--------------+
|order_id|order_date|order_customer_id|order_status|new_order_date|
+--------+----------+-----------------+------------+--------------+
|       1|2024-03-01|              101|     Shipped|    2024/03/01|
|       2|2024-03-02|              102|     Pending|    2024/03/02|
|       3|2024-03-03|              103|   Delivered|    2024/03/03|
|       4|2024-03-04|              104|   Cancelled|    2024/03/04|
|       5|2024-03-05|              105|  Processing|    2024/03/05|
+--------+----------+-----------------+------------+--------------+
```

```
>>> spark.range(1).select(current_timestamp(),unix_timestamp(current_timestamp())).show()
+-------------------+-----------------------------------------------------------+
| current_timestamp()|unix_timestamp(current_timestamp(), yyyy-MM-dd HH:mm:ss)|
+-------------------+-----------------------------------------------------------+
|2025-03-17 16:06:...|                                                 1742207803|
+-------------------+-----------------------------------------------------------+
```

```
>>> df1=spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df1.show()
+-------------------+
|                  t|
+-------------------+
|1997-02-28 10:30:00|
+-------------------+
```

```
>>> time_df=spark.createDataFrame([(1428476400,)],['unix_time'])
>>> time_df.show()
+----------+
| unix_time|
+----------+
|1428476400|
+----------+

>>> time_df.select(from_unixtime('unix_time')).show()
+-------------------------------------------+
|from_unixtime(unix_time, yyyy-MM-dd HH:mm:ss)|
+-------------------------------------------+
|                        2015-04-08 12:30:00|
+-------------------------------------------+

>>> df=spark.createDataFrame([('1997-02-28 10:30:00', 'JST')], ['ts', 'tz'])
>>> df.show()
+-------------------+---+
|                 ts| tz|
+-------------------+---+
|1997-02-28 10:30:00|JST|
+-------------------+---+
```

```
>>> df.select(from_utc_timestamp(df.ts,'PST')).show()
+---------------------------+
|from_utc_timestamp(ts, PST)|
+---------------------------+
|        1997-02-28 02:30:00|
+---------------------------+

>>> df.select(df.ts,from_utc_timestamp(df.ts,'PST')).show()
+-------------------+---------------------------+
|                 ts|from_utc_timestamp(ts, PST)|
+-------------------+---------------------------+
|1997-02-28 10:30:00|        1997-02-28 02:30:00|
+-------------------+---------------------------+
```

```
>>> df=spark.createDataFrame([('Robert',1,None,114.0),('John',None,2577,float('nan'))],("name","id","phone","stdAdd"))
>>> df.show()
+------+----+-----+------+
|  name|  id|phone|stdAdd|
+------+----+-----+------+
|Robert|   1| NULL| 114.0|
|  John|NULL| 2577|   NaN|
+------+----+-----+------+
```

```
>>> df.select(df.phone,isnull(df.phone)).show()
+-----+----------------+
|phone|(phone IS NULL)|
+-----+----------------+
| NULL|            true|
| 2577|           false|
+-----+----------------+
```

```
>>> data=[('Alice',80,10),('Bob',None,5),('Tom',50,50),(None,None,None),('Robert',30,35)]
>>> schema='name string, age int, height int'
>>> df=spark.createDataFrame(data,schema)
>>> df.show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|  NULL|NULL|  NULL|
|Robert|  30|    35|
+------+----+------+

>>> df.na.drop().show()
+------+---+------+
|  name|age|height|
+------+---+------+
| Alice| 80|    10|
|   Tom| 50|    50|
|Robert| 30|    35|
+------+---+------+

>>> df.na.drop(how='all').show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|Robert|  30|    35|
+------+----+------+
```

```
>>> df.na.drop(thresh=1).show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|Robert|  30|    35|
+------+----+------+

>>> df.na.drop(thresh=2).show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|Robert|  30|    35|
+------+----+------+

>>> df.na.drop(thresh=3).show()
+------+---+------+
|  name|age|height|
+------+---+------+
| Alice| 80|    10|
|   Tom| 50|    50|
|Robert| 30|    35|
+------+---+------+

>>> df.na.drop(thresh=4).show()
+----+---+------+
|name|age|height|
+----+---+------+
+----+---+------+
```

```
>>> df.show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|  NULL|NULL|  NULL|
|Robert|  30|    35|
+------+----+------+
```

```
>>> df.na.drop(subset='age').show()
+------+---+------+
|  name|age|height|
+------+---+------+
| Alice| 80|    10|
|   Tom| 50|    50|
|Robert| 30|    35|
+------+---+------+

>>> df.na.drop(subset='name').show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|Robert|  30|    35|
+------+----+------+
```

```
>>> df.na.fill(50).show()
+------+---+------+
|  name|age|height|
+------+---+------+
| Alice| 80|    10|
|   Bob| 50|     5|
|   Tom| 50|    50|
|  NULL| 50|    50|
|Robert| 30|    35|
+------+---+------+
```

```
>>> df.na.fill({'age':50}).show()
+------+---+------+
|  name|age|height|
+------+---+------+
| Alice| 80|    10|
|   Bob| 50|     5|
|   Tom| 50|    50|
|  NULL| 50|  NULL|
|Robert| 30|    35|
+------+---+------+

>>> df.na.fill({'name':'Ram'}).show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  50|    50|
|   Ram|NULL|  NULL|
|Robert|  30|    35|
+------+----+------+
```

```
>>> df.na.replace(50,99).show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Bob|NULL|     5|
|   Tom|  99|    99|
|  NULL|NULL|  NULL|
|Robert|  30|    35|
+------+----+------+
```

```
>>> df.na.replace({'Alicia':'Alex','Bob':'Cob'},subset='name').show()
+------+----+------+
|  name| age|height|
+------+----+------+
| Alice|  80|    10|
|   Cob|NULL|     5|
|   Tom|  50|    50|
|  NULL|NULL|  NULL|
|Robert|  30|    35|
+------+----+------+
```

```
>>> from pyspark.sql import Row
>>> spark = SparkSession.builder.appName("DataFrameExample").getOrCreate();
25/03/18 10:39:34 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take effect.
>>> spark = SparkSession.builder.appName("DataFrameExample").getOrCreate();
>>> df1 = spark.createDataFrame([
...     Row(col1=-5, col2=0),
...     Row(col1=1, col2=3),
...     Row(col1=7, col2=9)
... ])
>>> df1.show()
+----+----+
|col1|col2|
+----+----+
|  -5|   0|
|   1|   3|
|   7|   9|
+----+----+

>>> from pyspark.sql.functions import *
>>> df1.select(df1.col1,abs(df1.col1)).show()
+----+---------+
|col1|abs(col1)|
+----+---------+
|  -5|        5|
|   1|        1|
|   7|        7|
+----+---------+
```

```
>>> df1.select(df1.col1,exp(df1.col1)).show()
+----+------------------+
|col1|          EXP(col1)|
+----+------------------+
|  -5|0.006737946999085467|
|   1| 2.7182818284590455|
|   7| 1096.6331584284585|
+----+------------------+

>>> df1.select(df1.col1,factorial(df1.col1)).show()
+----+---------------+
|col1|factorial(col1)|
+----+---------------+
|  -5|           NULL|
|   1|              1|
|   7|           5040|
+----+---------------+
```

```
>>> df1.select(df1.col1,sqrt(df1.col1)).show()
+----+------------------+
|col1|        SQRT(col1)|
+----+------------------+
|  -5|               NaN|
|   1|               1.0|
|   7|2.6457513110645907|
+----+------------------+
```

```
>>> df2 = spark.createDataFrame([Row(col1=5.4,col2=3),Row(col1=7.9,col2=1),Row(col1=0.7,col2=3)])
>>> df2.show()
+----+----+
|col1|col2|
+----+----+
| 5.4|   3|
| 7.9|   1|
| 0.7|   3|
+----+----+

>>> df2.select(df2.col1,floor(df2.col1),ceil(df2.col1)).show()
+----+-----------+----------+
|col1|FLOOR(col1)|CEIL(col1)|
+----+-----------+----------+
| 5.4|          5|         6|
| 7.9|          7|         8|
| 0.7|          0|         1|
+----+-----------+----------+
```

```
>>> df2.select(df2.col1,round(df2.col1,0)).show()
+----+--------------+
|col1|round(col1, 0)|
+----+--------------+
| 5.4|           5.0|
| 7.9|           8.0|
| 0.7|           1.0|
+----+--------------+
```

```
>>> order_df.select(order_df.order_date,trunc(order_df.order_date,'yyyy')).show()
+----------+-----------------------+
|order_date|trunc(order_date, yyyy)|
+----------+-----------------------+
|2024-03-01|             2024-01-01|
|2024-03-02|             2024-01-01|
|2024-03-03|             2024-01-01|
|2024-03-04|             2024-01-01|
|2024-03-05|             2024-01-01|
|2024-03-06|             2024-01-01|
|2024-03-07|             2024-01-01|
|2024-03-08|             2024-01-01|
|2024-03-09|             2024-01-01|
|2024-03-10|             2024-01-01|
|2024-03-11|             2024-01-01|
|2024-03-12|             2024-01-01|
|2024-03-13|             2024-01-01|
|2024-03-14|             2024-01-01|
|2024-03-15|             2024-01-01|
|2024-03-16|             2024-01-01|
|2024-03-17|             2024-01-01|
|2024-03-18|             2024-01-01|
|2024-03-19|             2024-01-01|
|2024-03-20|             2024-01-01|
+----------+-----------------------+
```

```
>>> order_df.select(order_df.order_date,trunc(order_df.order_date,'month')).show()
+----------+------------------------+
|order_date|trunc(order_date, month)|
+----------+------------------------+
|2024-03-01|              2024-03-01|
|2024-03-02|              2024-03-01|
|2024-03-03|              2024-03-01|
|2024-03-04|              2024-03-01|
|2024-03-05|              2024-03-01|
|2024-03-06|              2024-03-01|
|2024-03-07|              2024-03-01|
|2024-03-08|              2024-03-01|
|2024-03-09|              2024-03-01|
|2024-03-10|              2024-03-01|
|2024-03-11|              2024-03-01|
|2024-03-12|              2024-03-01|
|2024-03-13|              2024-03-01|
|2024-03-14|              2024-03-01|
|2024-03-15|              2024-03-01|
|2024-03-16|              2024-03-01|
|2024-03-17|              2024-03-01|
|2024-03-18|              2024-03-01|
|2024-03-19|              2024-03-01|
|2024-03-20|              2024-03-01|
+----------+------------------------+
```

```
>>> df1.select(df1.col1,signum(df1.col1)).show()
+----+------------+
|col1|SIGNUM(col1)|
+----+------------+
|  -5|        -1.0|
|   1|         1.0|
|   7|         1.0|
+----+------------+
```

```
>>> order_df.select(corr(order_df.order_id,order_df.order_customer_id)).show()
+--------------------------------+
|corr(order_id, order_customer_id)|
+--------------------------------+
|                             1.0|
+--------------------------------+
```

```
>>> df=spark.createDataFrame([(5,"hello")],['a','b'])
>>> df.show()
+---+-----+
|  a|    b|
+---+-----+
|  5|hello|
+---+-----+

>>> df.select(format_string('%d %s',df.a,df.b).alias('v')).show()
+-------+
|      v|
+-------+
|5 hello|
+-------+

>>> df.select(format_string('%d *** %s',df.a,df.b).alias('v')).show()
+-----------+
|          v|
+-----------+
|5 *** hello|
+-----------+
```

```
>>> data=[(1,"""{"Zipcode":85016,"ZipCodeType":"STANDARD","City":"Phoenix","State":"AZ"}""")]
>>> df_map=spark.createDataFrame(data,["id","value"])
>>> df_map.show()
+---+--------------------+
| id|               value|
+---+--------------------+
|  1|{"Zipcode":85016,...|
+---+--------------------+

>>> df_map.show(truncate=False)
+---+-------------------------------------------------------------------------------+
|id |value                                                                          |
+---+-------------------------------------------------------------------------------+
|1  |{"Zipcode":85016,"ZipCodeType":"STANDARD","City":"Phoenix","State":"AZ"}|
+---+-------------------------------------------------------------------------------+
```

```
>>> data=[(1, '''[1,2,3]''')]
>>> df_arr=spark.createDataFrame(data,["id","value"])
>>> df_arr.show()
+---+-------+
| id|  value|
+---+-------+
|  1|[1,2,3]|
+---+-------+

>>> from pyspark.sql.types import *
>>> schema=MapType(StringType(),StringType())
>>> df_map.withColumn('map_column',from_json(df_map.value,schema)).printSchema()
root
 |-- id: long (nullable = true)
 |-- value: string (nullable = true)
 |-- map_column: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)

>>> df_map_new=df_map.withColumn('map_column',from_json(df_map.value,schema))
>>> df_map_new.show(truncate=False)
+---+-------------------------------------------------------------------------------+-----------------------------------------------------------------------------------------+
|id |value                                                                          |map_column                                                                               |
+---+-------------------------------------------------------------------------------+-----------------------------------------------------------------------------------------+
|1  |{"Zipcode":85016,"ZipCodeType":"STANDARD","City":"Phoenix","State":"AZ"}|{Zipcode -> 85016, ZipCodeType -> STANDARD, City -> Phoenix, State -> AZ}|
+---+-------------------------------------------------------------------------------+-----------------------------------------------------------------------------------------+
```

```
>>> schema=ArrayType(IntegerType())
>>> df_arr.show()
+---+-------+
| id|  value|
+---+-------+
|  1|[1,2,3]|
+---+-------+

>>> df_arr.withColumn('arr_column',from_json(df_arr.value,schema)).printSchema()
root
 |-- id: long (nullable = true)
 |-- value: string (nullable = true)
 |-- arr_column: array (nullable = true)
 |    |-- element: integer (containsNull = true)

>>> df_arr_new=df_arr.withColumn('arr_column',from_json(df_arr.value,schema))
>>> df_arr_new.printSchema()
root
 |-- id: long (nullable = true)
 |-- value: string (nullable = true)
 |-- arr_column: array (nullable = true)
 |    |-- element: integer (containsNull = true)
```

```
>>> df_arr_new.show()
+---+-------+----------+
| id|  value|arr_column|
+---+-------+----------+
|  1|[1,2,3]| [1, 2, 3]|
+---+-------+----------+

>>> df_arr_new.select(df_arr_new.arr_column[1]).show()
+-------------+
|arr_column[1]|
+-------------+
|            2|
+-------------+
```

What is a Partition?

A **partition** is like a small chunk of data that is stored on a single computer in the cluster. These partitions help Spark process data **in parallel**, making it much faster.

Important Points About Partitions:

1. A **partition is the smallest unit of data** that Spark works with in parallel.
2. **One partition stays on one machine**—it does not split across multiple machines.
3. **Spark automatically partitions RDDs and DataFrames** and distributes them across nodes in the cluster.
4. **We can configure** the number of partitions to improve performance. Having **too many or too few** partitions is not ideal.

How Does Spark Decide the Default Number of Partitions?

- Spark checks the **HDFS block size** (the storage unit in Hadoop).
- In **Hadoop 1.0**, the default block size is **64MB**.
- In **Hadoop 2.0/YARN**, the default block size is **128MB**.
- Spark creates **one partition per block**.

Example:
If you have a **500MB file** and your block size is **128MB**, Spark will create **4 partitions** (500 ÷ 128 ≈ 4).

Can We Change the Number of Partitions?

Yes! Sometimes, we need to **increase or decrease** the number of partitions based on our application's needs.
To do this, we use **two functions**:

1. **repartition(n)** – Increases or decreases partitions but may involve shuffling data.
2. **coalesce(n)** – Reduces the number of partitions efficiently, avoiding unnecessary shuffling.

Partitioning helps **speed up big data processing** by dividing work among multiple machines.

Features	Repartition	Coalesce
Shuffle	Full shuffle	No full shuffle (less expensive)
Partitions	Creates new partitions	Merges existing partitions
Use Case	Increasing partitions, balancing load	Decreasing partitions, avoiding shuffle
Speed	Slower (more data movement)	Faster (less data movement)

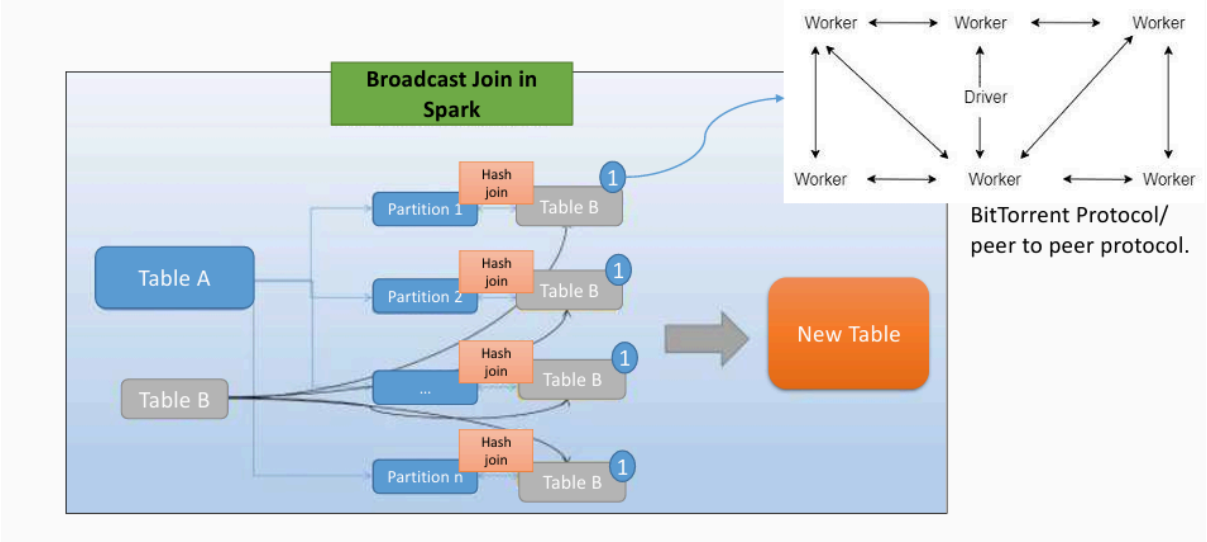
Section-21 (Performance and Optimization)

Spark has below important join strategies:

1. Broadcast Join

It is divided into two steps:

- Broadcast the smallest dataset to all executors.
- Perform a hash join.



- Imagine we have two tables:
    - **Table A** (a huge table)
    - **Table B** (a small table)
- The size of **Table B** can be controlled using a configuration setting in Spark:
  `spark.sql.autoBroadcastJoinThreshold`
    - Default value = **10MB**
    - If **Table B** is **≤ 10MB**, Spark can **broadcast** it.

Step-by-Step Process of Broadcast Join

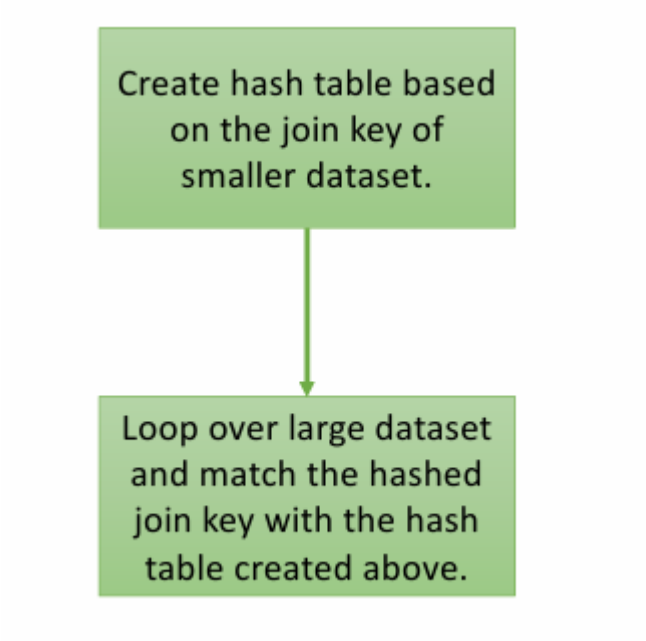
Step 1: Broadcasting the Smaller Table

- Since **Table B** is small (≤ 10MB), the **driver node** (Spark's coordinator) **sends Table B** to all the **executor nodes** (worker machines running tasks).
- This is done using a **peer-to-peer protocol** like **BitTorrent** (The driver would send the first data set to any two worker nodes, and afterwards the worker nodes will send the copy of the data file to each other), making the transfer **super fast**.

Step 2: Hash Join Execution

- Spark performs a **hash join**, which only works for **equality-based joins** (= operator).

- **How Hash Join Works:**
    - Spark **creates a hash table** from the **join column** in **Table B** (small table).
    - It then **scans Table A** and **matches** rows based on the hash table.



What is Broadcast Join?

Broadcast Join is a powerful technique in **Apache Spark** that helps speed up **joining two datasets** by sending the smaller dataset to all worker nodes (**executors**).

How Does it Work?

- **Spark identifies the smaller table** (less than **10MB** by default).
- **This small table is "broadcasted" (copied)** to all Spark executors using **BitTorrent protocol** (a fast peer-to-peer method).
- **Now, each executor has all the required data**, so the join happens **locally** without needing data movement (**no shuffling**).

Why is Broadcast Join Fast?

- **Avoids data movement** (shuffling), making it much faster.
- Works best when joining a **large table** (fact table) with a **small table** (dimension table) in **star-schema joins** (common in data warehouses).

Important Points:

- The **default size limit** for the smaller table is **10MB** (can be changed using `spark.sql.autoBroadcastJoinThreshold`).
- **Recent Spark versions support broadcasting up to 8GB** (previously 2GB).
- Also called **"Map-Side Join"** or **"Replicated Join"** because the small table is copied everywhere.
- You can **force Spark** to use Broadcast Join by adding a hint:

```
from pyspark.sql.functions import
broadcast

df_joined =
largeDF.join(broadcast(smallDF), "id")
```

What is Auto Detection?

Auto detection means **Spark can automatically decide** whether to use **Broadcast Join** based on the **size of the data**.

How Does it Work?

- If one of the DataFrames in the join is **small (≤ 10MB by default)**, Spark **automatically** broadcasts it.
- This happens **without** us needing to manually specify a Broadcast Join.

When Does Auto Detection Work?

- **When Spark constructs a DataFrame from scratch**, like using `spark.range()`.
- **When Spark reads data from files** that contain size information, such as **Parquet or Avro** files.

When Auto Detection Doesn't Work

- Spark **cannot** automatically determine the size of a **local collection** (data stored in Python lists, dictionaries, etc.).
- This is because local collections **could be large**, and Spark doesn't want to assume it's small.
- This tells Spark:
    - **Treat `smallDF` as a small table**
    - **Broadcast it manually to all nodes**

Why Test Without Auto Optimization?

Sometimes, we may want to **disable Spark's automatic Broadcast Join detection** to:

- **Manually test different join strategies**
- **Understand how Spark behaves without optimization**
- **Compare performance with and without Broadcast Join**

How to Disable Auto Broadcast Join?

- Spark automatically decides to **broadcast a small table** (≤ 10MB by default).
- To **turn off** this automatic detection, we set:

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

- This tells Spark **not to use Broadcast Join**, even if the table is small.

What Happens Now?

- Spark **will not broadcast** `smallDF`, even though it's small.
- Instead, it will use a **regular join**, which may be slower.
- This helps us **test the difference** between a normal join and a Broadcast Join.

Example:

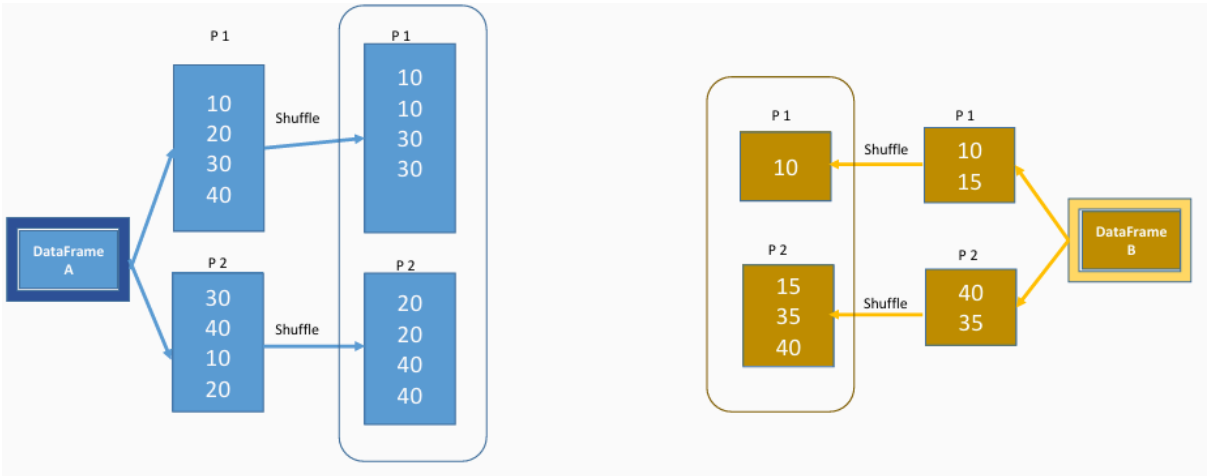
```
>>> from pyspark.sql.functions import broadcast
>>> spark = SparkSession.builder \
...     .appName("BroadcastJoinExample") \
...     .getOrCreate()
25/03/19 10:36:48 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take effect.
>>> spark = SparkSession.builder.appName("BroadcastJoinExample").getOrCreate();
>>> largeDF = spark.range(1, 10000000)
>>> data = [(1, 'A'), (2, 'B'), (3, 'C')]
>>> schema = ['id', 'name']
>>> smallDF = spark.createDataFrame(data, schema)
>>> joinedDF = largeDF.join(broadcast(smallDF), "id")
>>> joinedDF.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [id#0L, name#3]
   +- BroadcastHashJoin [id#0L], [id#2L], Inner, BuildRight, false
      :- Range (1, 10000000, step=1, splits=8)
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]),false), [plan_id=20]
         +- Filter isnotnull(id#2L)
            +- Scan ExistingRDD[id#2L,name#3]
```

2. Shuffle Hash Join

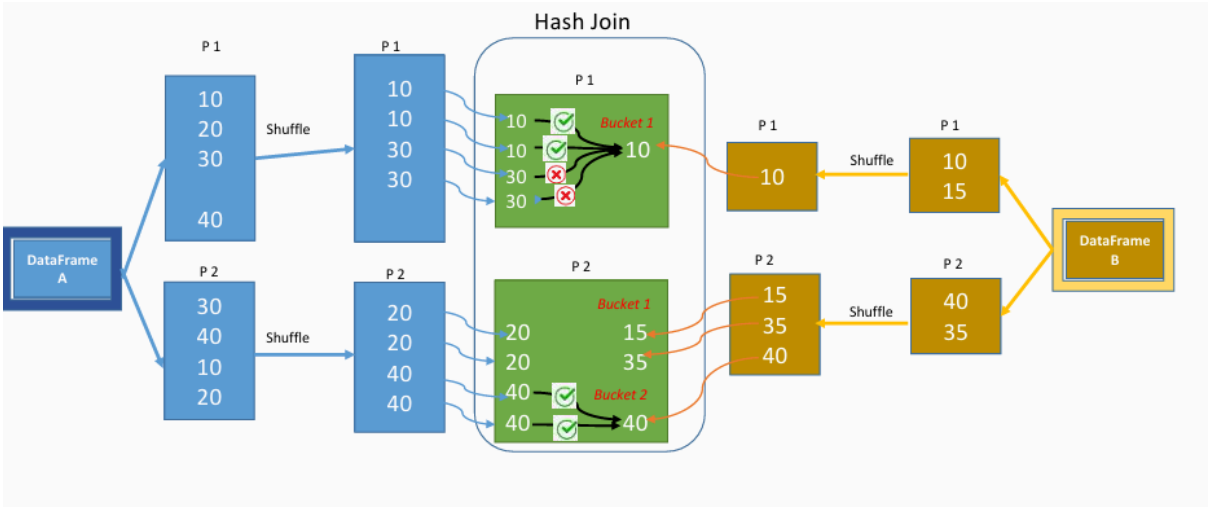
Shuffle Hash Join is a **two-step** join process used in Spark when working with large datasets that **cannot be broadcasted**.

It consists of **two phases**:

- **Shuffle Phase** – Data is **rearranged** so that matching keys from both datasets end up in the **same partition**.



- **Hash Join Phase** – The **smaller** dataset is placed into a **hash table**, and the larger dataset is **matched** against it.



Example:

```
>>> df1=spark.range(1,10000000)
>>> df2=spark.range(1,100000)
>>> joined=df1.join(df2,"id")
>>> joined.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [id#0L]
   +- BroadcastHashJoin [id#0L], [id#2L], Inner, BuildRight, false
      :- Range (1, 10000000, step=1, splits=8)
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]),false), [plan_id=16]
         +- Range (1, 100000, step=1, splits=8)
```

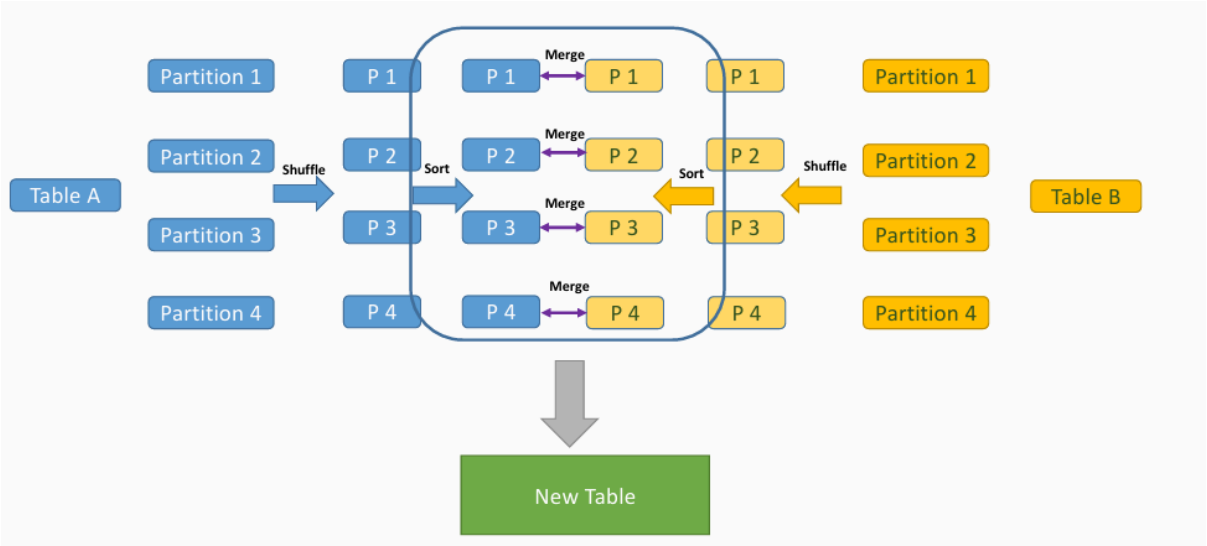
```
>>> joined=df1.hint('shuffle_hash').join(df2,"id")
>>> joined.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [id#0L]
   +- ShuffledHashJoin [id#0L], [id#2L], Inner, BuildLeft
      :- Exchange hashpartitioning(id#0L, 200), ENSURE_REQUIREMENTS, [plan_id=36]
      :  +- Range (1, 10000000, step=1, splits=8)
      +- Exchange hashpartitioning(id#2L, 200), ENSURE_REQUIREMENTS, [plan_id=37]
         +- Range (1, 100000, step=1, splits=8)
```

3. Sort Merge Join

Sort Merge Join is a **two-step** join process used in Spark for **large datasets** that do not fit in memory.
It is **the default join strategy** in Spark 2.3 and later (if broadcast join is not possible).

It consists of **two phases**:

- **Shuffle & Sort Phase** – Both datasets are **shuffled** and **sorted** on the join key.
- **Merge Phase** – The two sorted datasets are **merged efficiently**.



User Table

Id	Name
2	Json
1	Kyle
4	Reid
3	Robert

Product Table

Product Id	User Id
101	2
102	4
103	4
104	1

Sorting on Joining Key

Id	Name
1	Kyle
2	Json
3	Robert
4	Reid

Product Id	User Id
104	1
101	2
102	4
103	4

Merge

Example:

```
>>> df1=spark.range(1,10000000)
>>> df2=spark.range(1,100000)
>>> joined=df1.join(df2,"id")
>>> joined.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [id#8L]
   +- BroadcastHashJoin [id#8L], [id#10L], Inner, BuildRight, false
      :- Range (1, 10000000, step=1, splits=8)
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]),false), [plan_id=56]
         +- Range (1, 100000, step=1, splits=8)


>>> joined=df1.hint('merge').join(df2,"id")
>>> joined.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [id#8L]
   +- SortMergeJoin [id#8L], [id#10L], Inner
      :- Sort [id#8L ASC NULLS FIRST], false, 0
      :  +- Exchange hashpartitioning(id#8L, 200), ENSURE_REQUIREMENTS, [plan_id=76]
      :     +- Range (1, 10000000, step=1, splits=8)
      +- Sort [id#10L ASC NULLS FIRST], false, 0
         +- Exchange hashpartitioning(id#10L, 200), ENSURE_REQUIREMENTS, [plan_id=77]
            +- Range (1, 100000, step=1, splits=8)
```

4. Cartesian Product Join

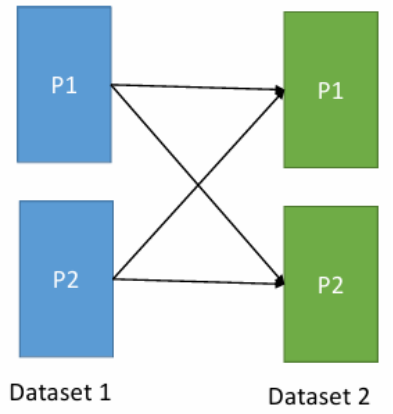
A **Cartesian Product Join** (also called **Cross Join**) **pairs every record from one dataset** with **every record from another dataset**.
It is the **most expensive type of join** because it creates **M × N combinations**, where:

- **M = Number of records in Dataset 1**
- **N = Number of records in Dataset 2**

This can cause **Out of Memory (OOM) errors** because of **massive data explosion**.

Steps in Cartesian Product Join

- **Step 1**: Read both datasets.
- **Step 2**: **Shuffle occurs** – each partition from **Dataset 1** is sent to all partitions of **Dataset 2**.
- **Step 3**: **Nested loop join** is performed, pairing **each row** from **Dataset 1** with **each row** from **Dataset 2**.



5. Broadcast Nested Loop Join

This join is used when:

- The **join condition is not equality-based** (e.g., `<, >, <=, >=`).
- The datasets are **small enough** for broadcasting.
- We want to **avoid shuffle**, but **performance is still slow** due to nested looping.

Steps in Broadcast Nested Loop Join

Step 1: Broadcast the Smaller Dataset

- Spark **copies** (broadcasts) the **smaller dataset** to **all executors** that are processing the **bigger dataset**.
- This helps **avoid expensive shuffle operations**.

Step 2: Nested Loop Join

- Every row from **Dataset 1** is **compared with every row** from **Dataset 2** using a **nested loop**.
- Since this join is used for **non-equi conditions** (`<, >, <=, >=`), it **does not stop** after finding a match.
- Instead, it **iterates over all records** in the other dataset.

Example:

```
>>> spark.conf.set("spark.sql.crossJoin.enabled",True)
>>> df1=spark.range(1,1000)
>>> df2=spark.range(1,100)
>>> joined=df1.join(df2)
>>> joined.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- BroadcastNestedLoopJoin BuildRight, Inner
   :- Range (1, 1000, step=1, splits=8)
   +- BroadcastExchange IdentityBroadcastMode, [plan_id=94]
      +- Range (1, 100, step=1, splits=8)
```

How Spark Decides which Join to Use?

	Cartesian Join	Broadcast Hash Join	Sort merge Join	Shuffle Hash Join	Broadcast Nested Loop Join
Inner-Join	✓	✓	✓	✓	✓
Left-Join	✗	✓	✓	✓	✓
Right-Join	✗	✓	✓	✓	✓
Cross-Join	✗	✓	✓	✓	✓
Left-Semi_Join	✗	✓	✓	✓	✓
Left-Anti-Semi-Join	✗	✓	✓	✓	✓
Outer-Join	✗	✗	✓	✓	✓
Support Equi Join	✓	✓	✓	✓	✓
Support Non Equi Join	✓	✗	✗	✗	✓
Require Sortable Join key	✗	✗	✓	✗	✗

Join Condition	If Hints Are Given	If No Hints Are Given
Equality (=) Joins	**Use Hints (`BROADCAST`, `MERGE`, `SHUFFLE_HASH`)**	**Broadcast Hash Join** if one side is small enough to broadcast.
		Shuffle Hash Join if one side is much smaller than the other.
		Sort Merge Join if join keys are sortable.
		Cartesian Product Join (only if inner join and no other option).
		Broadcast Nested Loop Join (last resort, can cause OOM).

Non-Equality (`<, >, <=, >=`) Joins	**Use Hints (`BROADCAST`, `SHUFFLE_REPLICATE_NL`)**	**Broadcast Nested Loop Join** if one side is small enough to broadcast.
		Cartesian Product Join (only if inner join and no other option).
		Broadcast Nested Loop Join (last resort, can cause OOM).

Driver Configurations:

What is the Driver in Spark?

- The **Driver** is like the **brain** of a Spark application.
- It is responsible for coordinating tasks and managing the Spark job.

Why is Driver Memory Important?

- When you use functions like `collect()` or `take()`, the data from Executors (worker nodes) is brought back to the Driver.
- If the dataset is too big, the Driver may **run out of memory** and **crash** (called **Out of Memory (OOM) error**).

Why Don't We Tune the Driver Often?

- Most of the work in Spark happens in **Executors** (worker nodes), not the Driver.
- So, we usually don't need to optimize the Driver much.
- But, **if too much data is collected to the Driver, it may fail.**

How to Prevent Driver from Crashing?

- We should **avoid using `collect()` or `take()` on large datasets**.
- We can **increase the Driver memory** to handle more data safely.

Spark-submit Options for Driver

--driver-memory

  - Controls how much memory is given to the Driver.
  - Example:
    - `--driver-memory 2G` (Gives **2GB** memory to the Driver)
    - `--driver-memory 1000M` (Gives **1000MB** memory)
  - Default is **1024MB (1GB)**

--driver-cores

- Controls how many CPU cores the Driver can use (**only in cluster mode**).
- Default is **1 core**.
- Usually not needed unless you want the Driver to do computations in parallel.

`spark.driver.memory` (Default: 1024MB or 1GB)

- This decides how much memory is allocated to the **Driver**.
- If the Driver needs more memory (e.g., handling large datasets), increase this value.
- Example:
  - `spark.driver.memory = 2G` (Allocates **2GB** memory to the Driver)

`spark.driver.cores` (Default: 1 core)

- Defines how many CPU cores the **Driver** can use.
- More cores allow the Driver to process tasks faster (useful for local computations).
- Example:
  - `spark.driver.cores = 2` (Driver can use **2 CPU cores**)

`spark.driver.maxResultSize`

- Limits the **amount of data** the Driver can collect from Executors at once (in bytes).
- If a Spark action (like `collect()`) **exceeds this limit**, the job will **fail**.
- Default: At least **1MB**, or **0 for unlimited** (not recommended).
- **Why is this important?**
  - If this limit is **too high**, the Driver might run **out of memory (OOM error)**.
  - If this limit is **too low**, jobs might fail even if the Driver has enough memory.
- Example:
  - `spark.driver.maxResultSize = 1G` (Driver can collect up to **1GB** of data)

`spark.driver.memoryOverhead` (Default: 10% of `driverMemory`, minimum 384MB)

- Extra memory for **non-heap** tasks like:
  - JVM overhead
  - Native system processes
  - Interned strings (strings stored in memory for efficiency)
- If you set `spark.driver.memory = 2G`, the **default overhead** would be **200MB (10%)**.
- **Why is this needed?**
  - If overhead memory is too **low**, the Driver might crash due to extra system tasks.
  - If overhead memory is too **high**, less memory is available for computations.
- Example:
  - `spark.driver.memoryOverhead = 500M` (Allocates **500MB** extra memory)

How a Spark Job is Executed

1. **Submitting a Job**:
    - The **client** submits a Spark job.
    - The **Driver** takes the job and starts processing.
2. **Driver Converts the Job into Tasks**:
    - The **Driver** runs the `main()` method.
    - It **divides** the job into small tasks using a **DAG Scheduler**.
3. **Cluster Manager Assigns Resources**:
    - The **Driver** asks the **Cluster Manager** for **Executors**.
    - Executors run on **Worker Nodes**.
4. **Executors Execute Tasks**:
    - The **Executors** process tasks and send results back to the **Driver**.
    - If a **Worker Node** crashes, Spark automatically **reschedules** tasks to another Executor.
5. **Job Completion**:
    - The **Driver** gets the final result.
    - Spark **stops** execution and releases resources.

What are Executors and Cores?

Executors

- Executors **run tasks** in Spark.
- Created on **Worker Nodes** by the **Cluster Manager**.
- Each **Executor** has its own **JVM (Java Virtual Machine)** process.
- Executors run for the **entire lifetime** of the Spark job.

Cores (CPU Cores in Executors)

- Each **Executor** has **multiple CPU cores**.
- Each **core** runs **one task at a time**.
- More cores = **better parallelism** (faster processing).

Best Practice

- **Fewer executors, more cores per executor → Efficient parallelism**
- Too many executors → **Wastes memory and increases overhead**
- Too few executors → **Under utilizes CPU resources**

Key Executor Configurations

1. `spark.executor.memory` (Executor Memory)

- Memory assigned to **each executor**.
- **Default**: 1GB
- **Example**:
  - `spark.executor.memory = 4G` (Each executor gets **4GB** memory)

2. `spark.executor.cores` (Executor Cores)

- Number of **CPU cores per executor**.
- **Default**: 2
- **Example**:
  - `spark.executor.cores = 4` (Each executor gets **4 CPU cores**)

3. `spark.executor.instances` (Number of Executors)

- How many **executors** should be launched.
- **Default**: 2
- **Example**:
  - `spark.executor.instances = 5` (Spark launches **5 Executors**)

4. `spark.executor.memoryOverhead` (Extra Memory for Executors)

- Extra **10% of executor memory** for **system tasks**.
- **Minimum**: 384MB
- **Example Calculation**:
  - If `spark.executor.memory = 16G`, overhead = **1.6GB** (10%)
  - If `spark.executor.memory = 2G`, overhead = **384MB** (since 384MB is higher than 10%)

How Executors Work in Spark Cluster Modes

1. Cluster Mode

- The **Driver** runs on **one of the Worker Nodes**.
- The **Client** submits the job and **disconnects** after submission.
- Best for **large-scale production jobs**.

2. Client Mode

- The **Driver** runs **locally** where the job is submitted.
- The **Client** must stay **connected** until the job finishes.
- Best for **interactive debugging** and development.

When running a Spark job, we need to **optimize the number of executors and cores** to achieve the best performance. Let's compare two configurations to see why one is inefficient and the other is **well-balanced**.

First Configuration (NOT GOOD)

Cluster Details

- **10 Nodes**
- **16 Cores per Node**
- **64GB RAM per Node**
- **Spark Job Configuration: 1 core per executor**

Calculations:

- **Total Cores in Cluster** = 16 cores/node × 10 nodes = **160 cores**
- **Executors in Cluster** = 1 core per executor → **160 executors**
- **Executors per Node** = 160 executors / 10 nodes = **16 executors per node**
- **Memory per Executor** = 64GB / 16 executors = **4GB per executor**

Problems with this Setup:

1. **Too Many Executors:**
    - **160 executors** means **160 JVM processes**, leading to **high overhead**.
    - JVM processes consume memory and slow down execution.
2. **Less Parallelism within Executors:**
    - Each executor has only **1 core** → cannot process multiple tasks efficiently.
    - Multiple cores per executor allow better resource utilization.
3. **Memory Waste & Overhead Issues:**
    - **Each executor has only 4GB RAM**, and **10% JVM overhead** wastes a lot of memory.
    - **YARN Daemon & Application Manager need memory** but are not given enough.

Conclusion: NOT RECOMMENDED

- **Too many executors**
- **High JVM overhead**
- **Inefficient resource utilization**

Second Configuration (GOOD & BALANCED)

Cluster Details

- **10 Nodes**
- **16 Cores per Node**
- **64GB RAM per Node**
- **Spark Job Configuration: 5 cores per executor (`--executor-cores=5`)**

Calculations:

- **Leave 1 Core per Node for YARN Daemons** → Available Cores per Node = **16 - 1 = 15**
- **Total Cores in Cluster** = 15 cores/node × 10 nodes = **150 cores**
- **Executors in Cluster** = 150 cores / **5 cores per executor** = **30 executors**
- **Leave 1 Executor for YARN Application Manager** → **29 executors available**
- **Executors per Node** = 30 executors / 10 nodes = **3 executors per node**
- **Memory per Executor** = 64GB RAM / 3 executors = **21.3GB per executor**
- **JVM Heap Overhead (~10%)** = **2.13GB**
- **Actual Usable Memory per Executor** = **21.3GB - 2.13GB = 19.7GB**

Why This Setup is Better?

1. **Balanced Parallelism:**
    - Each executor has **5 cores**, allowing better **task parallelism**.
    - More tasks run **within the same JVM**, reducing **overhead**.
2. **Efficient Memory Usage:**
    - Only **29 executors** instead of **160**, reducing JVM **overhead**.
    - More memory (19.7GB per executor) ensures tasks run **smoothly**.
3. **YARN Resources Managed Well:**
    - **Leaves 1 core and 1 executor for YARN** to avoid system slowdowns.
    - Ensures smooth operation without overloading the cluster.

Parallelism Configurations

In Apache Spark, **shuffle operations** involve **redistributing data** across partitions, which can impact performance. To control the number of partitions created during shuffling, we use two settings:

1. **`spark.default.parallelism`** → Used for **RDDs**
2. **`spark.sql.shuffle.partitions`** → Used for **DataFrames**

`spark.default.parallelism` (For RDDs)

- **Applies only to RDDs (Resilient Distributed Datasets).**
- Default value = **Total number of cores in the cluster**
- **Affects transformations that involve shuffling**, like:
    - `reduceByKey()`
    - `groupByKey()`
    - `join()`

```
>>> rdd=sc.parallelize(range(1000))
>>> rdd.getNumPartitions()
8
```

Why is this important?

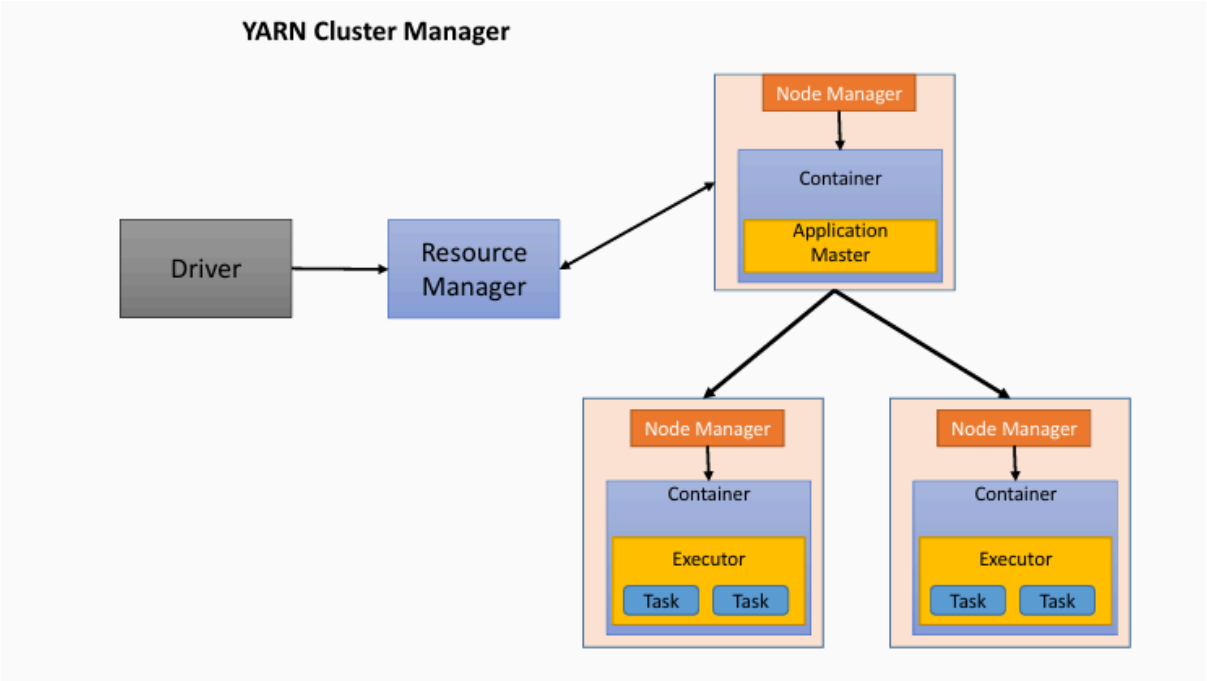
- If the number of partitions is too **low**, some partitions may become **too large**, leading to **slower performance**.
- If too **high**, there may be **too many small partitions**, causing **unnecessary overhead**.

`spark.sql.shuffle.partitions` (For DataFrames)

- **Applies only to DataFrames.**
- Default value = **200 partitions**
- **Affects transformations that involve shuffling**, like:
    - `groupBy()`
    - `join()`

```
>>> data = [('Ram',1),('Raj',2),('Ram',1),('Joann',4),('Raj',2),('Robert',5),('Reid',6),('Sam',7)]
>>> df = spark.createDataFrame(data, schema=('name','id'))
>>> df.rdd.getNumPartitions()
8
>>> df1 = df.groupBy("name").count()
>>> df1.rdd.getNumPartitions()
1Stage 0:>                                                          (0 + 8) / 8]
>>> df1.rdd.getNumPartitions()
1
```

Memory Management



YARN (**Yet Another Resource Negotiator**) is a **resource manager** used in Hadoop 2.x. It helps **allocate resources** (CPU & memory) and **schedule tasks** efficiently in a Spark cluster.

Key Components of YARN

Resource Manager (RM) → Brain of YARN

- Controls all cluster resources.
- Allocates resources to applications.

Node Manager (NM) → Worker of YARN

- Runs on each node in the cluster.
- Manages resources (CPU, Memory) on that node.
- Launches and monitors **containers** (where tasks run).

Application Master (AM) → Coordinator for a Spark job

- Launched for every application (job).
- Requests resources from the **Resource Manager**.
- Manages job execution.

How Spark Works with YARN (Step-by-Step Flow)

Step 1: User submits a Spark job

- The **Spark driver** starts and creates **SparkContext**.

Step 2: Driver requests resources from YARN

- The driver contacts the **YARN Resource Manager** to ask for resources.

Step 3: YARN Resource Manager finds a Node Manager

- The **Node Manager** starts an **Application Master** (AM) in a container.

Step 4: Application Master registers with Resource Manager

- The AM informs the **Resource Manager** that it's ready.

Step 5: Application Master requests executor containers

- The AM asks for more containers from the Resource Manager.

Step 6: Executors are launched

- The **Node Manager** starts containers and **executors** inside them.

Step 7: Driver communicates with executors

- The **Driver** assigns tasks to executors and coordinates job execution.

Step 8: Job completes & Application Master unregisters

- When all tasks are done, the **AM unregisters** from the **Resource Manager**.

Why is YARN Important for Spark?

- **Efficient Resource Sharing** → Multiple Spark jobs can run together.
- **Fault Tolerance** → If a node fails, YARN can recover jobs.
- **Scalability** → Can manage large-scale data processing efficiently.

YARN allows Spark to run in a distributed environment efficiently by managing CPU, memory, and job execution.

What is Off-Heap Memory?

Off-Heap Memory is memory outside the JVM (Java Virtual Machine), but it can still be used by Spark.

Why Use Off-Heap Memory?

- Reduces Garbage Collection (GC) overhead since it’s not managed by JVM.

- Useful for big data applications that need efficient memory management.
- Stores serialized DataFrames and RDDs.
- Must be explicitly enabled (disabled by default).

Overhead Memory in Spark

Overhead Memory is extra memory used for system-related operations.

Default Overhead Memory

- **10% of executor memory** (minimum **384MB**).
- Used for **VM operations, interned strings, and native overheads**.

How Spark Allocates Memory in Executors

Each **Executor (JVM process)** divides its memory into:

- **Heap Memory** (Managed by JVM)
- **Off-Heap Memory** (Optional, must be enabled)
- **Overhead Memory** (For system operations)

Heap Memory (Divided into Three Sections)

Heap Memory is further split into:

- **Reserved Memory** → Fixed 300MB, used for Spark's internal operations.
- **User Memory** → Stores user-defined data structures, UDFs (User-Defined Functions).
- **Spark Memory** → Used by Spark for caching and computations.

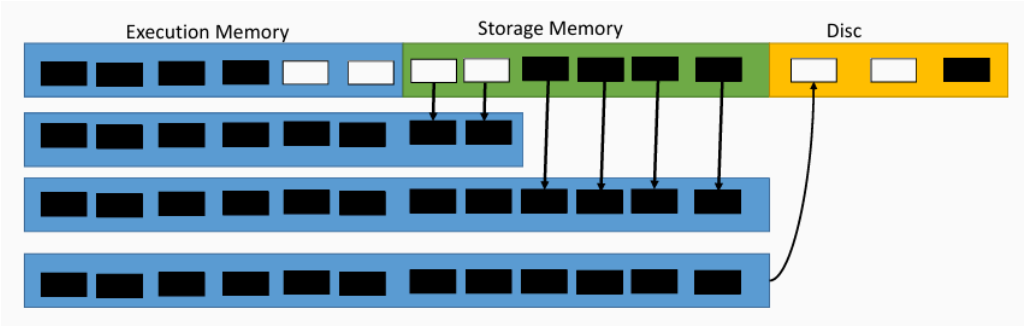
Storage Memory (Inside Spark Memory)

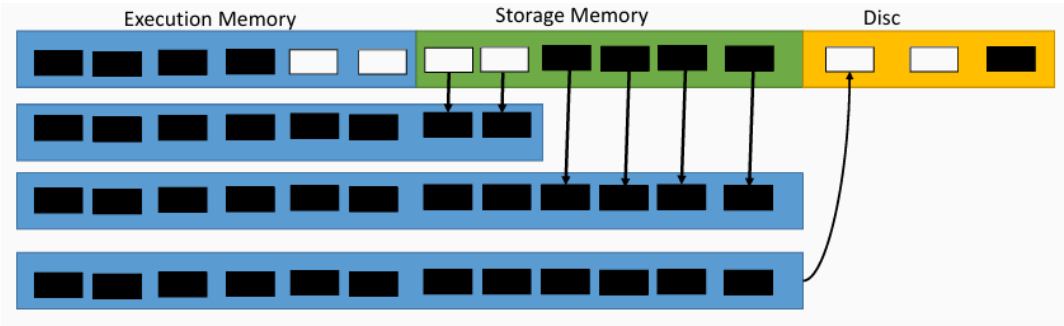
- Stores **cached data** (persisted RDDs, DataFrames).
- Uses **LRU (Least Recently Used) policy** to evict old data when needed.
- If **MEMORY_ONLY** is set, removed data must be recomputed.

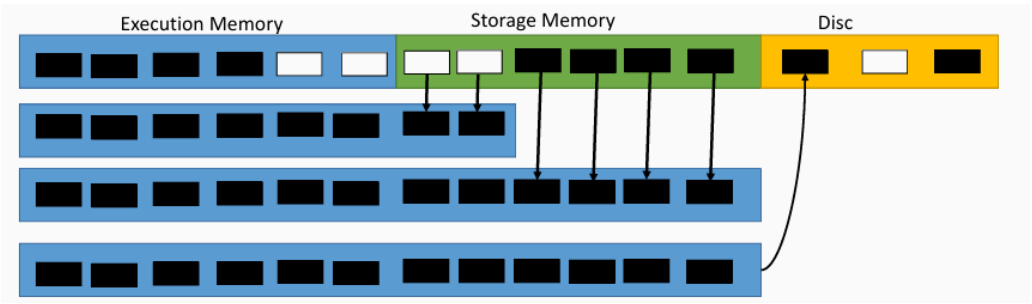
Execution Memory (Inside Spark Memory)

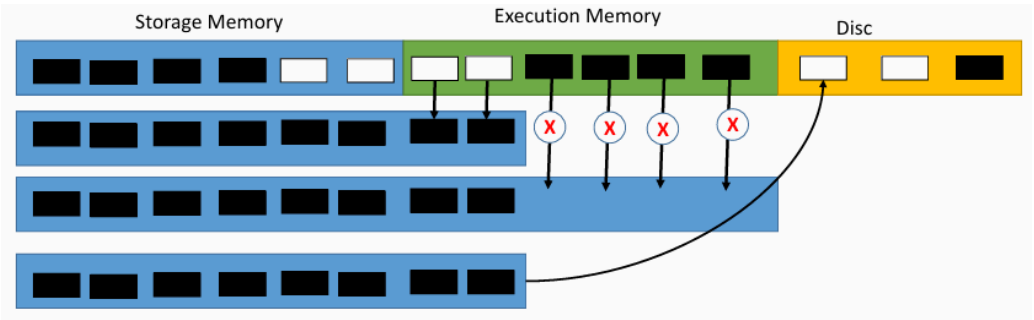
- Used for **storing intermediate results, hash tables, aggregations, etc.**
- If not enough memory is available, **data is spilled to disk**.

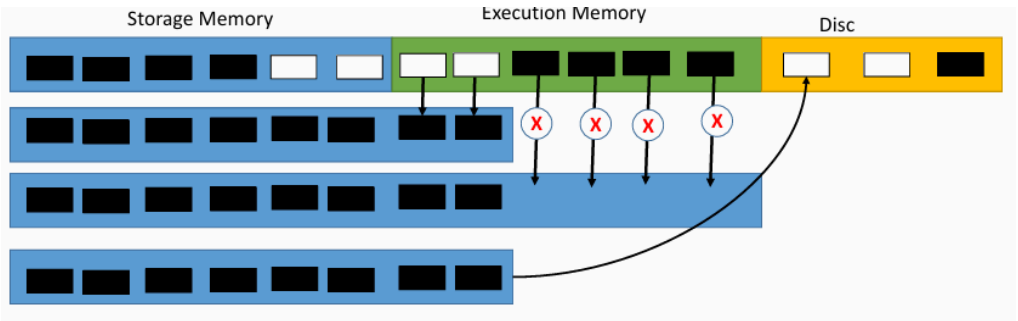
Dynamic Occupancy Mechanism:

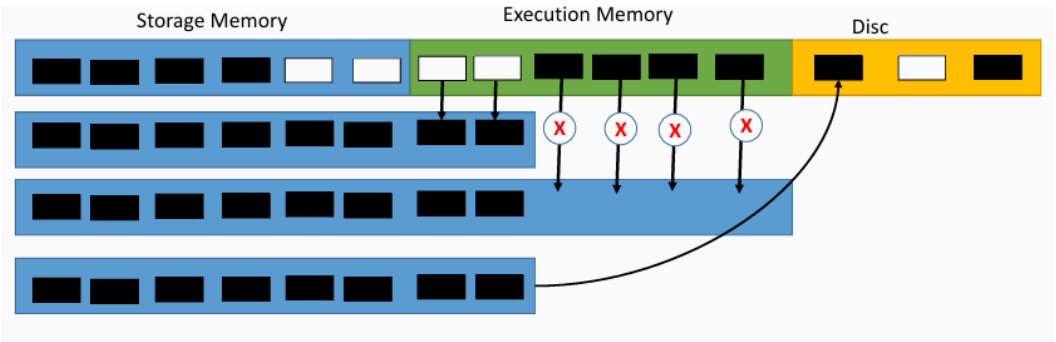











Storage Memory
Execution Memory
Disc
X
X
X
X

Cheatsheet

Basic PySpark Commands

- **Find Variable in PyCharm**: Ctrl+F
- **Recent Changes**: Alt + Shift + C
- **Recent Locations**: Ctrl + Shift + E
- **Variable Usage**: Ctrl + Alt + T

SparkSession Initialization

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local').appName('Test').getOrCreate()
```

RDD Creation

```
rdd = spark.sparkContext.parallelize([1,2,3])
print(rdd.first())
```

HDFS Commands

- **List all HDFS Commands**: hadoop fs or hdfs dfs
- **List usage of command**: hadoop fs -usage ls
- **Help for a command**: hadoop fs -help ls
- **Clone GitHub Repo**: git clone <repo-link>
- **Unzip File**: unzip file1.zip
- **Create Directory in HDFS**: hadoop fs -mkdir -p <dir-path>
- **Upload Multiple Files to HDFS**: hadoop fs -put datafiles/* <dir-path>
- **Remove Empty Directory**: rmdir
- **Delete Files and Directories**: rm
- **Copy Files from HDFS to Local**: hadoop fs -copyToLocal <HDFS_source> <local_destination>
- **List Directory Recursively**: hadoop fs -ls -R practice/retail_db
- **Display Paths Only**: hadoop fs -ls -C practice/retail_db
- **Sort by Size**: hadoop fs -ls -S practice/retail_db
- **Sort by Modification Time**: hadoop fs -ls -t practice/retail_db
- **Show First Few Lines of File**: hadoop fs -head <HDFS_file_path>
- **Show Last Few Lines of File**: hadoop fs -tail <HDFS_file_path>
- **View Full File**: hadoop fs -cat <HDFS_file_path>
- **Show File Statistics**: hadoop fs -stat <format> <HDFS_file_path>

Permissions in HDFS

- **Change File Permissions**: hadoop fs -chmod <mode> <HDFS_file_or_directory>

RDD Transformations

- **Map**: rdd.map(lambda x: x * 2)

- **FlatMap**: rdd.flatMap(lambda x: (x, x**2))
- **Filter**: rdd.filter(lambda x: x > 10)
- **Join**: rdd1.join(rdd2)
- **CoGroup**: rdd1.cogroup(rdd2)
- **Cartesian Product**: rdd1.cartesian(rdd2)
- **Union**: rdd1.union(rdd2)
- **Intersection**: rdd1.intersection(rdd2)
- **Distinct**: rdd.distinct()
- **Subtract**: rdd1.subtract(rdd2)
- **Sort by Key**: rdd.sortByKey()

RDD Actions

- **Reduce**: rdd.reduce(lambda a, b: a + b)
- **Count**: rdd.count()
- **Collect**: rdd.collect()
- **First Element**: rdd.first()
- **Take N Elements**: rdd.take(n)

Partitioning

- **Coalesce (Reduce Partitions)**: rdd.coalesce(n)
- **Repartition (Increase/Decrease Partitions)**: rdd.repartition(n)

Aggregation Operations

- **Group by Key**: rdd.groupByKey()
- **Reduce by Key**: rdd.reduceByKey(lambda a, b: a + b)
- **Aggregate by Key**: rdd.aggregateByKey(zeroValue, seqFunc, combFunc)
- **Count by Key**: rdd.countByKey()

DataFrame Operations

- **Create DataFrame from List**:
  df = spark.createDataFrame([(1, 'Alice'), (2, 'Bob')], ['id', 'name'])
- **Show Data**: df.show()
- **Filter Data**: df.filter(df.age > 30).show()
- **Group By and Aggregate**: df.groupBy('age').count().show()
- **Order By**: df.orderBy(df.age.desc()).show()
- **Select Specific Columns**: df.select('name', 'age').show()

SQL Query Execution:
df.createOrReplaceTempView("people")

- spark.sql("SELECT name FROM people WHERE age > 30").show()

DataFrame Persistence

- **Cache DataFrame**: df.cache()
- **Persist DataFrame**: df.persist(storageLevel)

SparkSession Commands

- **Spark Version**: spark.version
- **Range DataFrame**: spark.range(1, 10).show()
- **SQL Execution**: spark.sql("SELECT * FROM table")
- **Stop Spark**: spark.stop()

Performance Optimization

- **Explain Execution Plan**: df.explain(True)
- **Repartitioning Data**: df.repartition(n)
- **Coalescing Data**: df.coalesce(n)
- **Broadcasting Variables**: broadcastVar = sparkContext.broadcast(value)
- **Using Accumulators**: accum = sparkContext.accumulator(0)