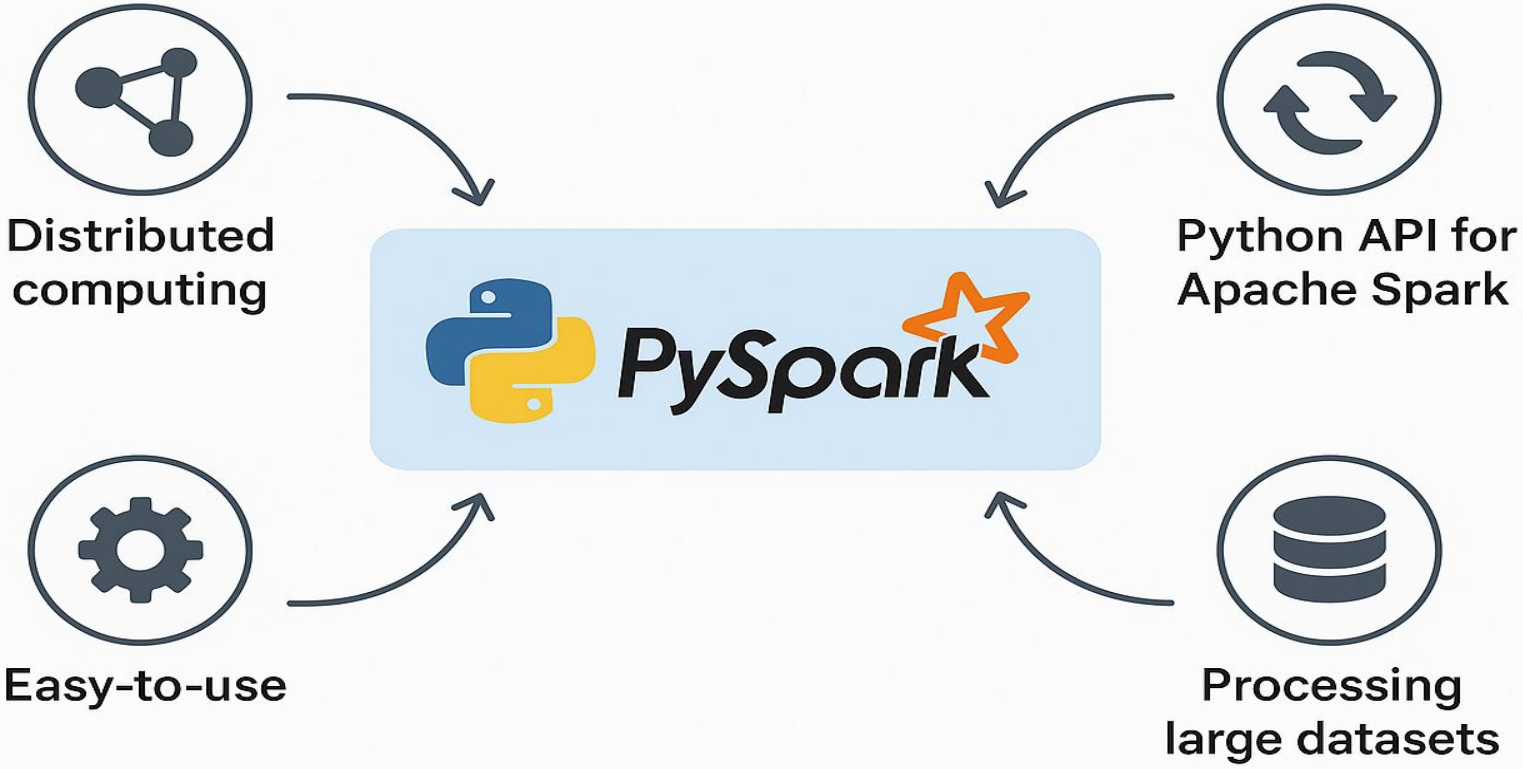
PySpark Demo

WHAT IS PYSPARK



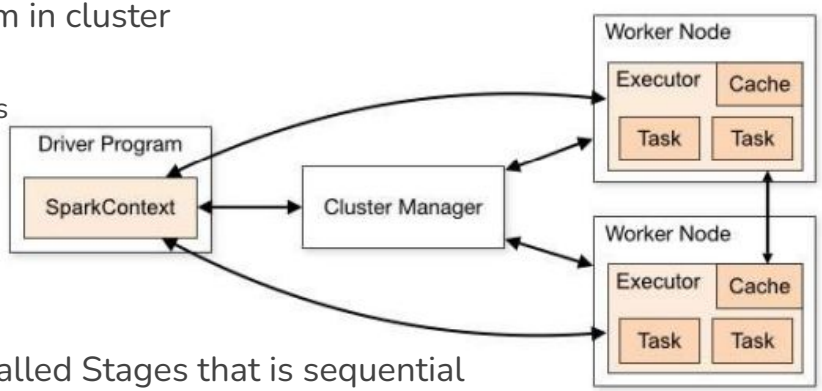
Why use PySpark

- **Python + Spark Power**: Combines the simplicity of Python with the distributed computing capabilities of Apache Spark.
- **Scalability**: Handles massive datasets effortlessly—from gigabytes to petabytes—across many machines.
- **Speed**: Executes tasks in memory, making it much faster than disk-based engines like Hadoop MapReduce.
- **Flexible APIs**: Offers both low-level (RDD) and high-level (DataFrame, SQL) APIs for data manipulation.

Feature	Apache Hadoop	Apache Spark
Processing Model	Uses MapReduce with "Map" and "Reduce" steps	Uses RDDs and a DAG execution engine
Speed	Slower, relies on disk-based storage	Faster, performs in-memory computations
Fault Tolerance	HDFS replicates data across multiple nodes	RDDs allow efficient recovery and re-computation of lost data
Ease of Use	Complex, low-level MapReduce programming model	High-level APIs in Python, Scala, and R, more developer-friendly

Spark Architecture Overview

- **Driver Program**: The process to start the execution (main() function)
- **Cluster Manager**: An external service to manage resources on the cluster (standalone manager,YARN, Apache Mesos)
- **Worker Node** : Node that run the application program in cluster
- **Executor** :
  - Process launched on a worker node, that runs the Tasks
  - Keep data in memory or disk storage
- **Task** : A unit of work that will be sent to executor
- **Job**
  - Consists multiple tasks
  - Created based on a Action
- **Stage** : Each Job is divided into smaller set of tasks called Stages that is sequential and depend on each other



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			

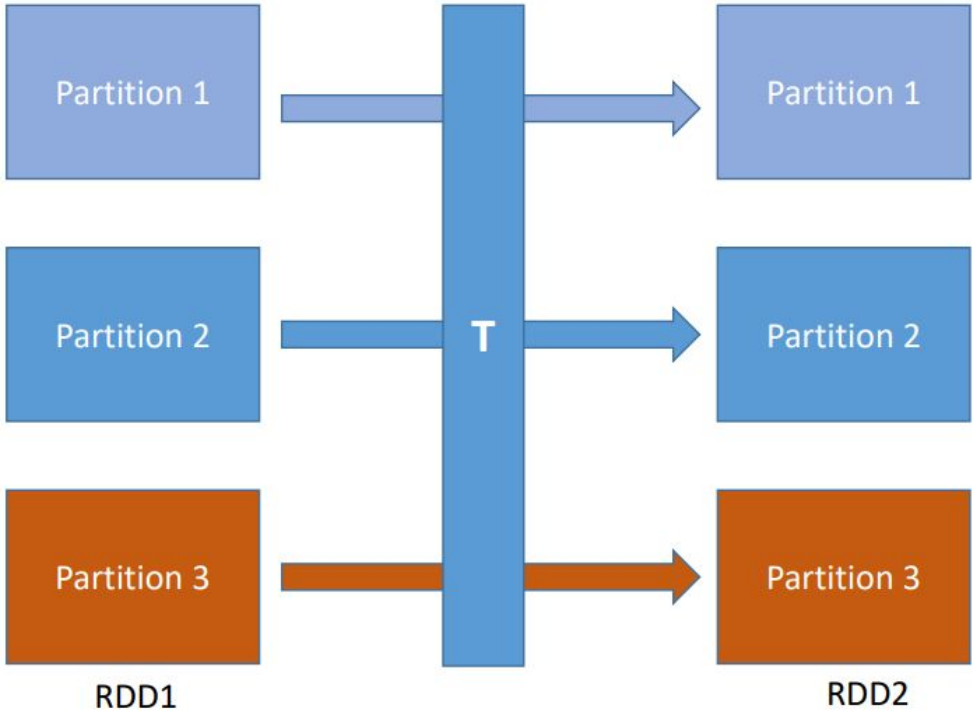
Actions

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

Narrow and Wide Transformations

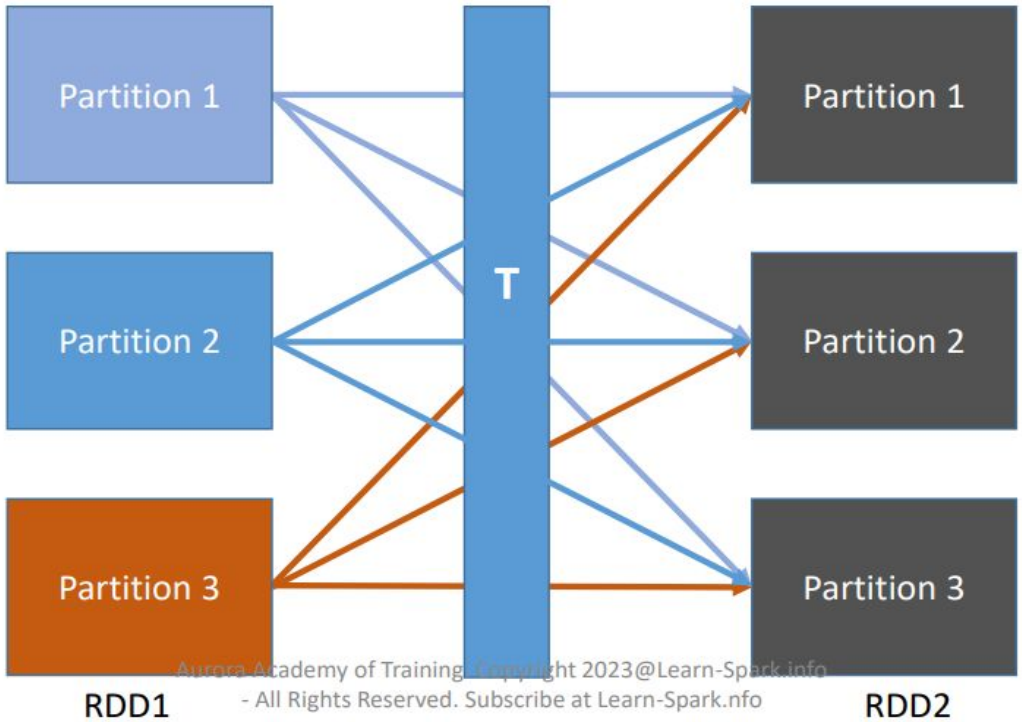
Narrow Transformations:

- These types of transformations convert each input partition to only one output partition.
- Spark merges all narrow transformations into one stage.
- Fast.
- No data shuffle.
- Ex- map(), filter()



Wide Transformations:

- This type of transformation will have input partitions contributing to many output partitions.
- Each Wide Transformations creates a new stage.
- Slow compared to Narrow.
- Data shuffle.
- Ex- groupByKey(), aggregateByKey(), join, distinct(), repartition() etc.



What is RDD

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each datasets in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

When to Use RDDs

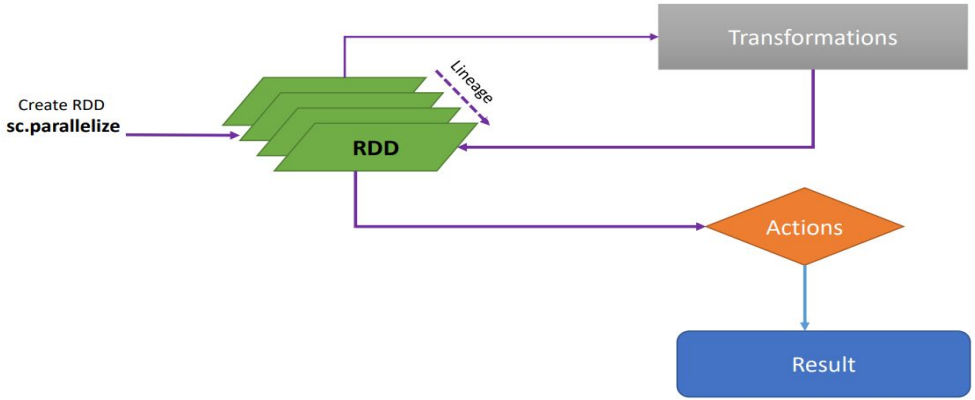
- You need **fine-grained control** over data and transformations.
- You're working with **unstructured data** or **custom processing logic**.
- You want to **manually manage partitions** or **optimize performance** at a low level.

Features of RDD

- **Fault Tolerance**

  RDDs track data lineage information to recover lost data quickly and automatically on failure at any point of execution cycle.

  Spark keeps a record of lineage while tracking the transformations that have been performed. If any part of RDD is lost, then spark will utilize this lineage record to quickly and efficiently re-compute the RDD using the identical operations.

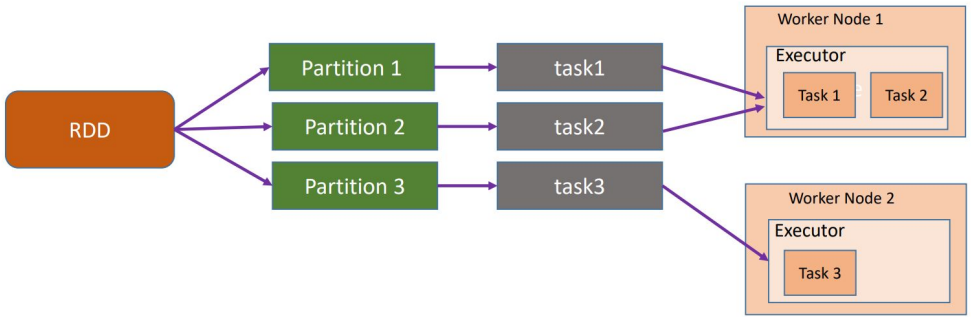


Features of RDD

- **Distributed**

  RDD will be divided into partitions while data being processed. Each partition will be processed by one task.

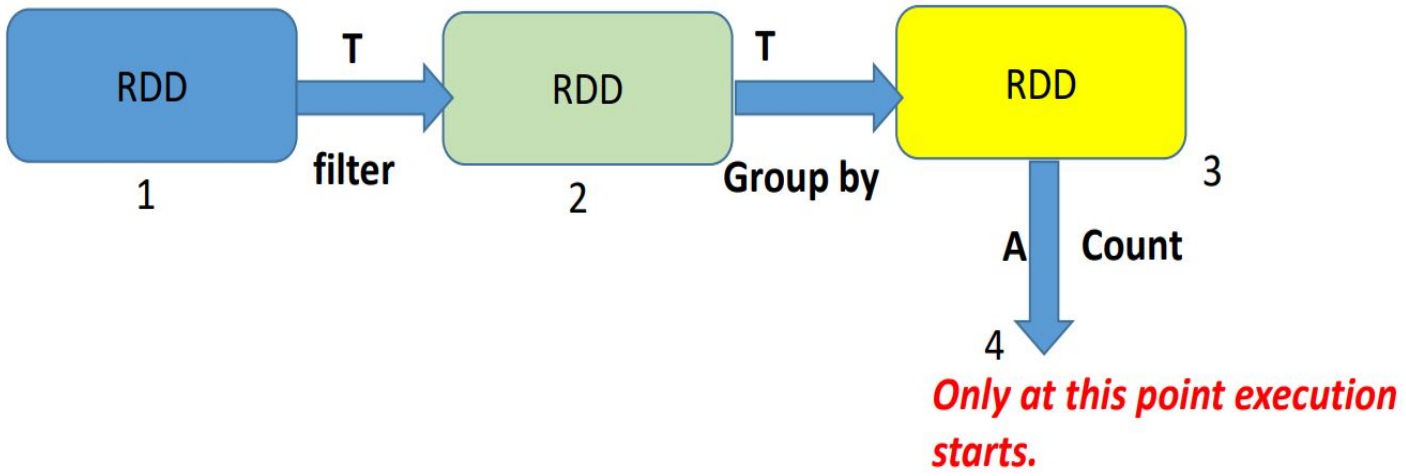
  If spark is running in YARN cluster, the number of RDD partitions is typically based on HDFS block size which is 128MB by default. We can control the number of minimum partitions by using additional arguments while invoking APIs such as textFile.



Features of RDD

- **Lazy Evaluation:**

  Each Transformation is a Lazy Operation. Evaluation is not started until an action is triggered.



Features of RDD

Example – Find out 10 sample records having a string “Robert” from a file(1TB).

With Out Lazy Evaluation:

Step 1	Load the 1TB File
Step 2	Perform full scan to find out all Records having “Robert”
Step 3	Retrieve 10 Records

With Lazy Evaluation:

Step 1	Wait for an Action and then Load the 1TB File
Step 2	Filter out first 10 records having “Robert” .

APACHE SPARK RDD VS. DATAFRAME

	RDD	DATAFRAME
DATA FORMAT	Structured or Unstructured	Structured or Semi-structured
API INTEGRATION	Any data source: Database or Text File	Specific data source: JSON, AVRO, CSV
COMPILE-TIME	Supports OOP	Does not support OOP
IMMUTABILITY	Consistent Nature	Non-regeneratable domain object

SparkContext vs SparkSession

SparkContext in PySpark

- SparkContext is the **entry point to Spark's core engine**.
- It connects the **driver program** to the **cluster manager** (e.g., YARN, Kubernetes, Mesos).
- Responsible for requesting resources and launching **executors** on worker nodes.
- All Spark jobs begin with the driver program calling **main()**, where SparkContext is initialized.

It manages:

- **RDD creation and lineage**
- **Job scheduling and task distribution**
- **Broadcast variables and accumulators**

SparkSession: Unified Entry Point (Since Spark 2.0)

Introduced in **Spark 2.0** as a single entry point for RDDs, DataFrames, SQL, Streaming, and ML.
Replaces older APIs:

- SparkContext → RDDs, Accumulators, Broadcasts
- SQLContext → DataFrames, SQL
- HiveContext → SQLContext + Hive support

- SparkSession internally creates and exposes SparkContext via spark.sparkContext.
- In Spark Shell (2.x+), spark is available by default. (sc was used in Spark 1.x.)
- Master can be yarn, mesos, Kubernetes or local(x) , x > 0
- Access SparkContext via spark.sparkContext

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .master('yarn') \
    .appName("Python Spark SQL basic example") \
    .getOrCreate()
```

Spark Execution Engine

The **Spark Execution Engine** is the **core runtime component** that takes optimized logical plans (from DAG and Catalyst) and turns them into **actual distributed execution** on a cluster.

It manages:

- **Job Scheduling**
- **Stage Creation**
- **Task Execution**
- **Resource Allocation**

Core components of the execution engine include:

- **DAG Scheduler**
- **Task Scheduler**
- **Cluster Manager**
- **Executors**

Action Triggered → Job Created

- A **job** is triggered when an **action** is called:

  Examples: .collect(), .show(), .count(), .write()
- Spark **does not execute** until an action is called (lazy evaluation).
- One **Spark job = one DAG = one execution plan**

DAG (Directed Acyclic Graph)

- A **DAG** represents the sequence of operations in a Spark job.
- Spark builds a **logical execution plan** as a DAG of transformations.

What it Means:

- **Directed** → Each operation flows in one direction (from one node to the next)
- **Acyclic** → No loops or cycles; you can't return to a previous step.
- **Graph** → Composed of:
  - **Vertices** → Represent RDDs or DataFrames
  - **Edges** → Represent transformations (e.g., map(), filter())

Steps to Build a DAG

1. User submits an application job to spark.

2. Drivers takes the application and create a Spark Context to process the application.

3. Spark Context identifies all the T and A operations present in the application.

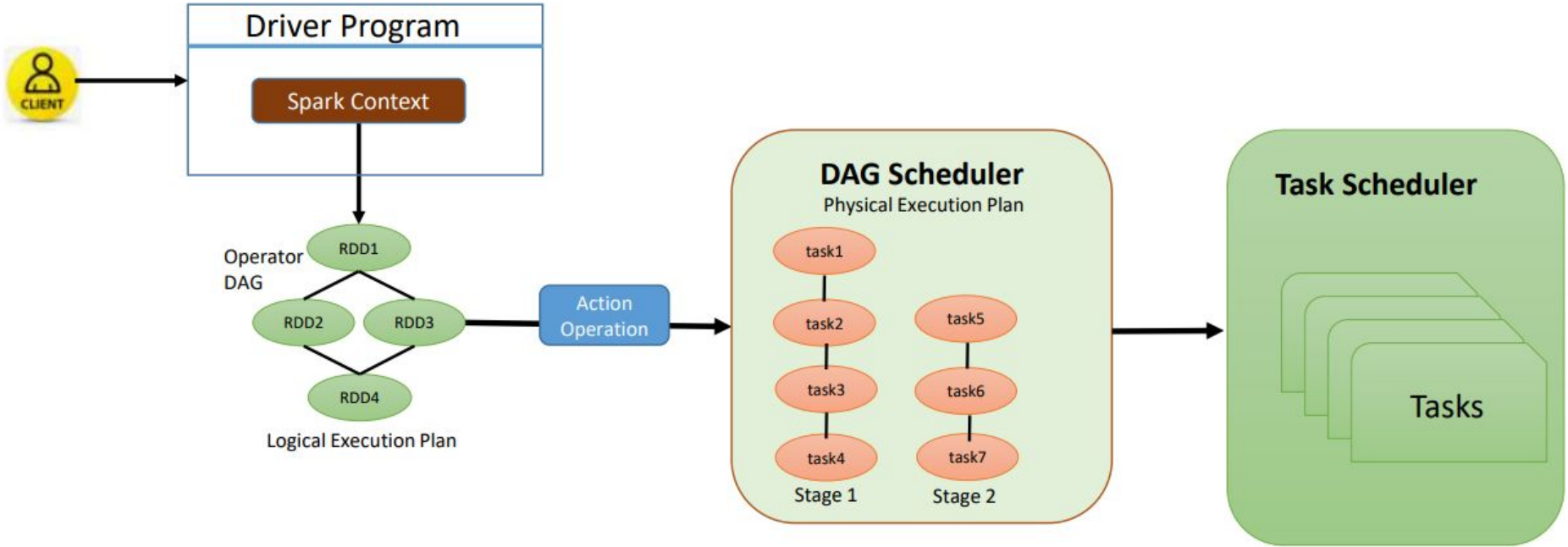
4. All the operations are arranged in a logical flow of operations called DAG (Logical Execution Plan).

5. It stops here if SC doesn't find any A Operations.

6. If it identifies an A operations, spark submit the Operator DAG to DAG scheduler.

7. DAG Scheduler converts the Logical Execution plan into Physical Execution plan and creates stages and tasks. Here Narrow T are fused together into one stage. Wide T involving shuffle process creates new stages.

8. DAG scheduler bundles all the tasks and send it Task Scheduler which then submit the job to cluster manager for execution.


Driver Program
CLIENT
Spark Context
Operator
DAG
RDD1
RDD2
RDD3
RDD4
Logical Execution Plan
Action
Operation
DAG Scheduler
Physical Execution Plan
task1
task2
task3
task4
Stage 1
task5
task6
task7
Stage 2
Task Scheduler
Tasks

Stages

- Stage is a set of operations that can be pipelined together.
- **Narrow Transformations**:
  - Example: map(), filter()
  - No data movement across partitions → same stage
- **Wide Transformations**:
  - Example: groupBy(), join()
  - Requires shuffling → new stage is created
- A job can have multiple stages depending on the logic.

Task Scheduler

- Receives ready-to-run stages from the DAG Scheduler.
- Breaks each stage into a set of independent tasks (one per data partition).
- Assigns tasks to available executors for execution.
- Makes scheduling decisions based on:
    - CPU/core availability
    - Data locality (to minimize network I/O)
    - Task retry policy (for fault tolerance)
- Stateless: It does not track lineage or data flow — only dispatches and monitors task completion.
- Optimized to achieve high throughput and parallelism with minimal overhead.

Tasks

- **Atomic unit of execution** in Spark — smallest work package.
- Each task runs the logic of its parent stage **on a single data partition**.
- All transformations within the stage (e.g., map, filter, reduce) are **executed inside the task**.
- Tasks are:
  - **Isolated** and **parallel** across executors
  - **Stateless** and **re-runnable** (supporting fault recovery)
- Execution characteristics:
  - High volume (100s or 1000s per job)
  - Fast and lightweight (compared to full jobs)
- Example:
  200 partitions → 200 parallel tasks for that stage

Cluster Manager

- Accepts resource request from Driver
- **Allocates containers / pods / processes** on worker nodes
- **Launches executors** and tracks their health
- Supported: **YARN, Kubernetes, Standalone**

Executors

- **Executors are** JVM processes running on **worker nodes.**
- **Each executor:**
  - **Executes tasks (map, filter, reduce, join, etc.)**
  - **Manages memory (caching, shuffling)**
  - **Sends results/errors back to Driver**
- **Executors live until application ends or fail**

RDD Execution Flow

User Code → RDD Operations

↓

RDD Lineage → DAG (RDD Graph)

↓

DAG Scheduler → Stages

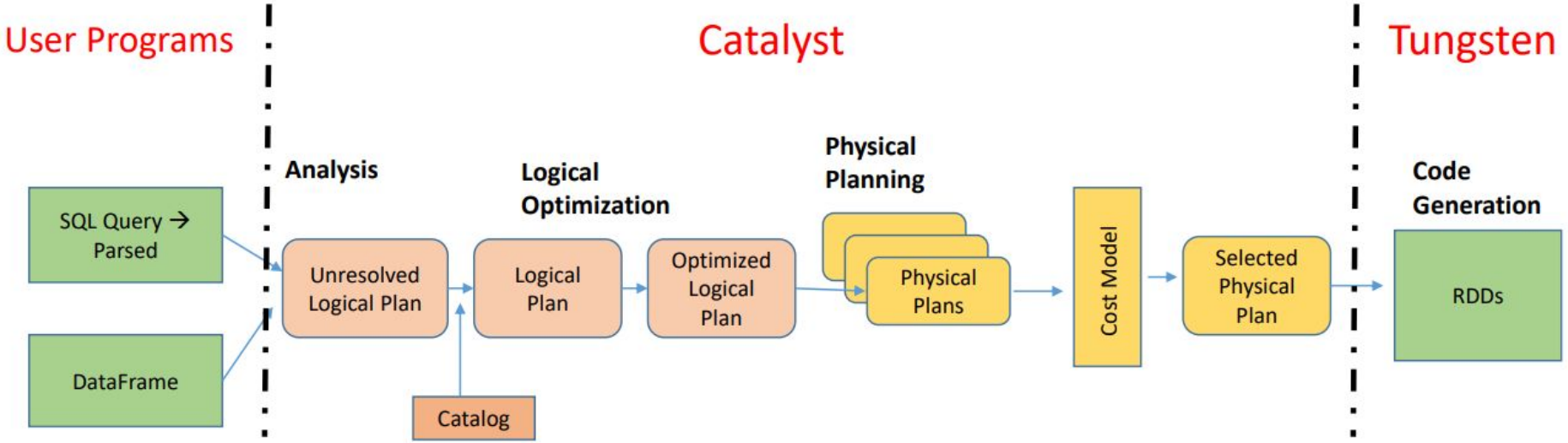
↓

Task Scheduler → Tasks

↓

Cluster Manager → Executors → Tasks run

Spark SQL Engine: Catalyst + Tungsten Pipeline



User Program

- User writes code using **SQL** or **DataFrame** APIs.
- Spark doesn't run this immediately.
- Instead, it builds a **logical plan** to understand **what** needs to be done.

Catalyst Optimizer

Catalyst is a **modular query optimizer** built on **functional programming principles (Scala)**.

Catalyst has 4 Steps:

1. **Analysis** – Understand what you wrote using Catalog (schemas)
2. **Logical Optimization** – Rewrites your plan (e.g., filter pushdown)
3. **Physical Planning** – Generates multiple execution strategies
4. **Cost Model** – Picks the most efficient one

Tungsten – The Execution Engine

Tungsten is Spark's **low-level execution engine**, introduced in Spark 1.4 and improved in 2.x+. It Runs the plan Catalyst created, but with **high performance.**

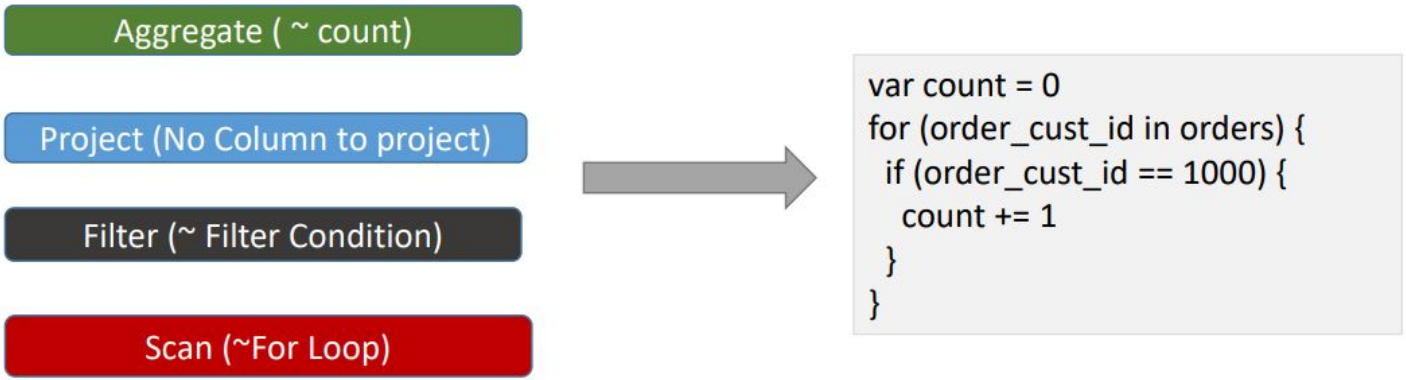
- Focused on **CPU and memory efficiency**
- Avoids overhead of older execution models (Volcano)
- Uses **whole-stage code generation** to convert stages into **Java bytecode**

Key Benefits:

- Reduces memory usage
- Runs faster by avoiding intermediate data
- Uses modern CPU features (SIMD, pipelining)

Whole-Stage Code Generation (Spark 2.x)

- Spark **compiles chains of operations into a single function**
- This makes the code run **like hand-written, optimized Java**
- Example: filter → select → groupBy is fused into one compiled unit



DataFrame / Dataset Execution Flow

User Code → DataFrame / Dataset Operations

↓

Unresolved Logical Plan

↓

Catalyst Optimizer → Optimized Logical Plan

↓

Physical Plan (with code generation)

↓

RDDs under the hood → DAG

↓

DAG Scheduler → Stages

↓

Task Scheduler → Tasks

↓

Cluster Manager → Executors → Tasks run

What is Partition

- Datasets are huge is size and they cannot fit into a single node and so they have to be partitioned across different nodes or machines.
- Partition in spark is basically an atomic chuck of data stored on a node in the cluster. They are the basic units of parallelism.
- One partition can not span over multiple machines.
- Spark automatically partitions RDDs/DataFrames and distributes the partitions across different nodes.
- We can also configure the optimal number of partitions. Having too few or many partitions is not good

How Spark does the default Partitioning of Data ?

- Spark checks the HDFS Block size. The HDFS block size for Hadoop 1.0 is 64mb and Hadoop 2.0/YARN is 128MB.
- It creates one partition for each block size.
- Ex- We have a file of 500MB, so 4 partitions would be created.
- At times programmers are required to change the number of partitions based on the requirements of the application job.
- The change can be to increase the number of partitions or decrease the number of partitions.
- So we would either apply the repartition or coalesce.

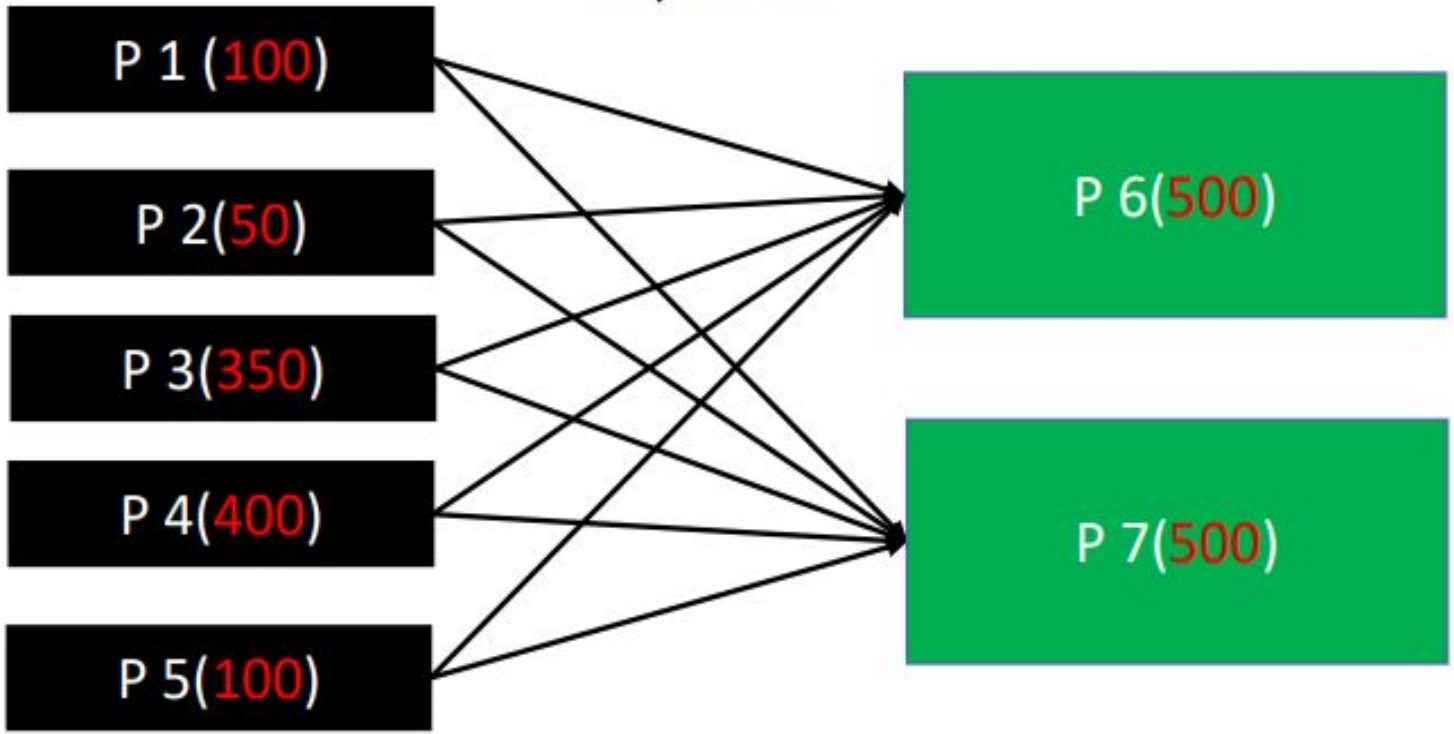
Repartition

Repartition(numPartitions):

- Return a new RDD that has exactly numPartitions partitions.
- Create almost equal-sized partitions.
- Can increase or decrease the level of parallelism.
- Spark performs better with equal-sized partitions. If you need further processing of huge data, it is preferred to have equal-sized partitions, and so we should consider using repartition.
- Internally, this uses a shuffle to redistribute data from all partitions, leading to a very expensive operation. So avoid if not required.
- If you are decreasing the number of partitions, consider using coalesce, where the movement of data shuffling across the partitions is lower

repartition(2)

Lots of Shuffling in Repartition.

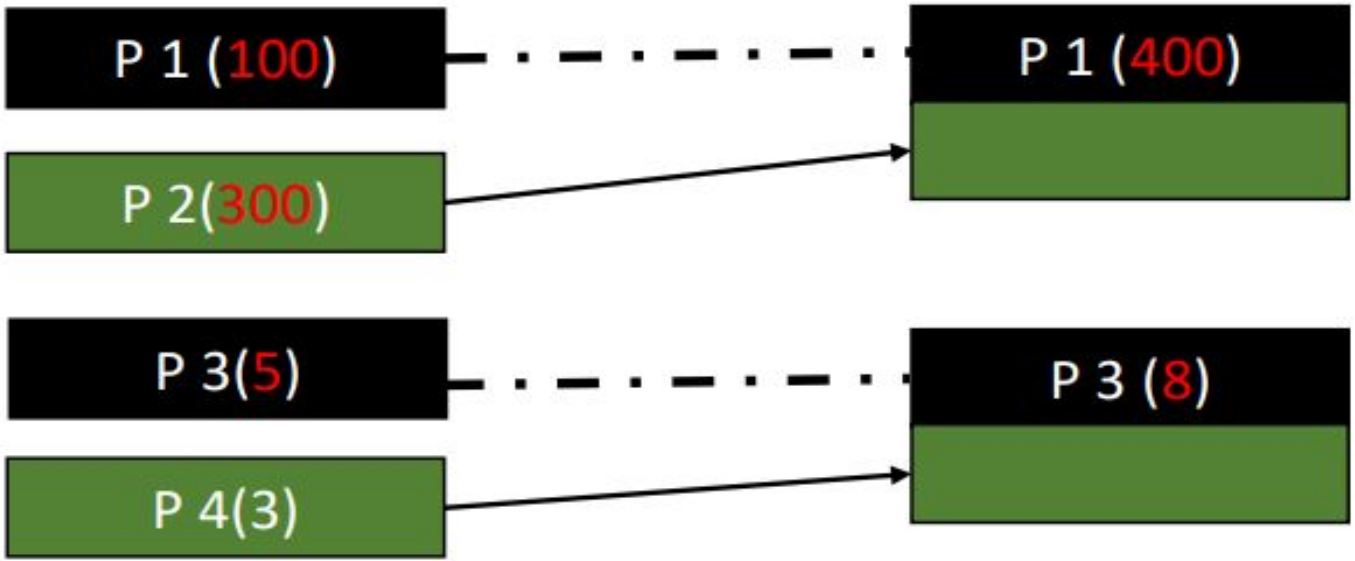


Coalesce

- coalesce(numPartitions, shuffle=False):
  - Return a new RDD that is reduced into `numPartitions` partitions.
  - Optimized version of repartition().
  - No shuffling.
  - Results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions.
  - If a larger number of partitions is requested, it will stay at the current number of partitions.
  - By Default Coalesce can be only used for decreasing the partitions.
  - But by passing shuffle=True parameter it behaves like repartition and we can increase the partitions as well.

coalesce(2)

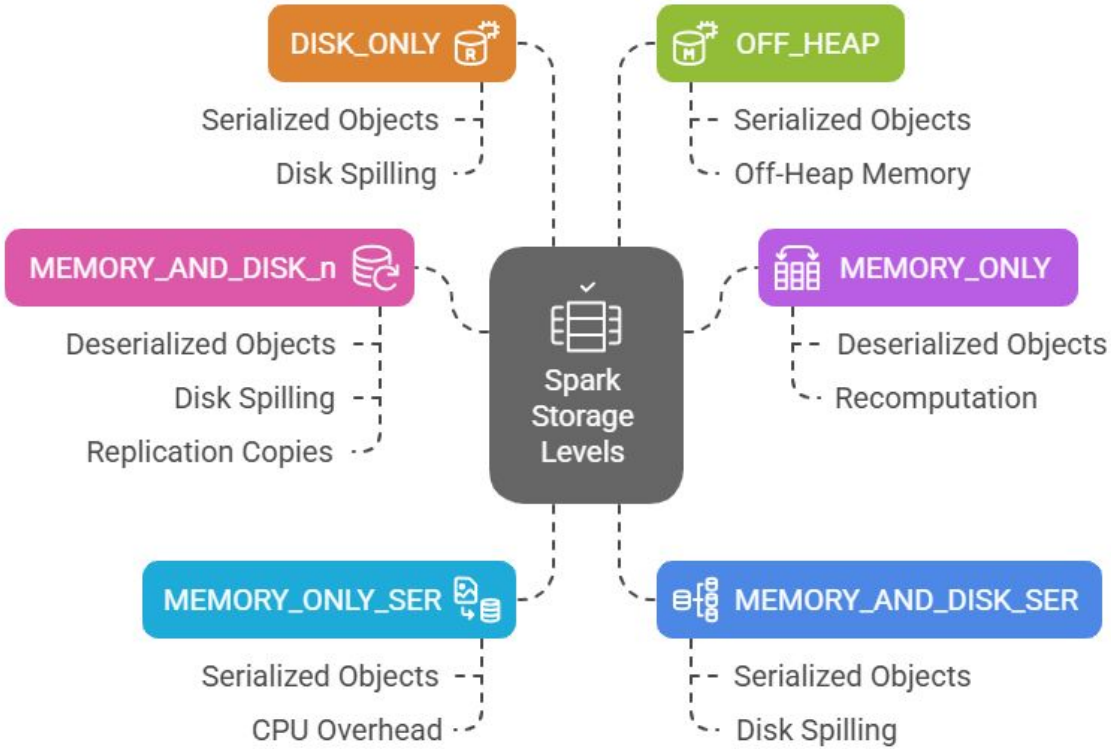
No Shuffle in coalesce.



Persistence

- One of the most important capabilities in Spark is persisting (or caching) a dataset in memory.
- When we persist an RDD, each node stores any partitions of it that it computes in memory and reuses them.
- We can mark an RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.
- Each persisted RDD can be stored using different storage level. These levels are set by passing a StorageLevel object

Spark Storage Levels: Characteristics and Trade-offs

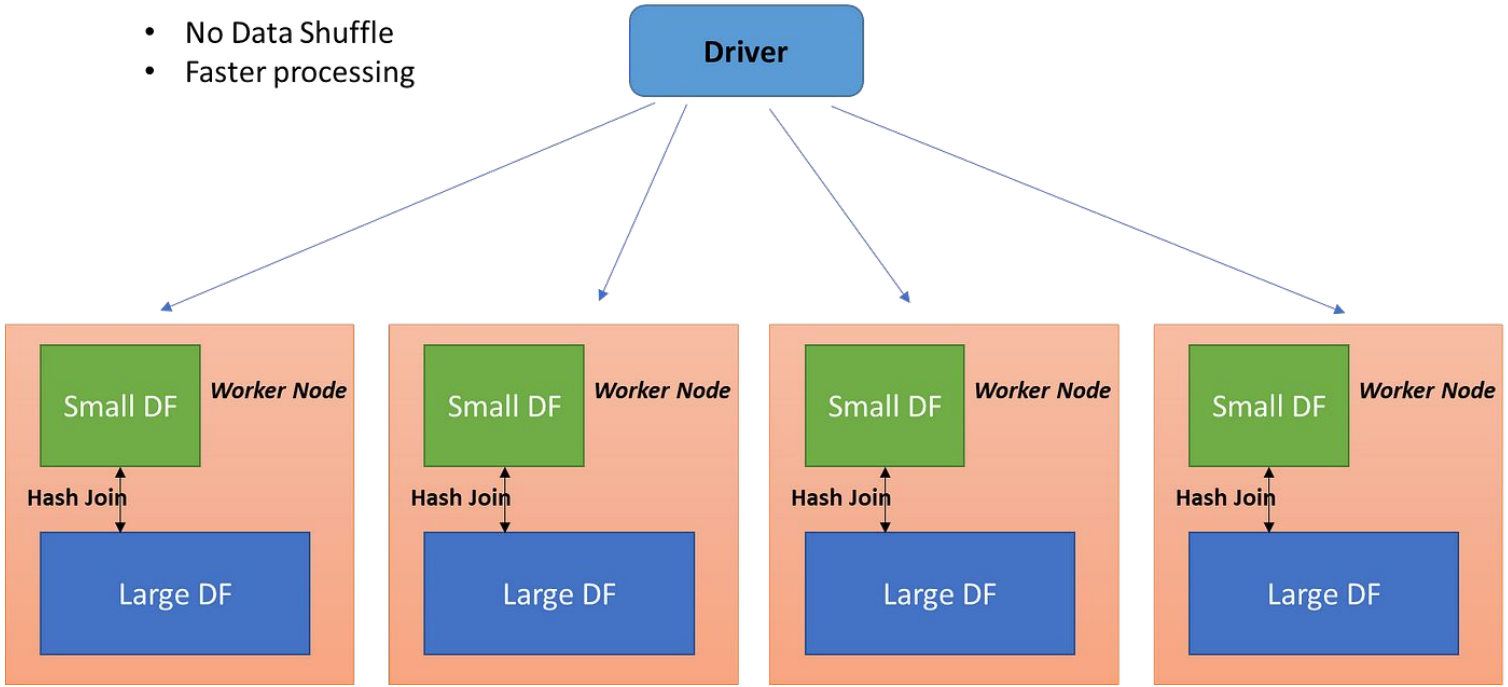


Join Strategies in PySpark

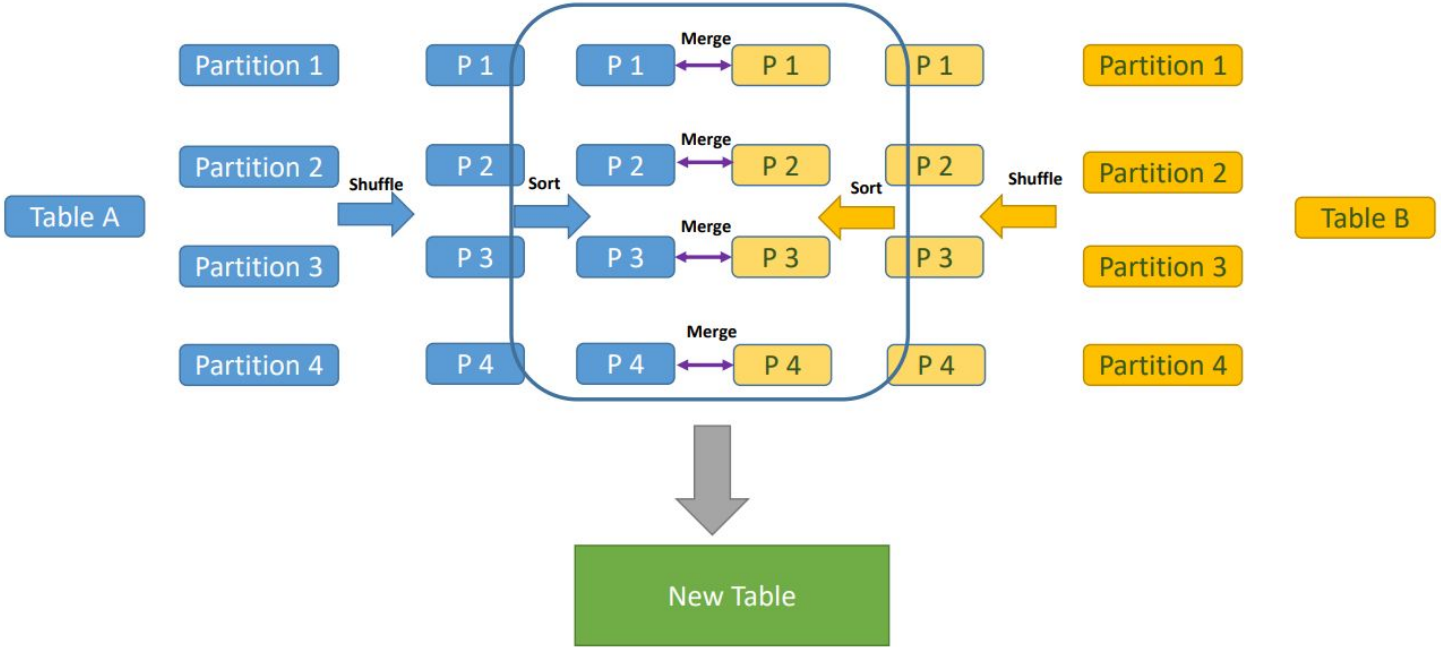
- Broadcast Hash Join
- Shuffle Sort Merge Join
- Shuffle Hash Join

Broadcast Hash Join

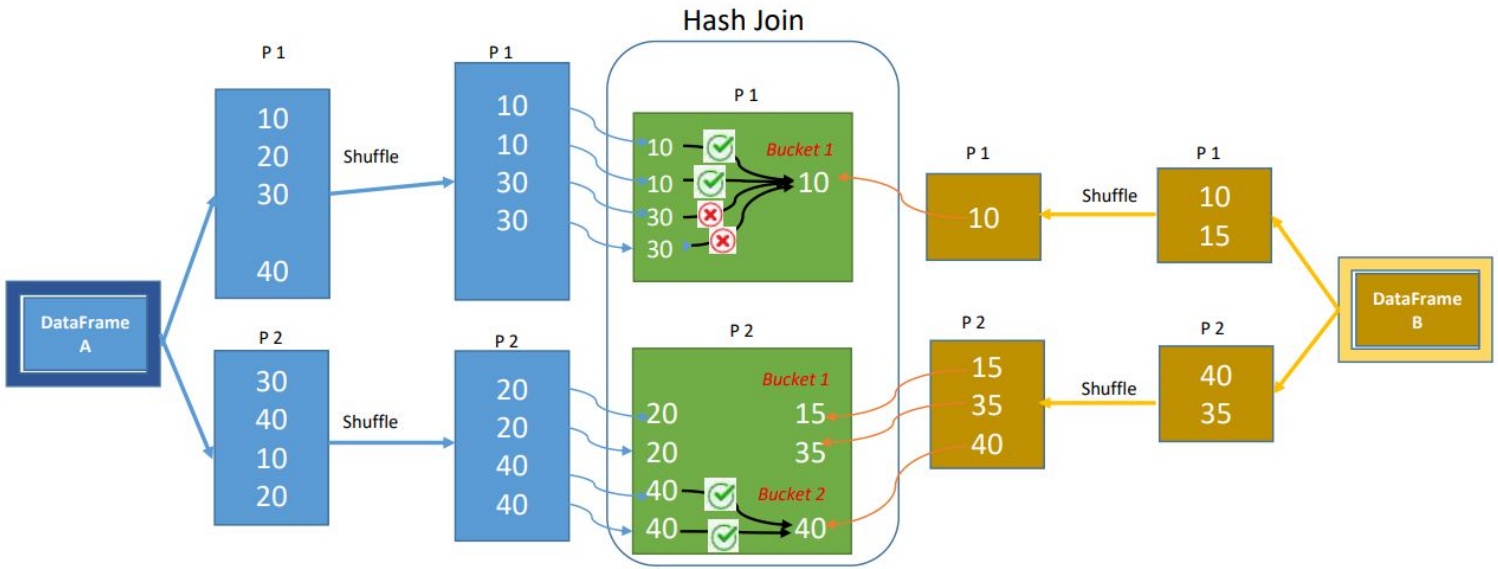
- No Data Shuffle
- Faster processing



Shuffle Sort Merge Join



Shuffle Hash Join



Dynamic Resource Allocation

Dynamic Resource Allocation in PySpark is a feature that allows Spark to automatically scale the number of executors (workers) up or down during runtime, based on the workload.

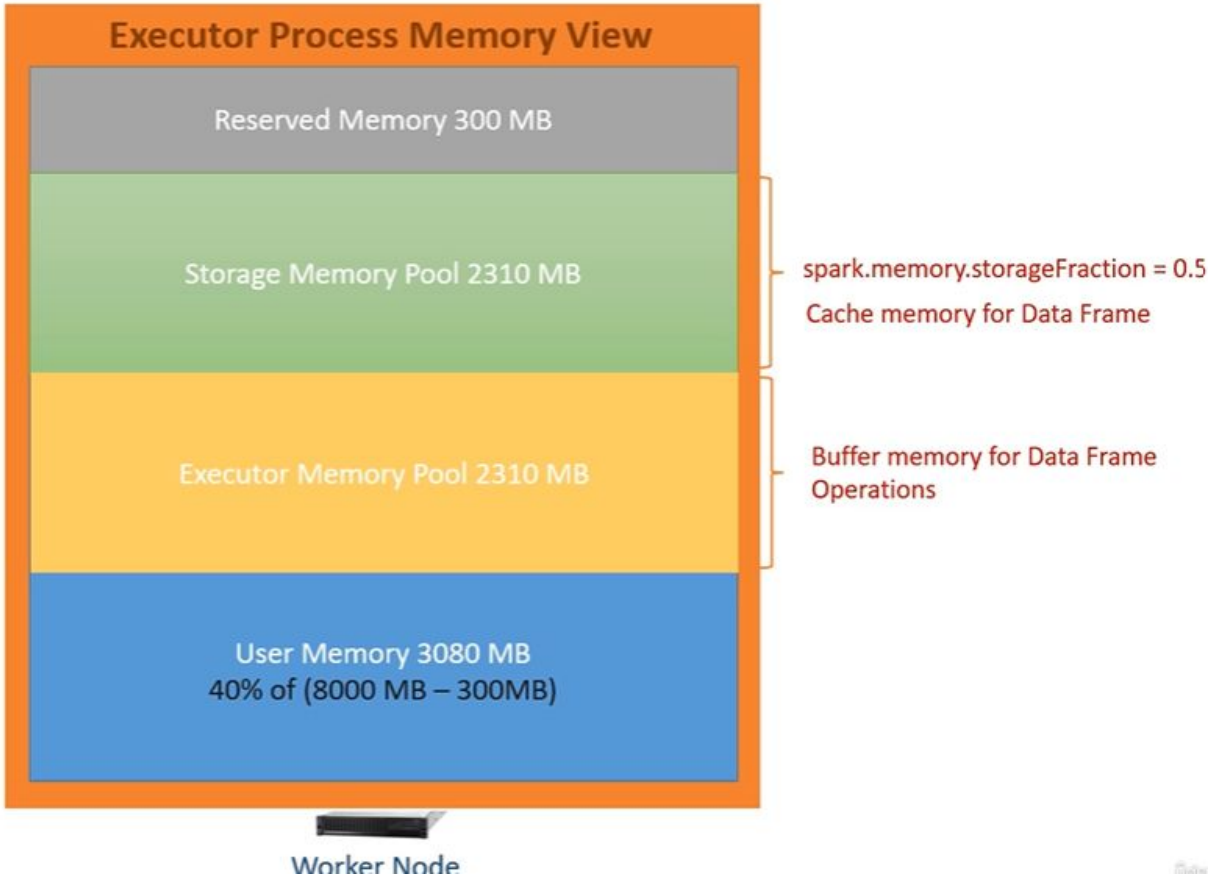
There are three main components involved:

1. **Driver**: Orchestrates the application, schedules tasks, and decides when to add or remove executors.
2. **Cluster Manager** (like YARN, Kubernetes, or Standalone): Allocates executors on request from the driver.
3. **Executors**: Actually run the tasks on worker nodes.

Spark Submit

```
--conf spark.dynamicAllocation.enabled=true \
--conf spark.dynamicAllocation.shuffleTracking.enabled=true \
--conf spark.dynamicAllocation.initialExecutors=4 \
--conf spark.dynamicAllocation.minExecutors=2 \
--conf spark.dynamicAllocation.maxExecutors=20 \
--conf spark.dynamicAllocation.executorIdleTimeout=60s \
--conf spark.dynamicAllocation.cachedExecutorIdleTimeout=300s \
--conf spark.dynamicAllocation.schedulerBacklogTimeout=1s \
--conf spark.dynamicAllocation.sustainedSchedulerBacklogTimeout=1s \
--conf spark.dynamicAllocation.executorAllocationRatio=1.0 \
--conf spark.dynamicAllocation.shuffleTracking.timeout=300s \
--class your.main.ClassName \
your-spark-app.jar
```

spark.executor.memory = 8 GB
spark.executor.cores = 4



Driver Out of Memory

A "driver out of memory" error in Apache Spark typically occurs when the driver program, responsible for orchestrating the Spark application, is allocated insufficient memory to store metadata or results, leading to the application's failure.

Common cause of this error:

- Collecting large amounts of data to the driver
- Large broadcast variables.
- Insufficient driver memory configuration
- Driver overhead

collect() IN APACHE SPARK

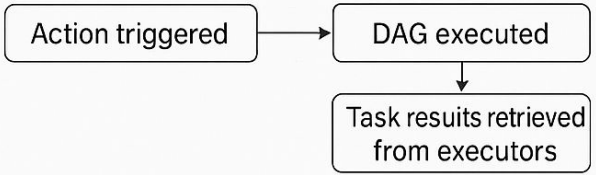
Explanation

- Returns the elements of the dataset as a Python list (or language-specific collection)
- Used to retrieve results to the driver program for use in local computations, debugging, etc.

Memory and Performance Risks

- Collects all elements to the driver, potentially causing Out Of Memory (OOM) errors with large datasets
- Moving data to the driver disrupts "distributed" model and slows down processing

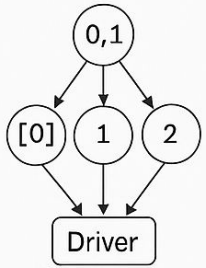
Internal Execution Flow



Real-World Example

```
nums = spark.range(5)
reuult = nums.collect)
```

Output: `[0, 1,2,3, 4]`



Best Practices

- Use collect() sparingly, especially with large datasets
- Prefer actions like take(), show() that limit

Real-Word Exammle

- Use collect() sparingly, especially with large dataseti
- Filter data before calling collect() to reduce resube result size

Executor Out of Memory

An "Executor Out of Memory" error in Apache Spark indicates that the memory allocated to a Spark executor is insufficient to process the data assigned to it.

- Insufficient Executor Memory
- Data Skew
- Large Data Shuffles
- Memory leaks
- Incorrect Configuration

Salting

- Some keys (like "India" or "USA") appear **disproportionately more often** than others.
- In Spark, this causes **uneven partitioning** and a **hotspot** in one executor.

Example:
"India" → 1 million records

"Brazil" → 1,000 records

"Spain" → 500 records

- During **joins or groupBy**, all "India" rows go to **one partition**, overloading it.
- Other partitions finish quickly → **one task delays the entire stage.**

What is Salting?

- **Salting** is a technique to **spread out skewed keys** by appending a **random or systematic suffix** to them — artificially **balancing the data distribution**.
- Each India_# is treated as a **separate key** → goes to **different tasks/executors.**
- **Before Salting:**

  India → India → India → India → India → [sent to one executor]
- **After Salting:**

  India_0 → India_1 → India_2 → India_3 → [sent to multiple executors]

How to do Salting?

Identify skewed keys using .groupBy("key").count().orderBy(desc("count"))

- On the larger DataFrame (left side of join):

  Duplicate skewed rows n times

  Add a random salt value (e.g., 0 to 9)

- On the smaller DataFrame (right side):

  Add all possible salt values for each skewed key

  Explode the salt column to match all variants

Join on key_salted = key + "_" + salt