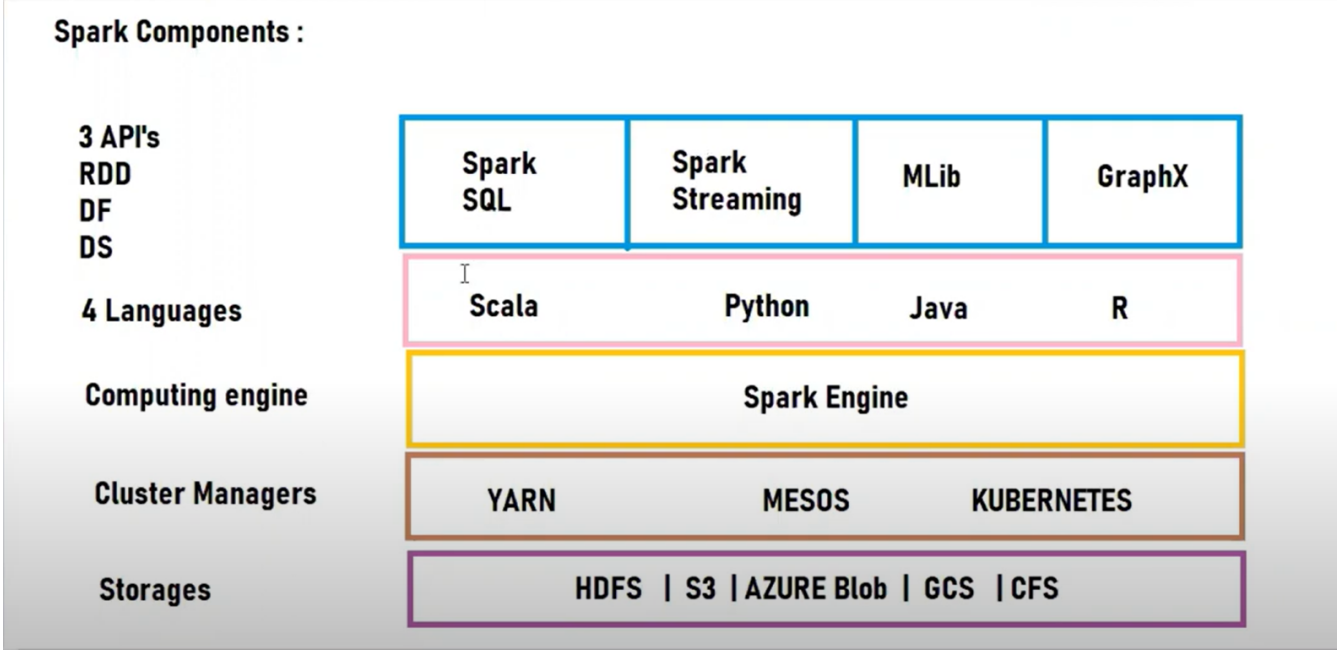
What is Spark?

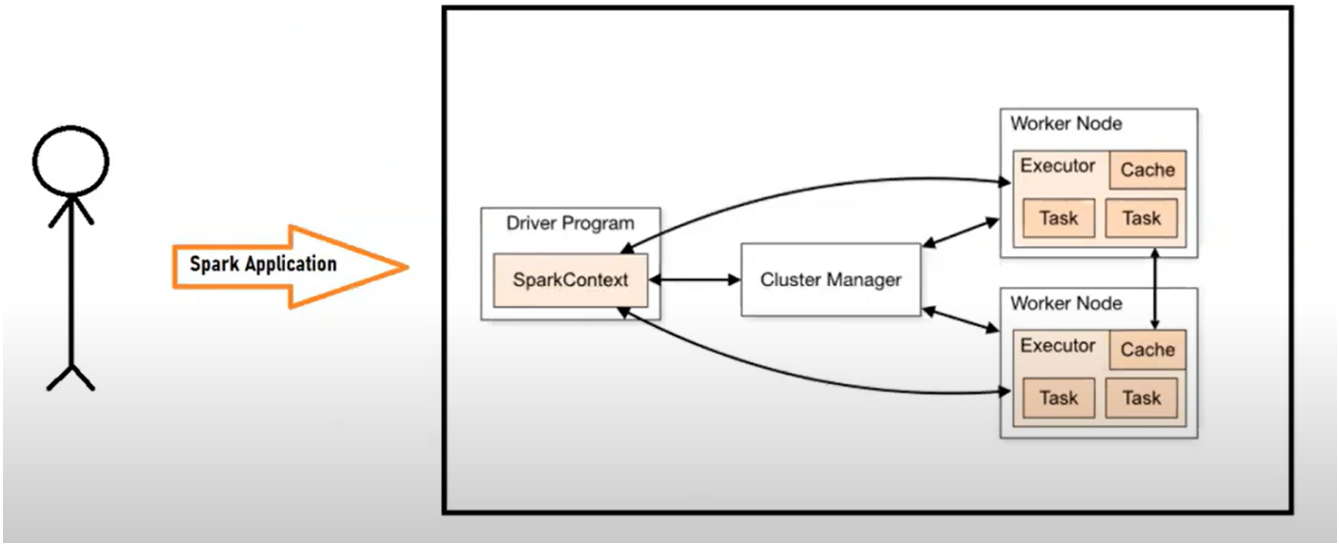
In memory distributed data processing framework/engine.

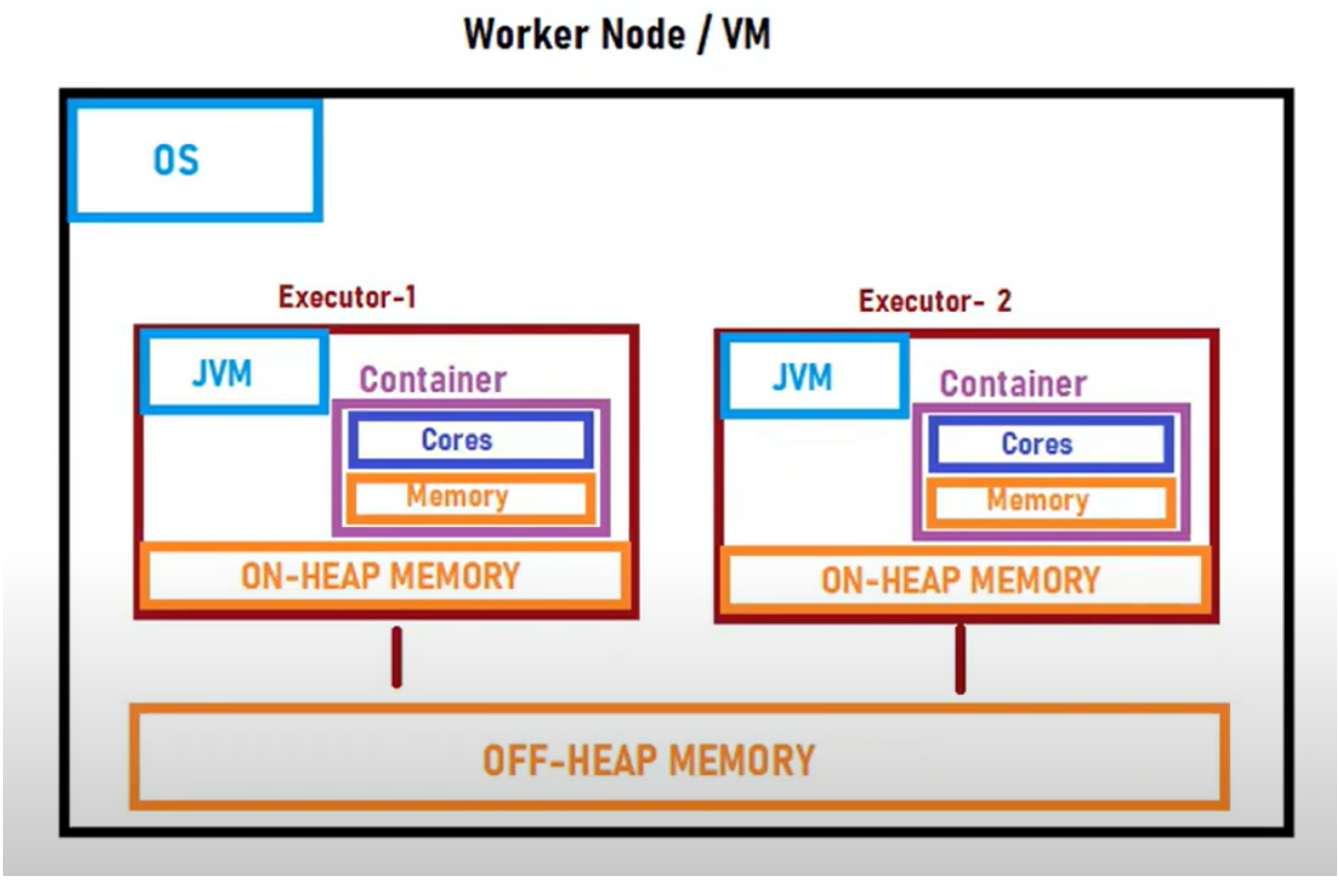
Spark Features:

- In memory computation
- Open source
- Cost effective
- Fault-tolerance
- Supports multiple languages (Python, Scala, Java, and R)
- Lazy Evaluation
- Batch/Near real-time data streaming
- Very rich built-in library support
- As compared to Java (20–25 lines of code), we can replace with single line code in Python/Scala
- 100 times faster in memory than Hadoop MR or other traditional systems
- 10 times faster on disk
- Spark can integrate with Hadoop ecosystem, ETL tools like Talend, Informatica, etc., and cloud platforms (AWS, Azure, GCP)



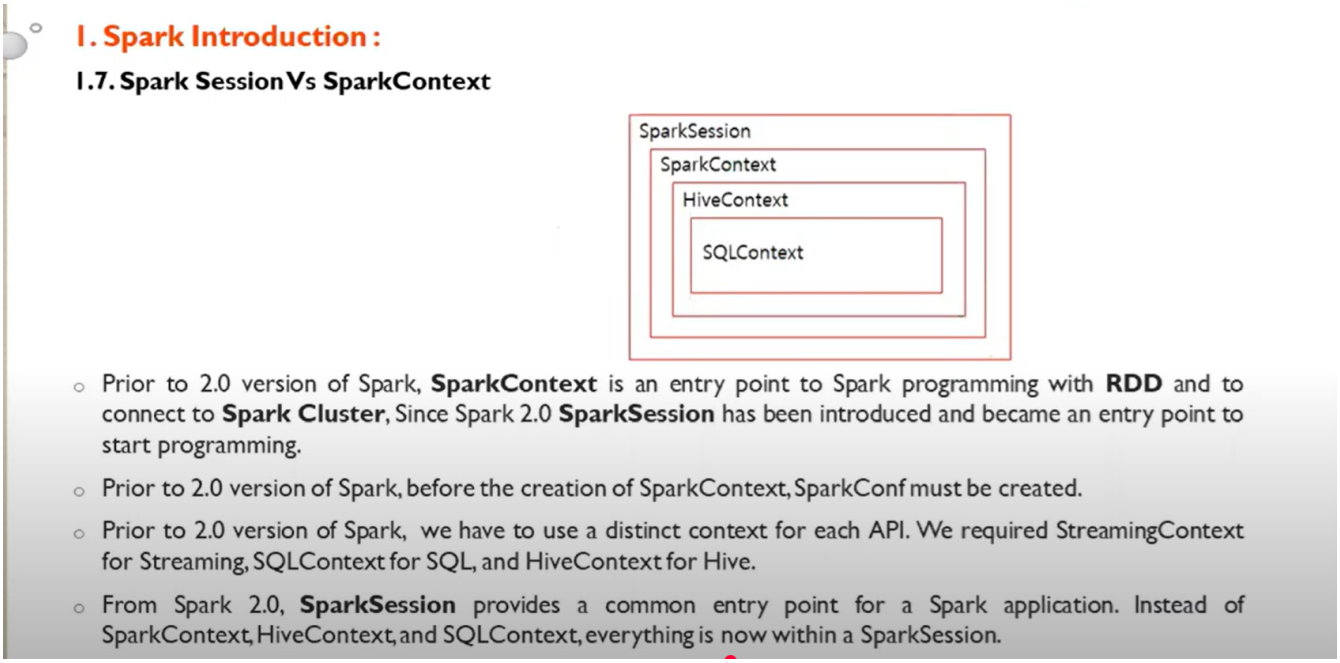
Spark Application Submition-Cluster Mode





On-Heap vs Off-Heap Memory

Aspect	On-Heap Memory	Off-Heap Memory
Location	Inside JVM Heap	Outside JVM Heap (native memory)
Management	JVM Garbage Collector	Spark (Tungsten engine)
Default	Yes	No (needs config)
Performance	Slower for large data (GC overhead)	Faster for large data (no GC)
Storage Format	Java objects	Serialized binary
Use Cases	Small to medium datasets, default Spark jobs	Large datasets, Spark SQL/DataFrame, caching, joins, shuffles
Risk	GC pauses, OutOfMemoryError	Native memory leaks if misused
Configuration	No extra setup	`spark.memory.offHeap.enabled=true` and `spark.memory.offHeap.size`
Example	Default caching `.cache()`	Optimized Tungsten memory, serialized storage



2. Spark RDD's:

2.1.0. Spark RDD – Resilient Distributed Dataset

PySpark RDD (Resilient Distributed Dataset) is a fundamental data structure of PySpark that is fault-tolerant, immutable distributed collections of objects, which means once you create an RDD you cannot change it. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

RDD Creation

In order to create an RDD, first, you need to create a SparkSession which is an entry point to the PySpark application.

Spark session internally creates a SparkContext

```
# Import SparkSession
from pyspark.sql import SparkSession

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

2. Spark RDD's:

2.1.0. Spark RDD – Ways of Creation

SparkContext has several functions to use with RDDs.

using parallelize()

parallelize() method is used to create an RDD from a list.

```
>>> # Create RDD from parallelize
>>> dataList = [("Surya", 10,"Ind"), ("Butler", 150,"England"), ("Rahul",80,"Ind")]
>>> rdd_player=spark.sparkContext.parallelize(dataList)
>>> rdd_player.collect()
[('Surya', 10, 'Ind'), ('Butler', 150, 'England'), ('Rahul', 80, 'Ind')]
```

using textFile()

RDD can also be created from a text file using textFile() function of the SparkContext..

```
>>> rdd2 = spark.sparkContext.textFile("C:/Users/BNK/Dropbox/Clever Studies/Pyspark/rdd2.csv")
>>> rdd2.collect()
['karuna,ECE,90', 'Varun,EEE,86', 'Spandan,IT,88']
```

2. Spark RDD's:

2.1.1. Spark RDD – Operations

In case of MR

In case of Spark

2. Spark RDD's:

2.1.0. RDD Operations in PySpark

RDD supports two types of operations.

1. Transformations:

Transformations are the operations which are applied to an RDD to create a new RDD.

Transformations follow the principle of Lazy Evaluations.

Lazy Evaluation: RDD's execution will not start until an action is triggered

Few transformations are,

map

flatMap

filter

distinct

reduceByKey

sortBy

2. Spark RDD's:

2.1.0. RDD Operations in PySpark

RDD supports two types of operations.

2. Actions:

Actions are the operations which are applied on an RDD, which return a value to the driver program after running a computation on the dataset.

Few actions are,

collect

reduce

countByKey/countByValue

foreach(func)

take

First

saveAsTextFile(path)

Spark Transformations and Actions:

1. Transformations:

Transformations are the operations which are applied to an RDD to create a new RDD. Ex:

map

flatMap

reduceByKey

2. Actions:

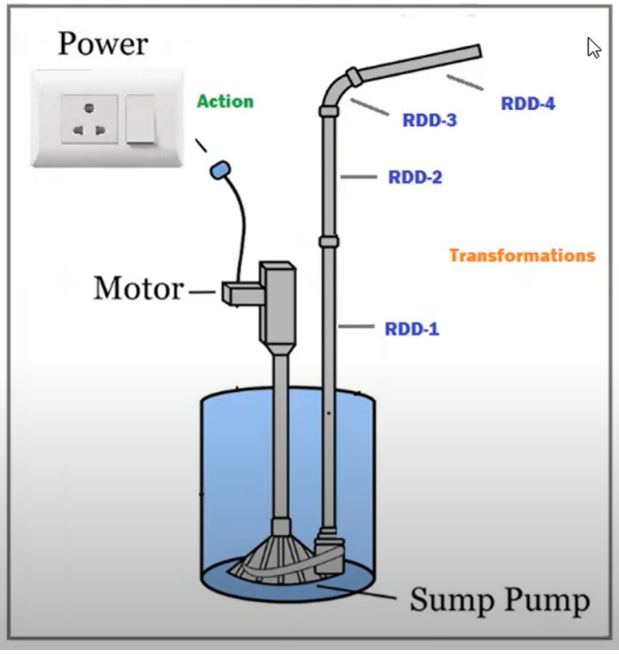
Actions are the operations which are applied on an RDD, which return a value to the driver program after running a computation on the dataset. Ex:

collect

reduce

countByKey/countByValue

foreach(func)

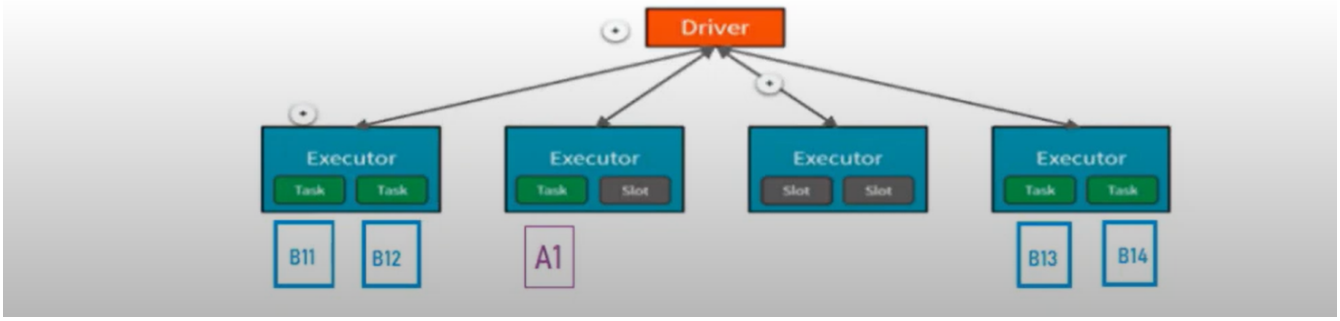


2. Spark Repartition() vs Coalesce()

1. repartition() is used to increase or decrease the RDD/DataFrame partitions
2. coalesce() is used to only decrease the number of partitions in an efficient way.

PySpark repartition() and coalesce() are very expensive operations as they shuffle the data across many partitions hence try to minimize using these as much as possible.

2. Spark partitionBy():



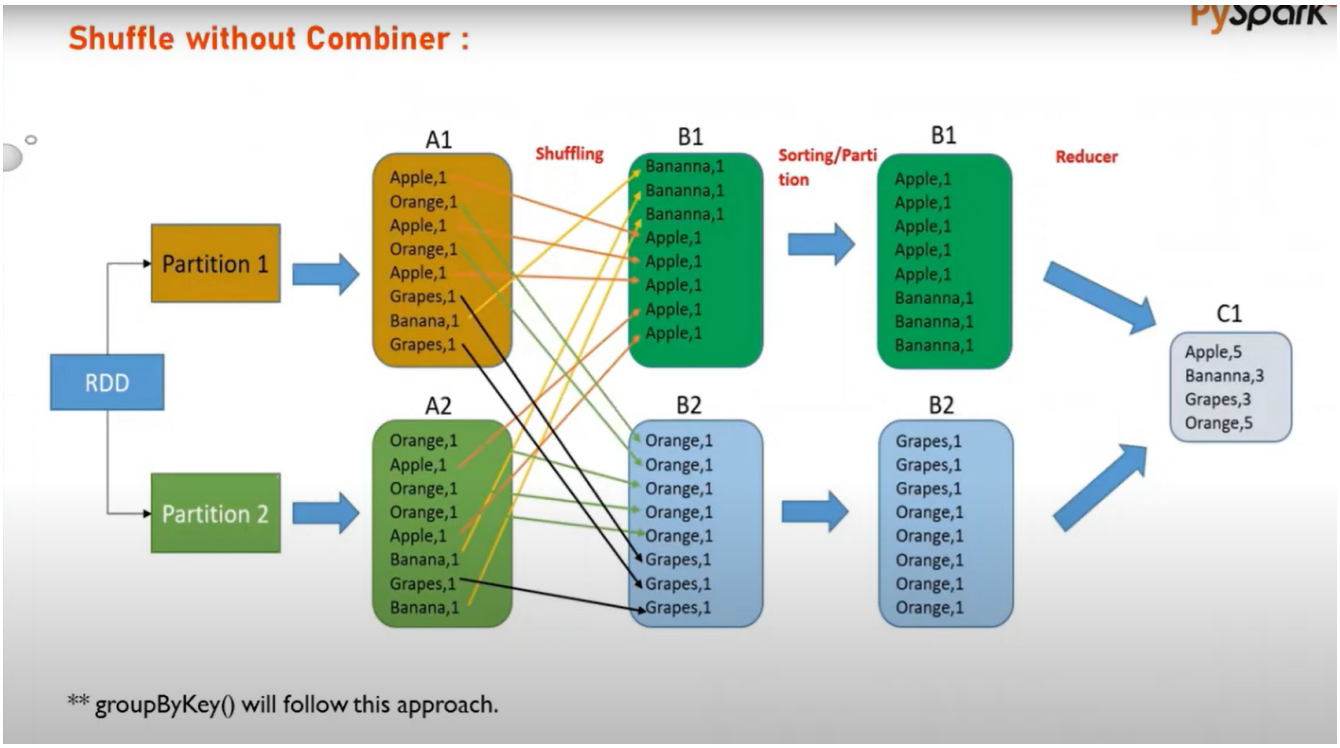
Shuffle and Combiner :

Shuffle:

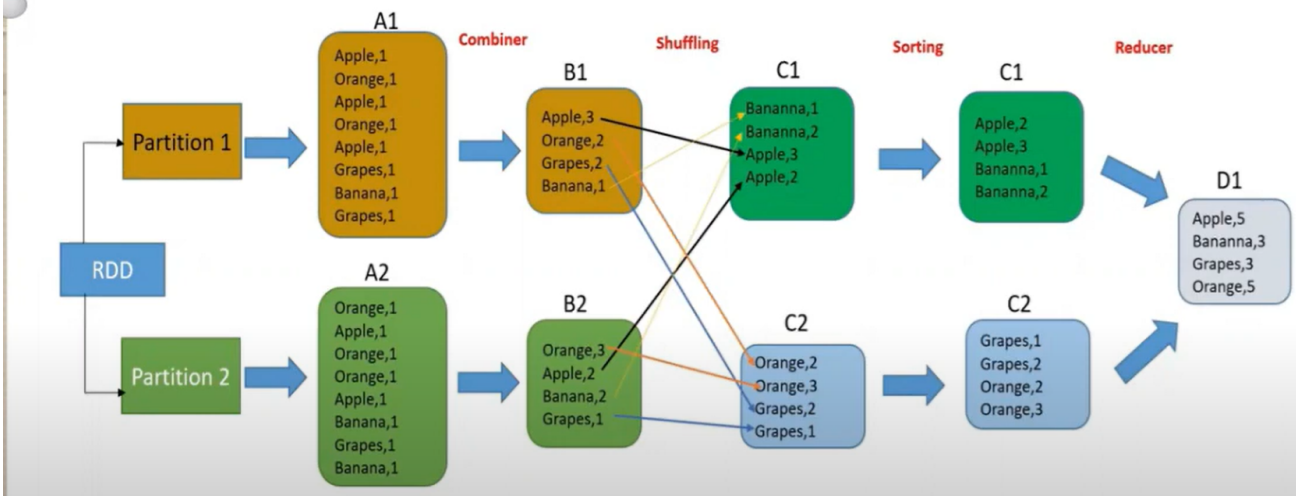
1. Shuffling is a process of redistributing data across partitions.
2. Shuffling will create a new stage in spark
3. Shuffling is a costly operation as it involves disk I/O , network I/O and data serialization/deserialization.
4. Spark reduceByKey(), groupByKey(), aggregateByKey(), join(), union() etc will involve shuffling.
5. Distinct operation will creates a shuffle.
6. If shuffling is necessary, then use combiner.
7. In combiner, shuffling may happen but it is in lesser side.
8. Since, groupByKey() won't use 'combiner', try to avoid to use this where ever is possible. Instead you can use reduceByKey() and aggregateByKey() if possible.

Combiner :

It computes intermediate values for each partition to avoid partial shuffling.



Shuffle with Combiner :



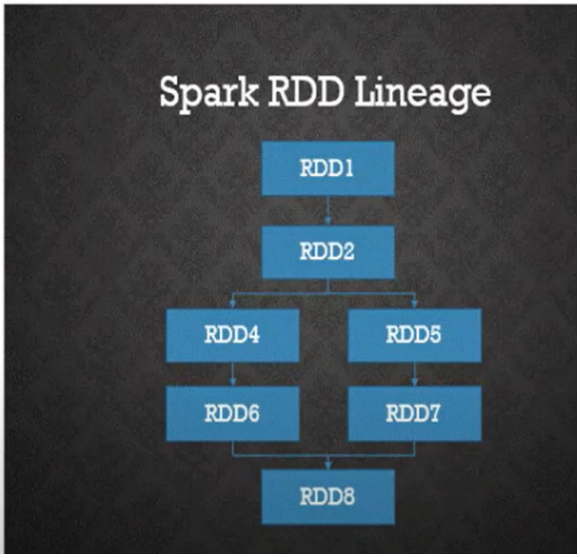
** reduceByKey(),aggregateByKey() will follow this approach.

Spark Type's of Transformations:

Transformations are the operations which are applied to an RDD. Since RDD are immutable in nature, transformations always create new RDD without updating an existing one hence, this creates an RDD lineage.

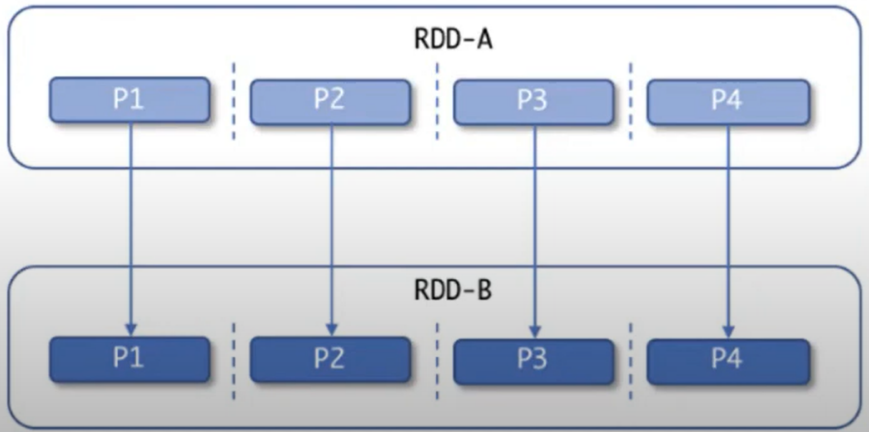
Two types of transformations :

1. Wide Transformations
2. Narrow Transformations



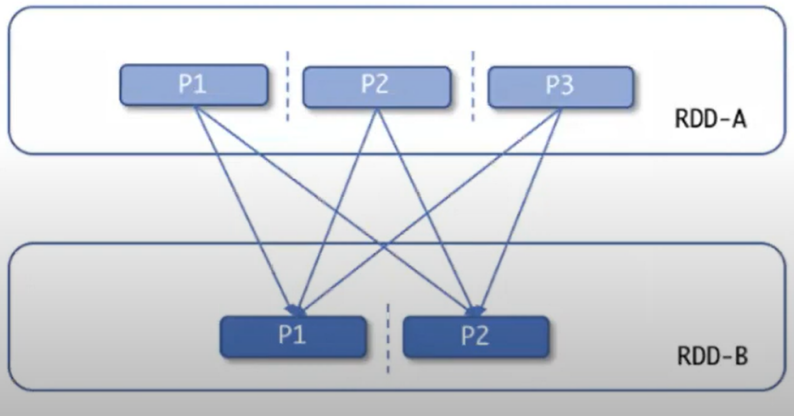
Spark Narrow Transformation:

- ✓ Narrow transformations transform data without any shuffle involved. These transformations transform the data on a per-partition basis; that is to say, each element of the output RDD can be computed without involving any elements from different partitions. This leads to an important point: The new RDD will always have the same number of partitions as its parent RDDs.
- ✓ The map(),filter() are some examples of Narrow transformations.
- ✓ This kind of transformation is basically fast.



Spark Wide Transformation :

- ✓ Wide transformations involve a shuffle of the data between the partitions. In the case of these transformations, the result will be computed using data from multiple partitions and thus requires a shuffle. Wide transformations are similar to the shuffle-and-sort phase of MapReduce.
- ✓ The groupByKey(), reduceByKey(), join(), distinct(), and intersect() are some examples of wide transformations.
- ✓ Slow as compare to narrow dependencies speed might be significantly affected as it might be required to shuffle data around different nodes when creating new partitions.



To understand, how spark executes a job internally

We must know the below concepts,

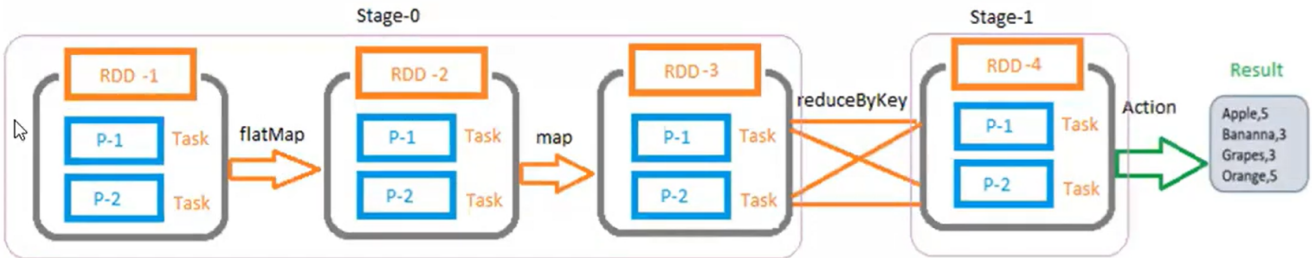
- RDD - An RDD is an immutable distributed collection of elements of a data, partitioned across nodes in the cluster
- Partition - A Partition is a single slice (<= 128 MB chunks of data) of a larger Dataset
- Transformations and Actions
- Types of Transformations
    - 1. Narrow Transformations
    - 2. Wide Transformations
- Spark Job- It is a single computation action that gets instantiated to complete a Spark Action.
- Tasks - The single computation unit performed on a single data partition is called a task. It is computed on a single core of the worker node.
- Stages - A stage is a set of independent tasks all computing the same function that need to run as part of a Spark job

Spark Stages and Tasks:

Input_data - 256 MB (128 MB + 128 MB)

```
List=["Apple Banana Grapes Orange",
      "Apple Banana Orange",
      "Apple Grapes Orange",
      "Apple Banana Orange",
      "Apple Grapes Orange",]
```

```
spark = SparkSession.builder()
   .appName("SparkByExample")
   .master("local")
   .getOrCreate()
rdd = spark.sparkContext.parallelize(List)          RDD-1
wordCount = rdd.flatMap(_.split(" "))               RDD-2
wordCount = wordsRdd.map(f=>(f,1))                  RDD-3
wordCount = wordCount.reduceByKey((a,b)=>a+b)       RDD-4
wordCount.foreach(println)                          Action
                                                    Transformations
```

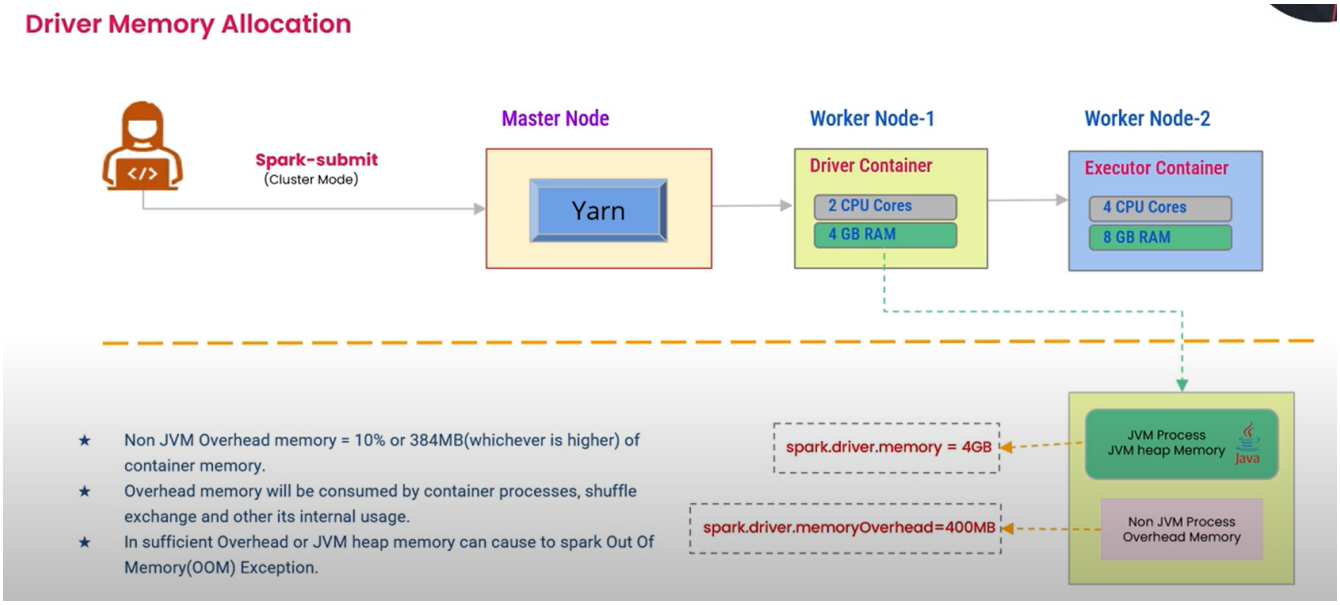


Note in above code, when we are using reduceByKey it involves shuffling, so it creates a new stage(step).

Spark Stages and Tasks:

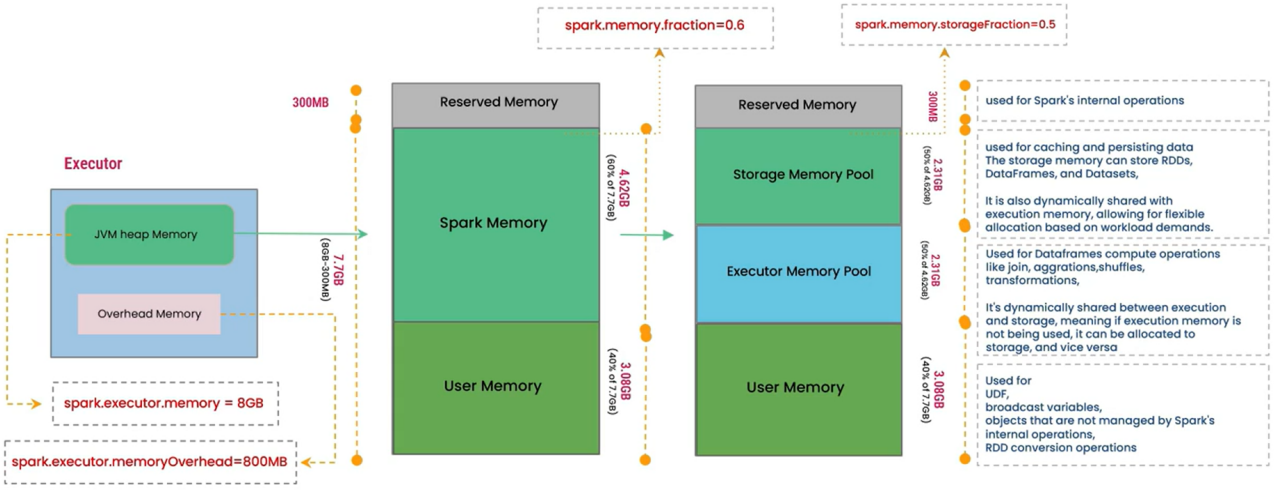
1. A graphical user interface that allows us to view all our Spark Application's jobs, stages, and tasks

Driver Memory Allocation



- ★ Non JVM Overhead memory = 10% or 384MB(whichever is higher) of container memory.
- ★ Overhead memory will be consumed by container processes, shuffle exchange and other its internal usage.
- ★ In sufficient Overhead or JVM heap memory can cause to spark Out Of Memory(OOM) Exception.

Executor Memory deep dive



Note: If the executor memory is less than 1.5 times of reserved memory, Spark will fail with a "please use larger heap size" error message.

To process 25GB data in spark

a. How many CPU cores are required?
b. How many executors are required?
c. How much each executor memory is required?
d. What is the total memory required?

How many executor CPU cores are required to process 25 GB data?

25GB = 25*1024 MB = 25600 MB
Number of Partitions = 25600 MB/128 MB = 200
Number of CPU Cores = Number of Partitions = 200

How much each executor memory is required to process 25 GB data?

CPU cores for each executor = 4
Memory for each executor = 4*512 MB = 2 GB

How many executors are required to process 25 GB data?

Avg CPU cores for each executor = 4
Total number of executor = 200/4 = 50

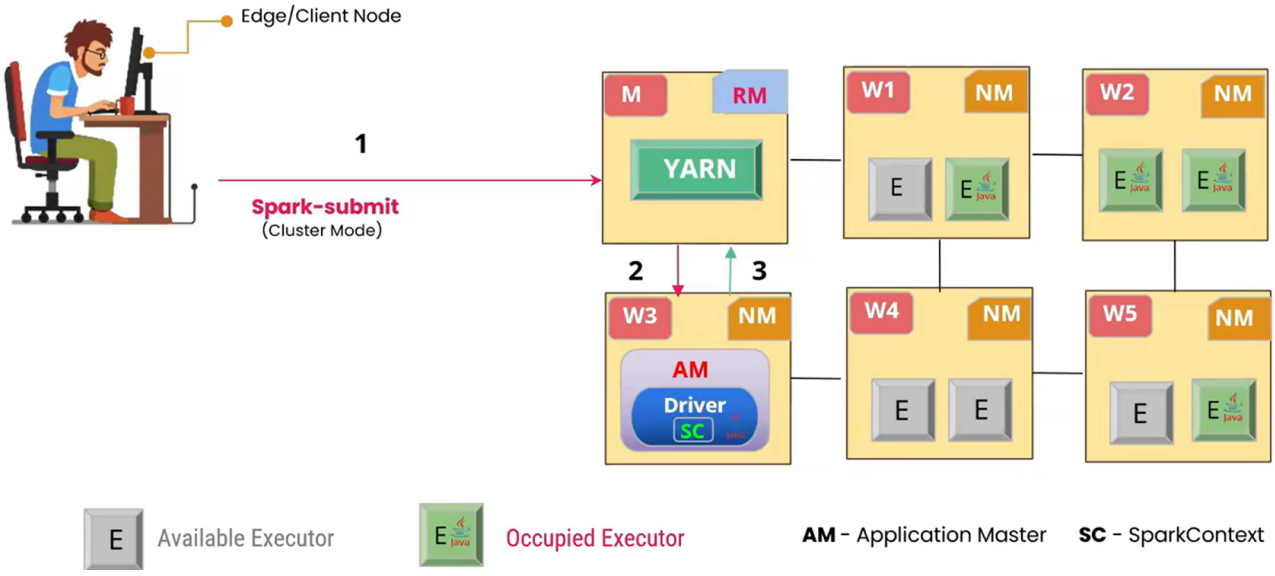
What is the total memory required to process 25 GB data?

Total number of executor = 50
Memory for each executor = 2 GB
Total Memory for all the executor = 50*2 GB = 100 GB

Note:

- By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value.
- To get the better job performance in spark, researchers have found that we can take 2 to 5 maximum core for each executor
- Expected memory for each core = Minimum 4 x (default partition size)=4*128 MB=512 MB
- Executor memory is not less than 1.5 times of spark reserved memory (Single core executor memory should not be less than 450MB)

Spark Runtime Architecture :: Cluster Mode



In [ ]: