RDD in Spark

RDD - Resilient Distributed Dataset

↳ RDD are the main logical data units in spark. They are distributed collection of objects (data / Rows etc), which are stored in memory or on disks of different machines on cluster.

Features of RDD

① Resilience: RDDs track data lineage information (lineage Graph) to recover from failed state automatically. It is also known as fault tolerance.

Ex: ① rdd1 = spark.read_csv("employee.csv")

② rdd2 = rdd1.filter("salary > 10000")

③ rdd3 = rdd2.groupBy("dept_id").sum()

rdd1 → rdd2 → rdd3 lineage

② Distributed: Data present in an RDD resides on multiple nodes.

③ Lazy Evaluation: Data does not get loaded in an RDD even if you define it.

④ Immutability: Data stored in an RDD is in the read only mode. We can't change it's value during run time.

⑤ In-memory Computation:

An RDD stores any intermediate dat that is generated in the RAM so that faster access can be provided.

*) RDD vs Dataframe ?

⇒ RDD = It can have any type of data (unstructured, semi, structured)

→ Rdd1 = spark.read("data.txt")

⇒ Dataframe = It is a wrapper on top of RDD and only deals with structured data.

df1 = spark.read("employee.csv")

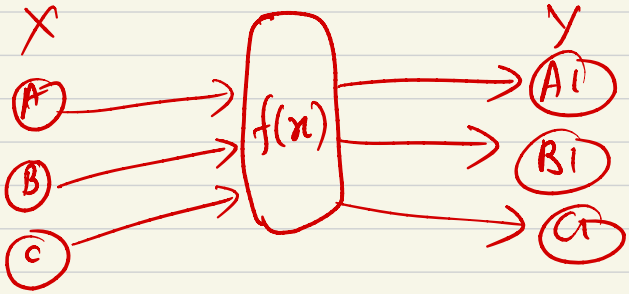
Actions & Transformations in Spark

*) Transformation : It is a function that produces a new RDD from existing RDD. It takes RDD as input & generate new RDD as output.

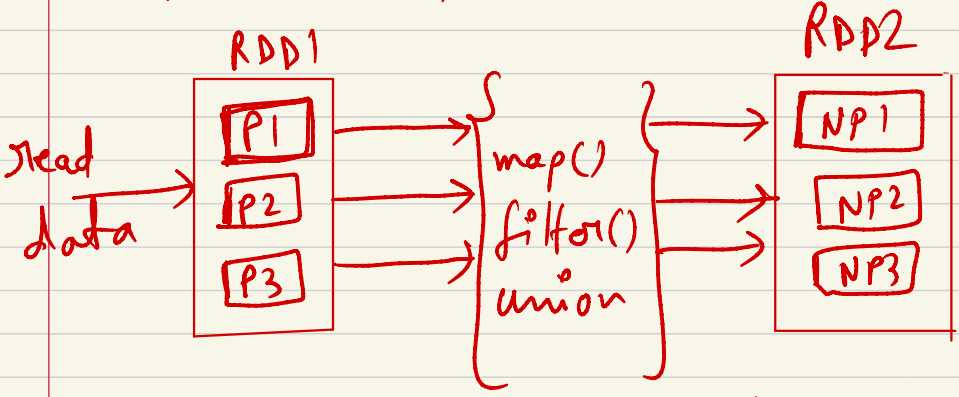
There are two types of transformation

↳ Narrow Transformation (No Data Shuffle)
↳ Wide Transformation (Data Shuffle)

*) Narrow Transformation

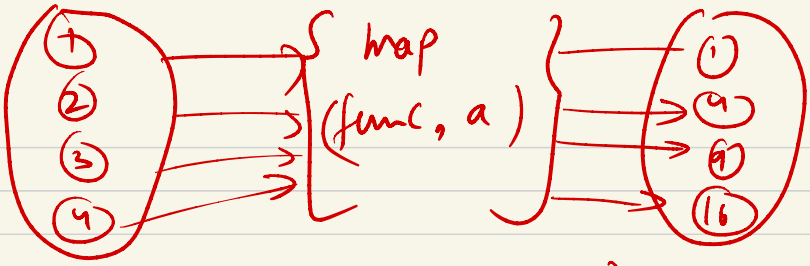


→ map()
→ flatmap()
→ filter()
→ Union()



a = [1, 2, 3, 4], result = [1, 4, 9, 16]

func = lambda x : x**2



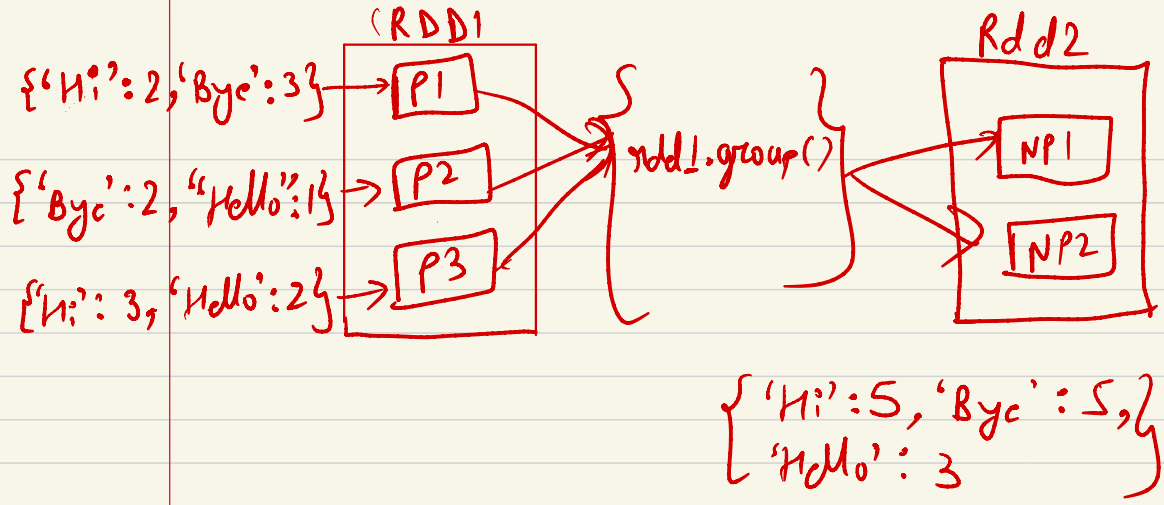
result = map(func, a)

*) <u>Wide Transformation</u>

↳ All the elements that are required to compute the records in a single partition, may live in many partitions. That's why we need to shuffle the data.

→ join()
→ GroupBy()
→ Repartition()
→ Coalesce()
→ ReduceByKey()
→ GroupByKey()

<u>Example of word count</u>



Actions in Spark

↳ Action is a way of sending data from executor to the driver.

- → Count()
- → Collect()
- → take()
- → top()
- → show()

employee.csv
- ↳ emp_id
- ↳ emp_name
- ↳ salary
- ↳ dept_id

Departme.csv
- ↳ dept_id
- ↳ dept_name

Demo Pyspark application (first.py)

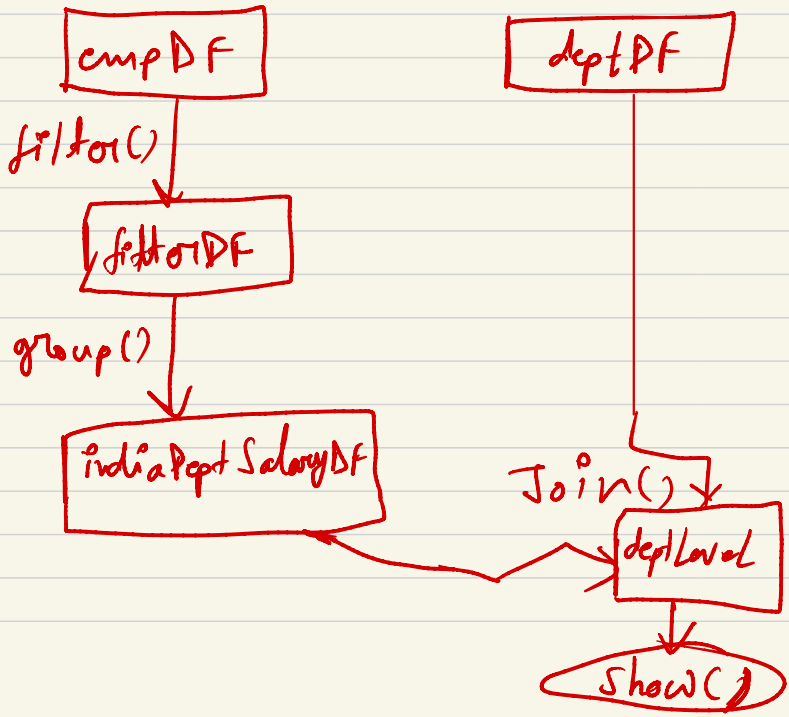
```
spark = SparkSession.getorCreate()

empDf = spark.read_csv("employee.csv")

deptDf = Spark.read_csv("department.csv")
```

```
filterDf = empDf.filter("country = 'INDIA'")

indiaDeptSalary = filterDf.groupBy('dept_id').agg()

deptLevelSalary = indiaDeptSalary.join(deptDF)

deptLevelSalary.show()
```

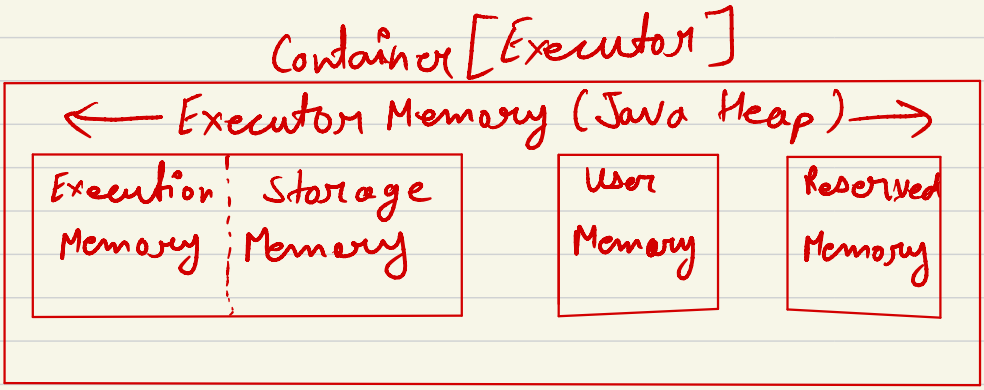
→ Display the data

Lineage Graph (DAG)
{Directed Acyclic Graph}

Lazy Evaluation



Spark Memory Management



Container (Executor)

- CPU Cores
- RAM
- Network Bandwidth

Spark Submit Parameters

```
spark-submit \
  --master <master-url> \
  --deploy-mode <cluster or client> \
  --conf <key:value>
  --driver-memory ng \
  --executor-memory ng \
  --num-executor N \
  --jars <comma separated files> \
```

Word_count.py [space separated command line args]

Q.) We have 1 node with 32 GB RAM and each executor can have 4 GB memory

2.

Total executors = 32/4 = 8

(i) Reserved Memory:

This is the storage for running executor. It will store the data of internal processing which is required to execute the container. After spark 1.6+ version, it is fixed (300MB)

(2) User Memory: It stores the data structures, metadata and user defined data structures.

(3) Storage Memory: It stores cache data, data of broadcast variables.

④ Execution Memory:

It's a storage for the data required for executions, shuffle, join, aggregation.

---

*) Example of memory distribution

Let say we have given 4GB memory to each executor.

Executor-memory (Java Heap) = 4GB
= 4096 MB

Spark's memory management depends on two important params.

spark.memory.fraction = 75% (0.75) → default

spark.memory.storagefraction = 50% (0.50)

→ Reserved Memory : It is fixed 300MB

→ User Memory:

formula = (Java Heap - Reserved Memory)
*
(1 - spark.memory.fraction

= (4096 - 300) * (1 - 0.75)

= 949 MB

→ Storage Memory -

formula = (Java Heap - Reserved Mem)
*
spark.memory.fraction
*
spark.memory.storagefraction

= (4096 - 300) * 0.75 * 0.5

≅ 1423.5 MB or 1.4GB

→ Execution Memory

formula = (Java Heap - Reserved Memory)
*
spark.memory.fraction
*
(1 - spark.memory.storagefrac)

$= (4096 - 300) * .75 * (0.5)$

$\cong 1423.5$ MB or 1.4 GB

After Calculation if we verify

Reserved Memory = 300MB = 7.32 %

User memory = 949 MB = 23.16 %

Spark Memory
- Storage = 2847 MB
- Execution = 69.5 %

Rules for execution & storage memory

Execution Memory →	Storage Memory
	← Memory

① Storage memory can borrow space from execution memory, only if memory blocks of execution memory are not in use.

② Execution memory can also borrow space from storage memory if blocks are not in use.

③ If blocks from execution memory is used by storage memory and execution memory need more memory, it can forcefully evict the excess blocks occupied by storage memory.

④ If blocks from storage memory is used by execution memory and storage memory needs more memory. It can not forcefully evict the excess blocks occupied by execution memory. It will end up having less memory. It will wait until spark releases the excess blocks stored by execution memory then occupies it.

Ex Resource allocation for spark application

Case 1: Hardware → 6 Nodes in cluster
each Node is having 16 Cores & 64 GB RAM

⇒ From Recommendation 5 CPU Core for a executor

① 1 GB & 1 Core of each node will be required for operating System.

1 Node → 15 Cores & 63 GB

*) How many executors?

1 executor = 5 Cores
15 Cores = 15/5

1 Node = 3 executors

Total executors on cluster
= 6 × 3 = 18 exe

1 executor will be created by Yarn as application master.

So final executors which we should mention in spark-submit

$\boxed{\text{num-executors} = 18 - 1 = 17}$

*) Memory for each executor?

1 Node = 3 executor

1 Node = 63 GB

1 Executor = 63/3 = 21 GB

formula for overhead memory,

$\boxed{\text{spark.memory.overhead} = \max(384\text{MB}, 0.07 * \text{exe mem})}$

overhead memory = max(384MB, 0.07 * 21)

= max(384 MB, 1.47G)

$\boxed{\text{Overhead mem} = 1.47\text{GB}}$

final executor mem = 21GB - 1.47G

Memory for each executor ≅ 19GB

final values of spark-submit params based on case-1, resources.

5 CPU core for executor
17 executors
19 GB for each executor