INTRODUCTION TO SPARK:

SPARK:
-->It is a general purpose
In memory
Compute/process engine

======================== compute engine
Hadoop provides us with the below:

1. Hdfs ==> storage
2. Map reduce ==> process engine
3. YARN ==> resource manager

-->spark is alternative/replacement of map-reduce

Spark can work with the below:
Storage==> Local, hdfs, amazon s3 (cloud)
Process engine ==> Spark
Resource manager: YARN, MESOS, kubernetes

======================== In memory
Why spark is very fast?
MAP-REDUCE: For every MR job, mapper reads the data and writes the data back to hdfs. It requires 2 disc operations for every mapper. In general we have hundreds of mappers for each task.
-->Disc I/O is a tedious task due to which the computation more time.

SPARK: For every spark job, the processing happens IN-MEMORY ie after the data is read from disc it processes the entire data in-memory and writes the final result to disc.
-->**Only 2 disc I/O** due for a spark job due to which it is very fast processing.

==============================General Purpose

Spark can be used for a wide range of activities like cleaning, querying, streaming data which is all under one roof.

How is data stored in spark?

-->The **data is stored in spark in the form of RDD** .

-->RDD is a distributed in-memory collection

-->RDD (resilient distributed dataset)

================================Resilient

What happens when an RDD fails/loses data ?

-->It means ability to get back to normal after a failure

-->We don't have replication as fault tolerance as we deal with in-memory and it is a costly process.

-->By making use of **RDD Lineage graph** we try to retrieve for failed RDD

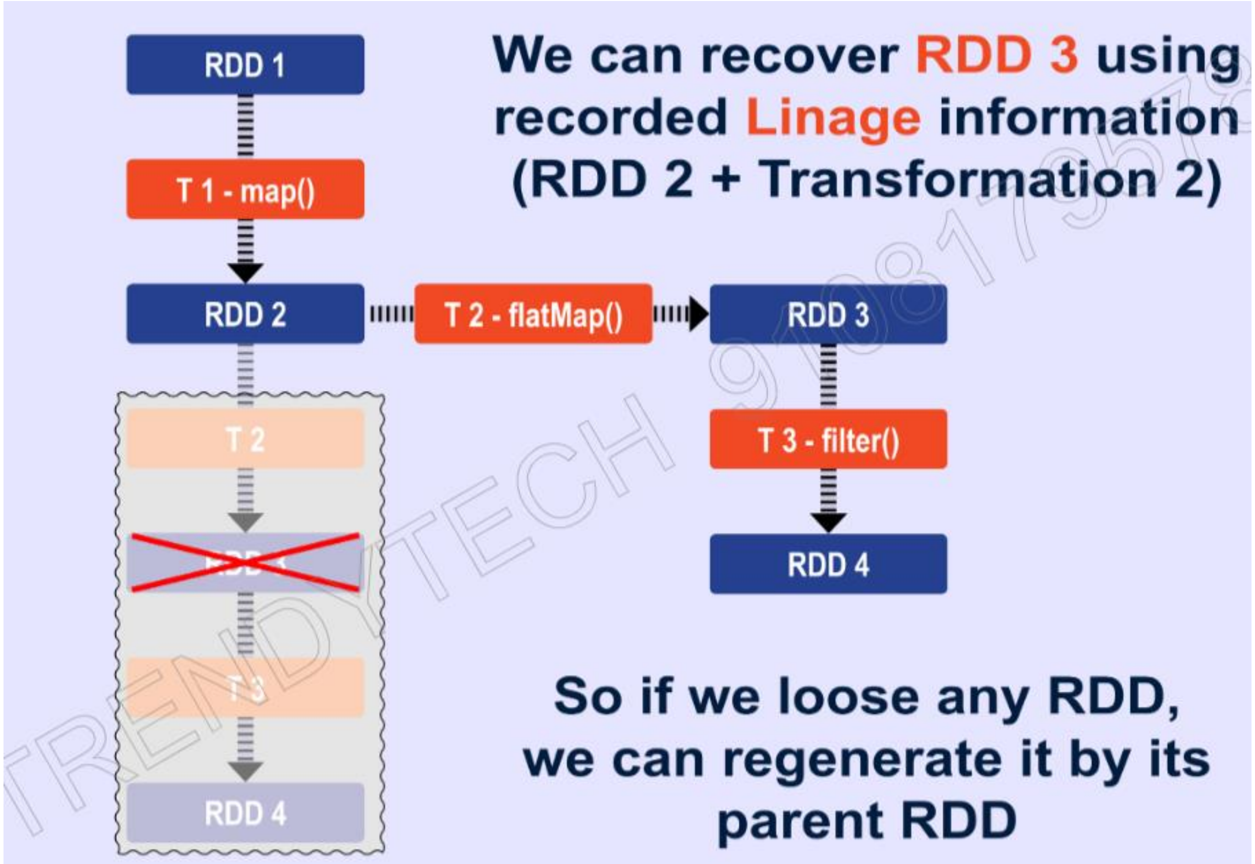
RDD1

|==> Map

RDD2

| ==> Filter

RDD3



We can recover RDD 3 using recorded Linage information (RDD 2 + Transformation 2)

So if we loose any RDD, we can regenerate it by its parent RDD

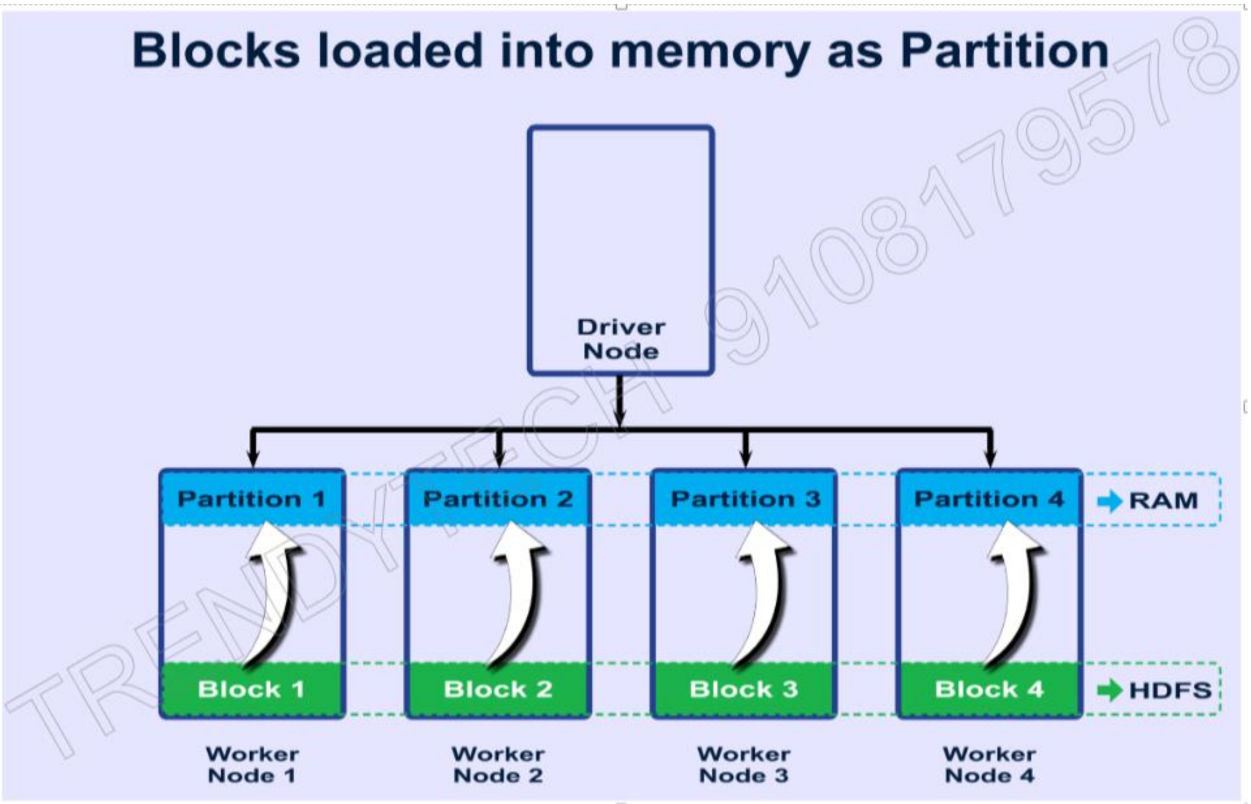
-->If RDD3 fails, then it gets back to its parent RDD2 and asks it to peform "flatMap" and "filter" again to generate RDD3 which got failed previously.

Why RDD are immutable?

-->If RDD are mutable ie if we store all the transformation & action on single RDD.

-->what if if this RDD fails, we just lose all the data?

-->In order to **make the RDD resilient we make it immutable**



What are the various operations in spark?

1. **Transformations**:
   -->They **change the data from one form to another**.
   -->They are **lazy** ie they don't execute until an "action" is performed

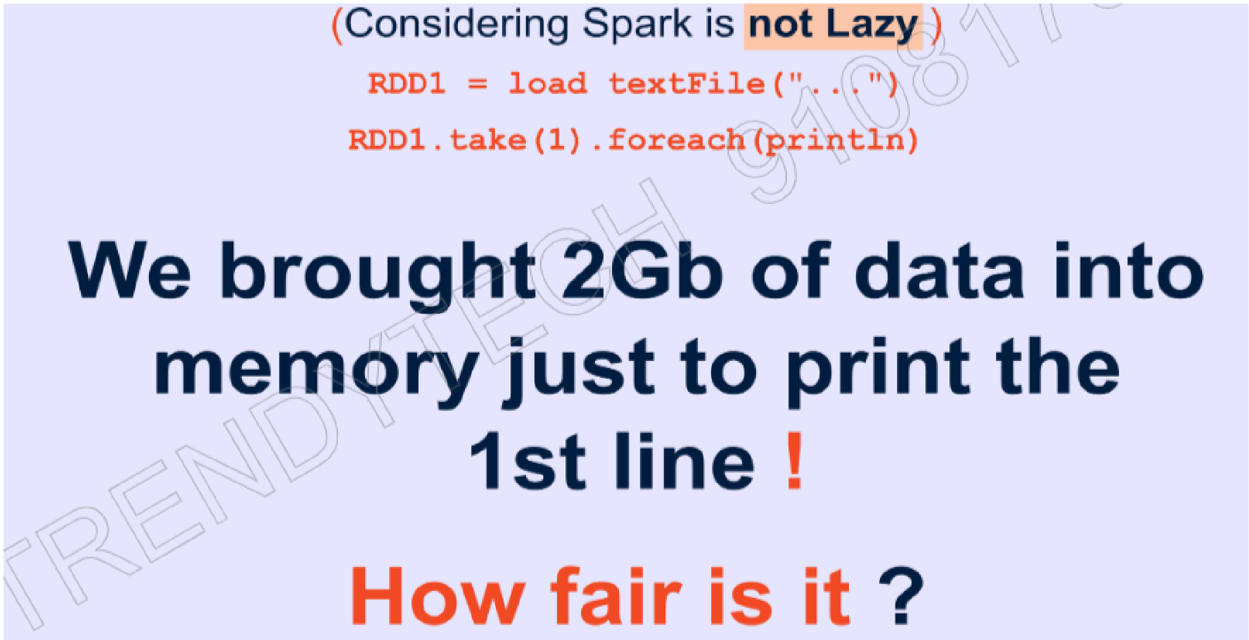
2. **Action**:
   -->They are in-charge of actually performing the **computation**
   -->They are **not lazy** and execute when they are called

Why spark is lazy?

1. Spark **computation are costly as it is in-memory**
2. We need to make sure that we optimize the computation in best possible way before giving it to spark

3. They are lazy so they don't compute each & every statement every time they were created
4. They prepare optimized plan sheet based **(DAG)on which they will compute when an "action" is called**

Ex1. To load 2GB file into spark and print 1st line

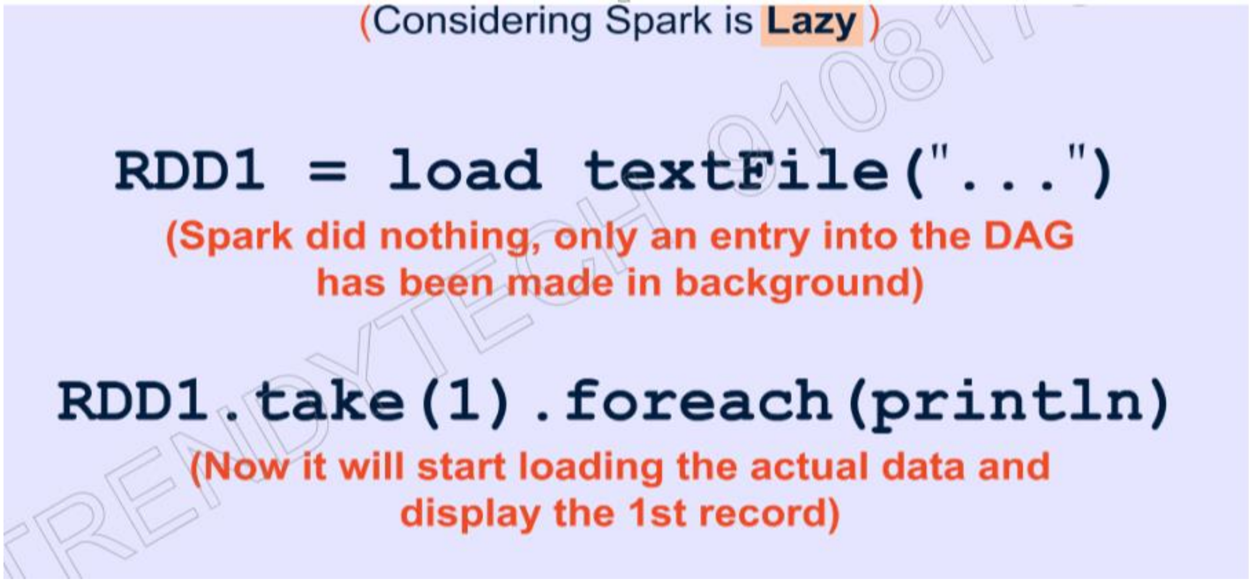


(Considering Spark is **not Lazy**)

```
RDD1 = load textFile("...")
RDD1.take(1).foreach(println)
```

We brought 2Gb of data into memory just to print the 1st line !

How fair is it ?



(Considering Spark is **Lazy**)

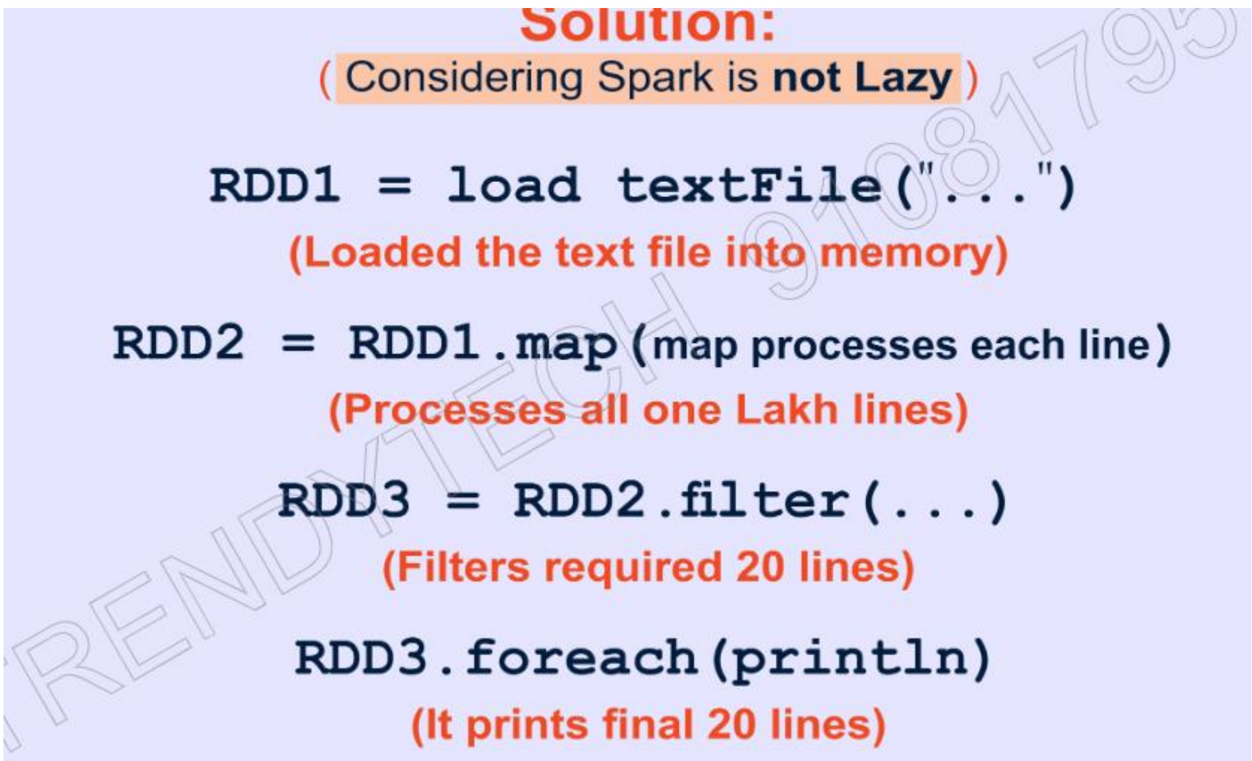
```
RDD1 = load textFile("...")
```

(Spark did nothing, only an entry into the DAG has been made in background)

```
RDD1.take(1).foreach(println)
```

(Now it will start loading the actual data and display the 1st record)

Ex2. We have to perform filtering of data



Solution:

(Considering Spark is **not Lazy**)

```
RDD1 = load textFile("...")
```

(Loaded the text file into memory)

```
RDD2 = RDD1.map (map processes each line)
```

(Processes all one Lakh lines)

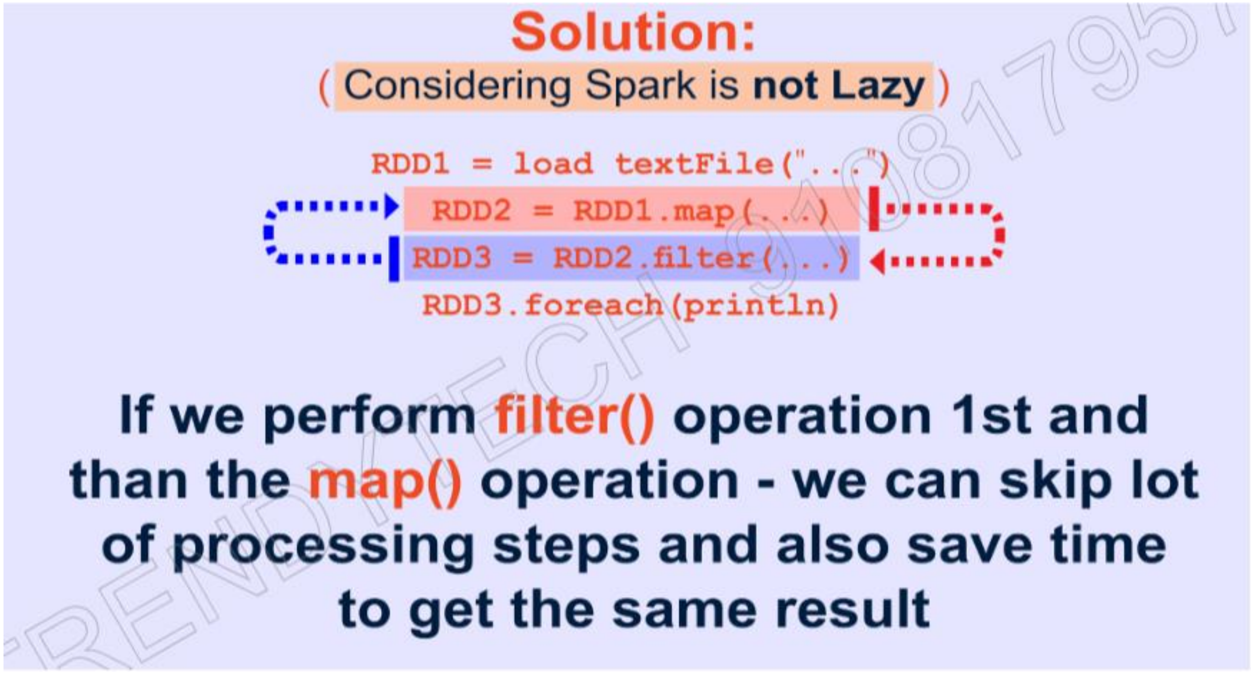
```
RDD3 = RDD2.filter(...)
```

(Filters required 20 lines)

```
RDD3.foreach(println)
```

(It prints final 20 lines)

-->It perform **Predicate pushup** to optimize the query plan ie it alters the steps to be performed

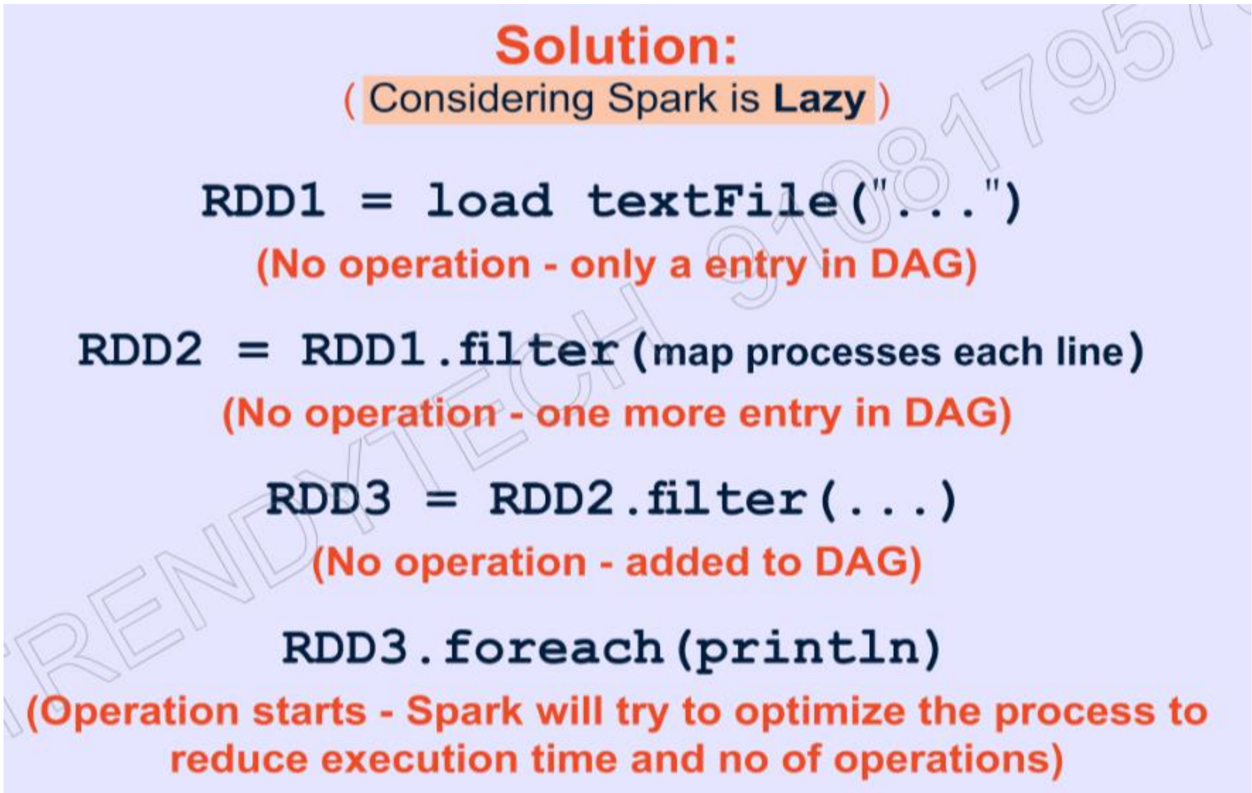


Solution:

(Considering Spark is **not Lazy**)

```
RDD1 = load textFile("...")
RDD2 = RDD1.map(...)
RDD3 = RDD2.filter(...)
RDD3.foreach(println)
```

If we perform filter() operation 1st and than the map() operation - we can skip lot of processing steps and also save time to get the same result



Solution:

(Considering Spark is **Lazy**)

```
RDD1 = load textFile("...")
```

(No operation - only a entry in DAG)

```
RDD2 = RDD1.filter(map processes each line)
```

(No operation - one more entry in DAG)

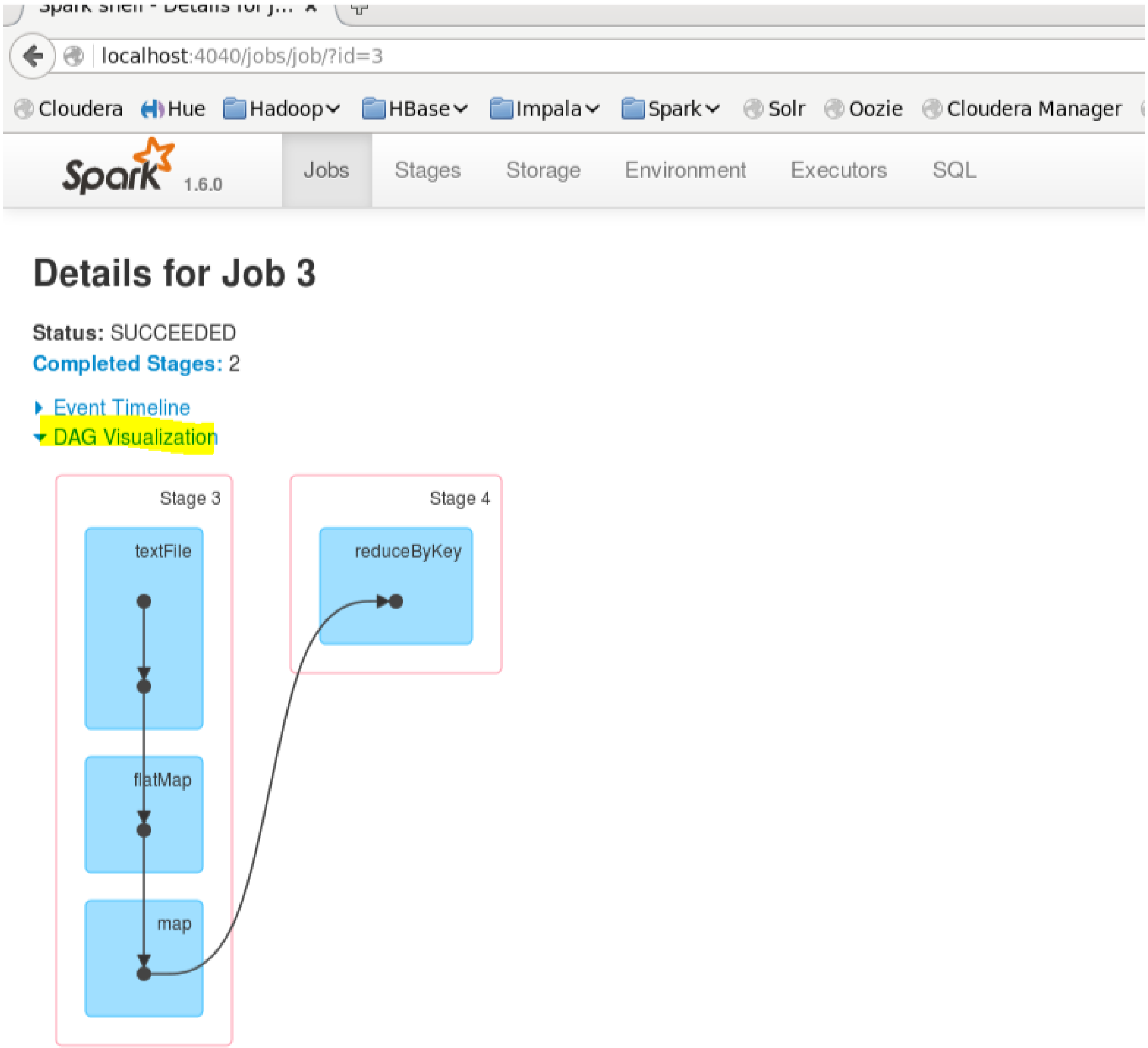
```
RDD3 = RDD2.filter(...)
```

(No operation - added to DAG)

```
RDD3.foreach(println)
```

(Operation starts - Spark will try to optimize the process to reduce execution time and no of operations)

-->spark-shell ==>to run spark commands in cloudera (we get an interactive scala shell)
-->localhost:4040 ==>UI for spark (only active when spark-shell is opened)



Details for Job 3

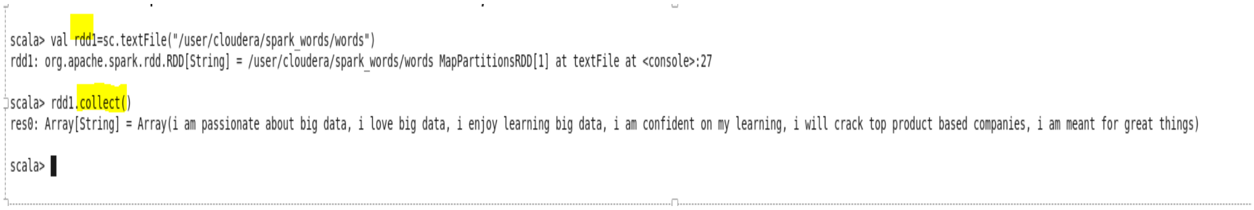
Status: SUCCEEDED

Completed Stages: 2

-->**SC** (spark context) is the **entry point for a spark cluster**. We need to mention it if we have to run our queries on spark cluster and achieve parallelism

-->we have to load the file, as it is lazy only DAG is created and loading not yet done

-->when action is performed "collect" then our file is actually loaded



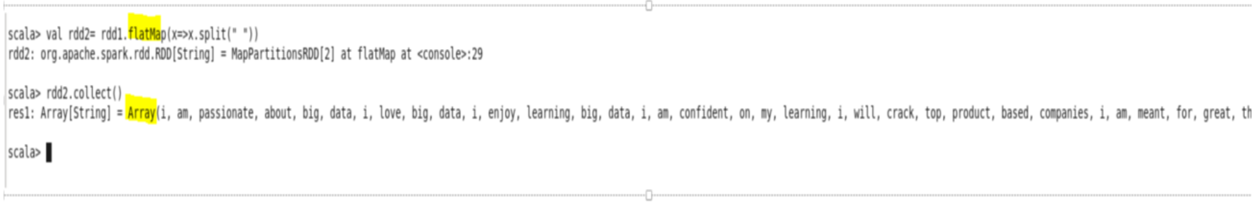
```
scala> val rdd1=sc.textFile("/user/cloudera/spark_words/words")
rdd1: org.apache.spark.rdd.RDD[String] = /user/cloudera/spark_words/words MapPartitionsRDD[1] at textFile at <console>:27

scala> rdd1.collect()
res0: Array[String] = Array(i am passionate about big data, i love big data, i enjoy learning big data, i am confident on my learning, i will crack top product based companies, i am meant for great things)

scala> ▌
```

-->flatMap() ==> It takes each line as a input and transforms it according to our operations. It is an array of arrays(each line stored as array)

-->Array(Array(I,am,passionate,about,big,data), Array(I,love,big,data)…..)

-->ie it takes each line as input and splits it based on " " and creates an array for each line

-->finally all the arrays are merged into single array as seen below.



```
scala> val rdd2= rdd1.flatMap(x=>x.split(" "))
rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:29

scala> rdd2.collect()
res1: Array[String] = Array(i, am, passionate, about, big, data, i, love, big, data, i, enjoy, learning, big, data, i, am, confident, on, my, learning, i, will, crack, top, product, based, companies, i, am, meant, for, great, th

scala> ▌
```

-->**map**: works on each input given. For **n inputs we get n outputs**.

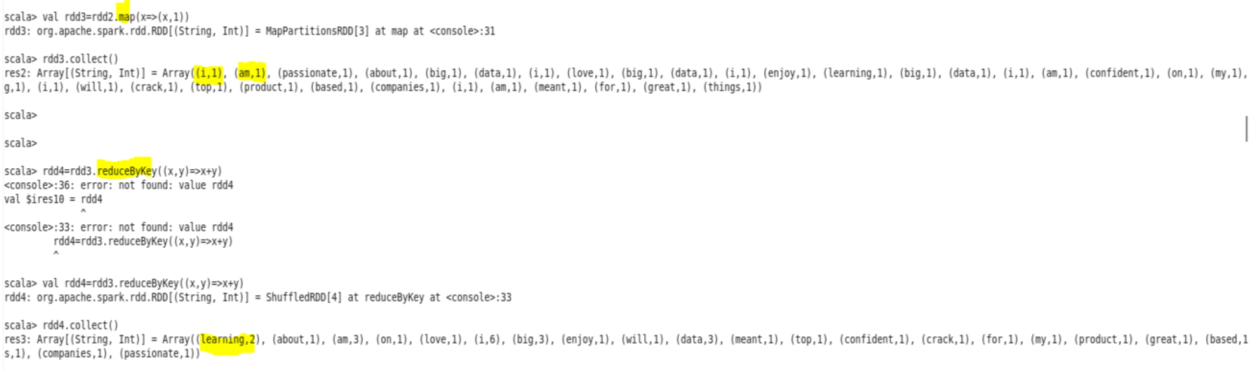
-->reduceByKey: It sorts the data and computes for 2 records at a time.

(am,1)

(am,1) ==>(am,2)

(am,1) ==>(am,3)

(am,1) ==>(am,4)

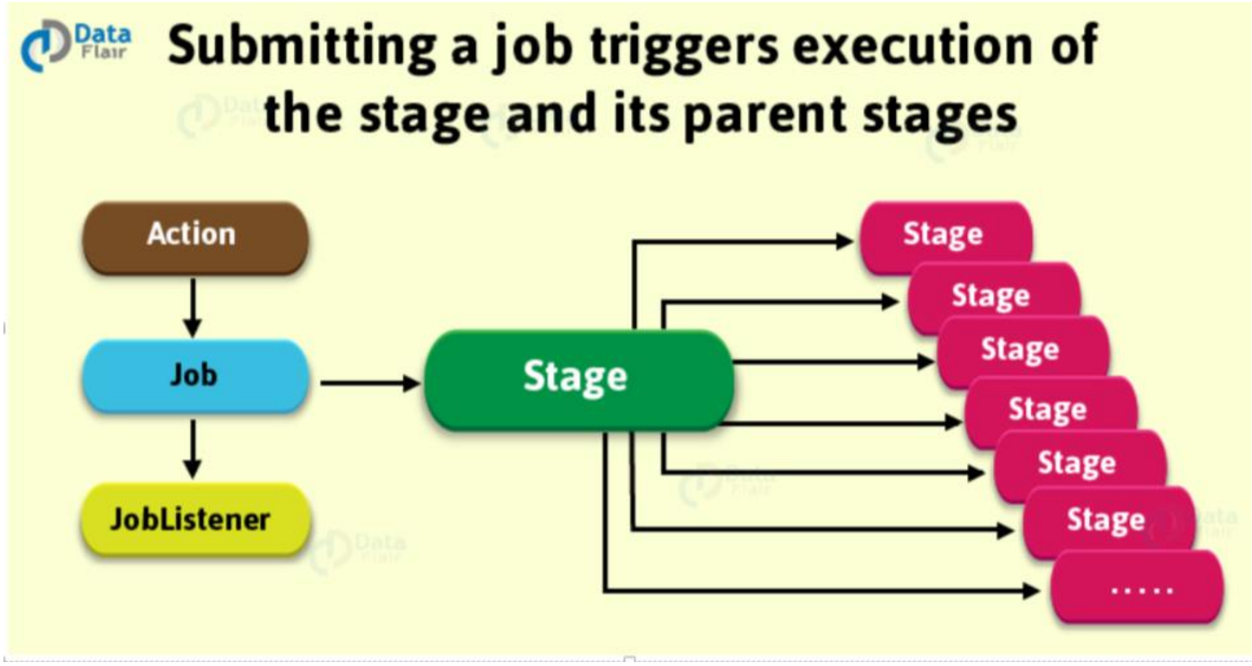


```
scala> val rdd3=rdd2.map(x=>(x,1))
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:31

scala> rdd3.collect()
res2: Array[(String, Int)] = Array((i,1), (am,1), (passionate,1), (about,1), (big,1), (data,1), (i,1), (love,1), (big,1), (data,1), (i,1), (enjoy,1), (learning,1), (big,1), (data,1), (i,1), (am,1), (confident,1), (on,1), (my,1),
g,1), (i,1), (will,1), (crack,1), (top,1), (product,1), (based,1), (companies,1), (i,1), (am,1), (meant,1), (for,1), (great,1), (things,1))

scala>

scala>

scala> rdd4=rdd3.reduceByKey((x,y)=>x+y)
<console>:36: error: not found: value rdd4
val $ires10 = rdd4
              ^

<console>:33: error: not found: value rdd4
        rdd4=rdd3.reduceByKey((x,y)=>x+y)
        ^

scala> val rdd4=rdd3.reduceByKey((x,y)=>x+y)
rdd4: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:33

scala> rdd4.collect()
res3: Array[(String, Int)] = Array((learning,2), (about,1), (am,3), (on,1), (love,1), (i,6), (big,3), (enjoy,1), (will,1), (data,3), (meant,1), (top,1), (confident,1), (crack,1), (for,1), (my,1), (product,1), (great,1), (based,1
s,1), (companies,1), (passionate,1))
```

SPARK PART 2:

What is "checkpointing" in spark?
-->To save the RDD state from lineage graph into a storage so that resiiency of the RDD is achieved.

What are stages in spark?
-->Each job triggers multiple no. of tasks which can be considered as a stage.


Data Flair
Submitting a job triggers execution of the stage and its parent stages
Action
Job
JobListener
Stage
Stage
Stage
Stage
Stage
Stage
Stage
.....

SPARK STAGES:

1. shuffleMap stage: It is an intermediatary stage in DAG which provides data for another stage ie shuffling of data takes place.
2. ResultStage: It produces the output for the "action" called which is the final result given by the spark

Broadcast Variables:

-->It contains a list of values (small data) which is broadcasted ie **distributed to all the systems** for a specific purpose to be met.

Problem: From above ex. We can see redundant words (in, the, a etc) which are not actually not very helpful.
We collect a list of such words and broadcast them and filter them out from using "filter" function.

-->**similar to map side join in hive**

What are Accumulators?
-->It is a shared copy of value kept in driver machine.
-->It is updated by executors but none can read them
-->**Similar to counter in map-reduce**

How MR jobs were managed before YARN ?

-->Map-Reduce jobs were handled by Job Tracker and Task tracker.

-->Job Tracker:

1. Clients submit the requests to it which it then scheduled to be distributed to slaves.
2. It **schedules** the jobs, priortizes jobs and allocates resources to them
3. It **monitors** the performance of jobs
4. It single handedly takes care of all these tasks

-->Task Tracker:

1. Inside each block where we have tasks to be performed, it is tracked by task tracker

Limitations:

1. **Scalability**: If cluster size increases then load on Job Tracker will be high and its performance dips.
2. **Resource Under-Utilization** : Instances where MR slots are fixed and they are not used to their full potential
3. **Only MR jobs** can be handled (no graphic processing etc)

Why YARN was introduced?

YARN (Yet Another Resource Negotiator)

-->As the job tracker has to manage huge no. of tasks and resources which it failed to perform well. We introduced a separate component just to schedule and manage resouces which is YARN

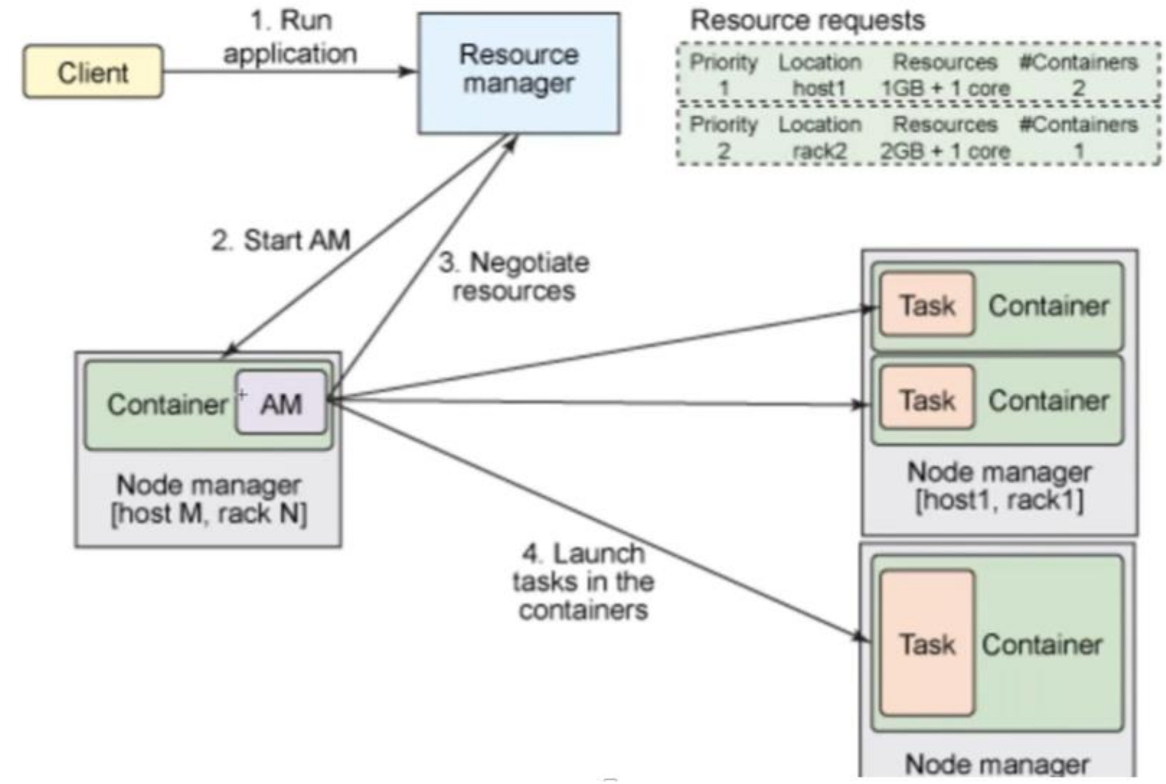
-->Components of YARN:

1. Resource Manager : Does only **scheduling** (load is now reduced)
2. Application Master: Does the **monitoring** and negotiates resources
3. Node Manager: **Manages all the tasks** which are assigned to it.

Architecture:

1. Client submits the requests to run its programs to "Resource Manager"
2. **RM creates a container (memory + cpu) and AM** on a node to take care of the job
   -->**Each job will have an AM**
3. AM will request RM for resources (containers) to run its tasks
   -->RM will have a priority chart based on which it allocates
   -->RM sends container location to AM

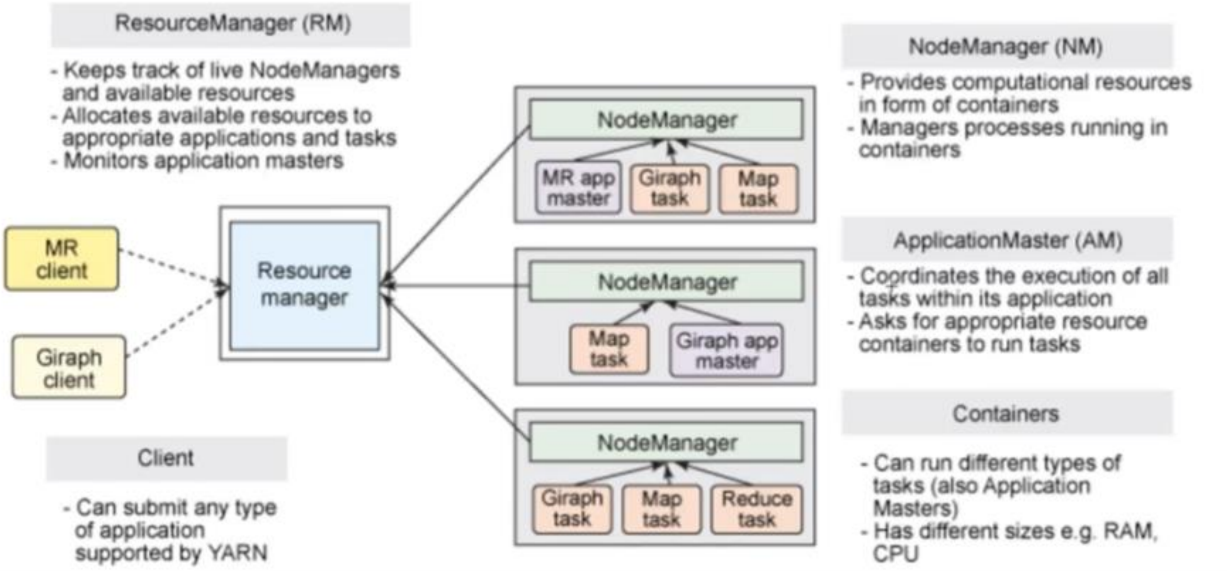
4. AM will then start its tasks in containers it got from RM
5. Tasks monitoring is done by AM and handles issues if any
6. Results from tasks are given to node manager



Resource requests

Priority	Location	Resources	#Containers
1	host1	1GB + 1 core	2
Priority	Location	Resources	#Containers
2	rack2	2GB + 1 core	1

Figure 3. Architecture of YARN


ResourceManager (RM)
- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters
MR client
Giraph client
Resource manager
Client
- Can submit any type of application supported by YARN
NodeManager
MR app master
Giraph task
Map task
NodeManager
Map task
Giraph app master
NodeManager
Giraph task
Map task
Reduce task
NodeManager (NM)
- Provides computational resources in form of containers
- Managers processes running in containers
ApplicationMaster (AM)
- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks
Containers
- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

How Limitations were overcome ?

1. Scalability : Each job can have one AM and no. of jobs can be easily increased
2. Resource Utilization: No fixed MR slots and containers can provide dynamic allocation
3. Apart from MR job we can run spark, giraph jobs as well

What happens when job is very small?
Uberization:

-->AM will not request for additional resources from RM
-->It will run the job in its own container

YARN on Spark Cluster:

How to run sparks jobs on spark cluster?

1. Interactive shell: spark-shell

2. Submit jobs: we write code and run jobs using Eclipse

How spark jobs run on spark cluster?

It follows master-slave architecture

1. Master (Driver) : Schedules, monitors, distributes tasks
2. Slave (Executor) : Executes code locally

Client(sends request) --> Driver -->Executors (
)

Which component executes Where?

-->**Executors** always run on cluster machines (data nodes, **in-memory**)

-->Driver runs on either client (client mode) or cluster (cluster mode)

-->**Cluster mode is preferrable** as it doesn't depend on client machines (mainly real-time production)

Who controls the spark cluster?

It can be done by few of below "cluster managers"

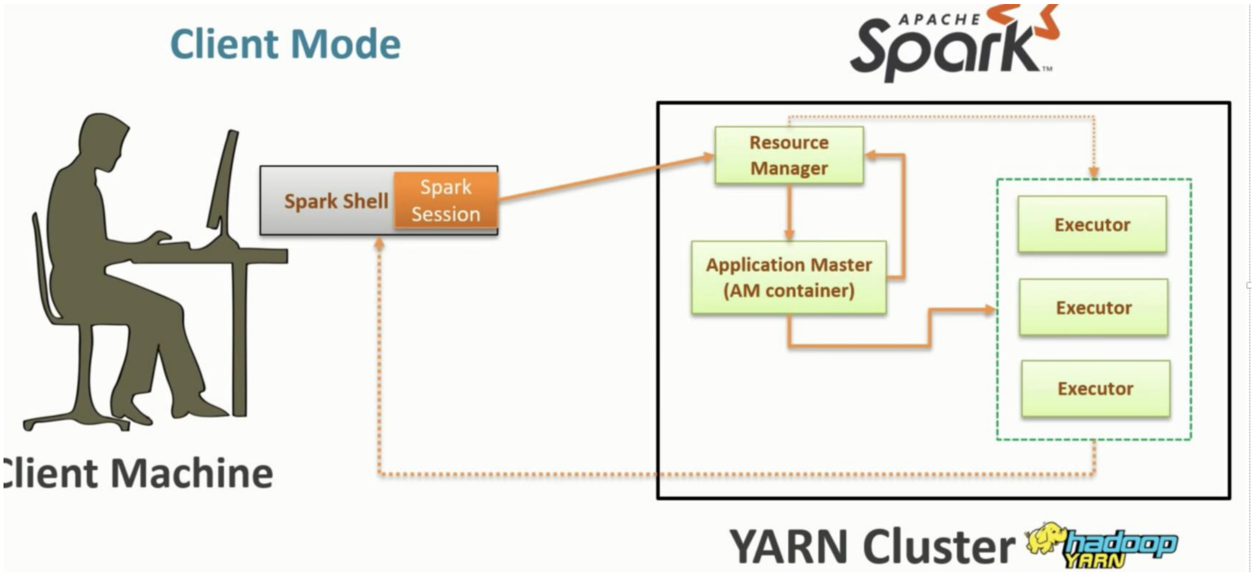
1. YARN , Mesos, Kubernetes

YARN Architecture in Spark:

1. Spark session : Entry point into spark cluster where **driver provides location of executors** where processing will happen

Yarn in client mode:

1. When spark job launched then Spark session created
2. Then Yarn RM creates AM on node manager
3. AM requests resources if req and launch executors in them
4. Driver and executors can communicate directly



Yarn in cluster mode:

1.The **only difference is that Driver runs on AM**

How to convert collections into RDD?

-->when we have Array, List, Set of values.

-->we can use "**parallelize**" method to convert them into RDD

-->val listOrder=List("mobile", "garment","purse")

sc.parallelize(listOrder)

What are types of transformations?

Narrow Transformations: **Data shuffling is not required** ie moving data from one machine to another

Ex. Map(), flatMap() , filter()

-->when we perform map(), all the data resides in one machine and we need not move it across various machines.

Val a="Vaishu is a big data developer" map(a=>a.split(" ") ==> Array[vaishu is a big data developer]

Wide Transformations: **Data shuffling in involved** ie data is moved across machines.
-->It works mainly on pair RDD
Ex. reduceByKey, groupByKey()

P1=({vaish,1},{big,1},{love,1})
P2=({love,1},{vaish,1})
|
|
P1={{vaish,2},{big,1}}
P2=({love,2})
Now, data has to be moved across the partitions to compute the final result

What are stages in spark?
-->Based on DAG we can see the stages of execution in spark.
-->Each **wide transformation invokes a separate stage** as data shuffling is required
-->If we have 2 stages then we have 1 wide transformation for sure
-->Stage 1: Processes the input and writes it to the disk
-->Stage 2: From disk wide transformation takes it and gives the final output

Difference btw reduceByKey() and reduce() :

-->reduceByKey() : It is a **transformation** ie one rdd is converted into another.
-->It works on only **pair rdd**
Pair RDD: It has tuple of 2 elements Ex. (vaish,1) ==>like (key,value) pair

-->reduce() : it is an **action** ie rdd is converted into local variable.

-->It gives us a **single output value**.
Ex. Val a=1 to 100 a.reduce((x,y)=>x+y)
Output:5050 (it just gives sum of all numbers)

reduceByKey()	**Reduce**()
1. It is a transformation 2. It works on only pair RDD 3. It gives multiple output values	1. It is an action 2. It gives a single output value

Why reduceByKey() is transformation and not action?

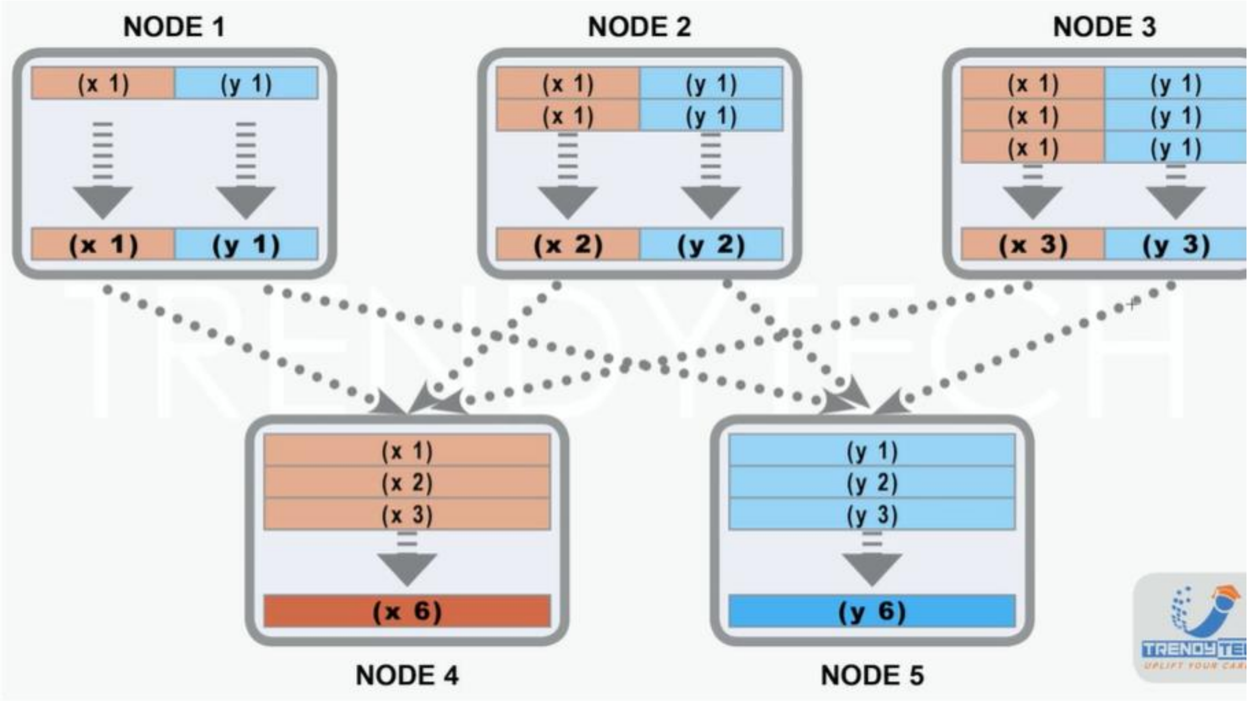
-->reduceByKey() gives us a list of output and not a single output
-->It can be used again for other transformation

reduceByKey() vs groupByKey() :

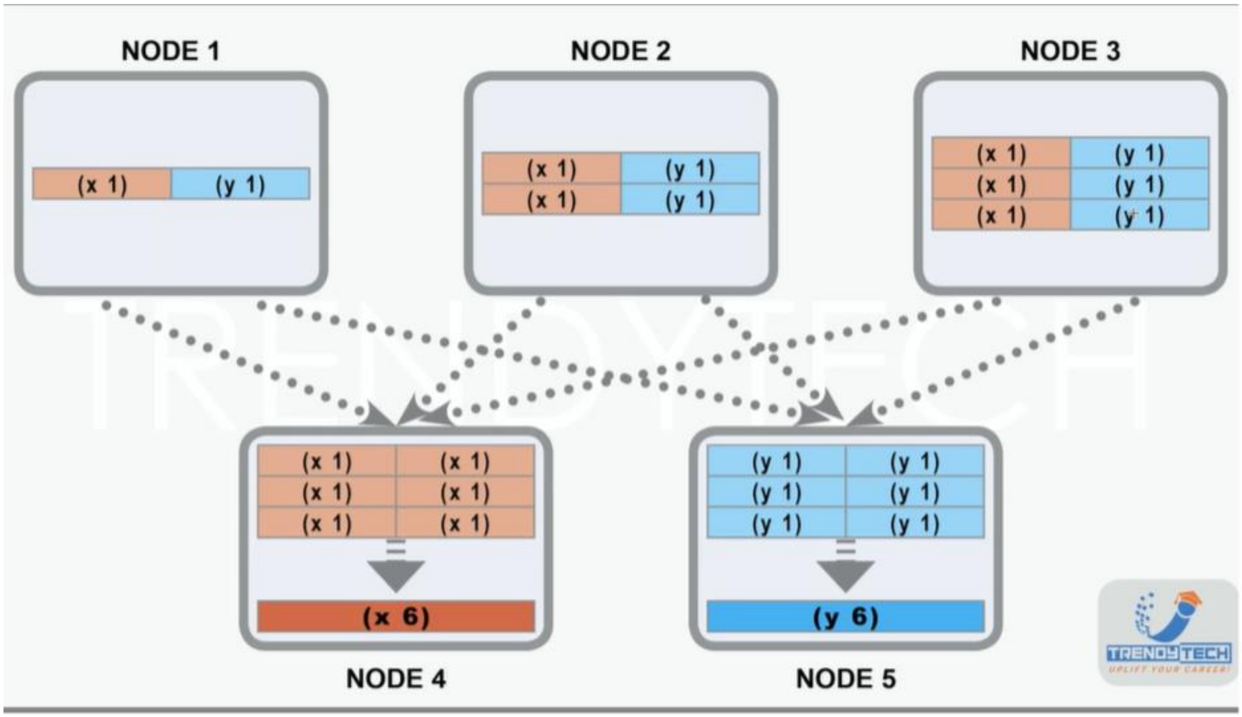
reduceByKey()	groupByKey()
1. **Each partition will process the data** and give the output to reducer machine 2. **Less processing** at reducer	1. **Each partition will just group the similar inputs** and give it to reducer machine 2. **More processing** at reducer 3. Less parallelism 4. Does not perform local aggregation

3. More parallelism 4. Performs local aggregation	5. If data is huge then reduce can't single handedly work on it due to which we should never use groupByKey()

reduceByKey():

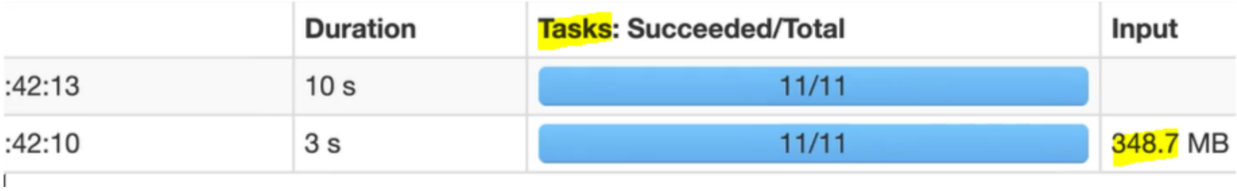


groupByKey:



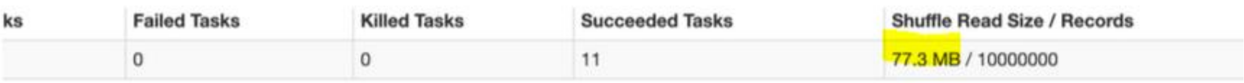
Practical: groupByKey()

File size ~ 350mb   Tasks=11 (350/12)

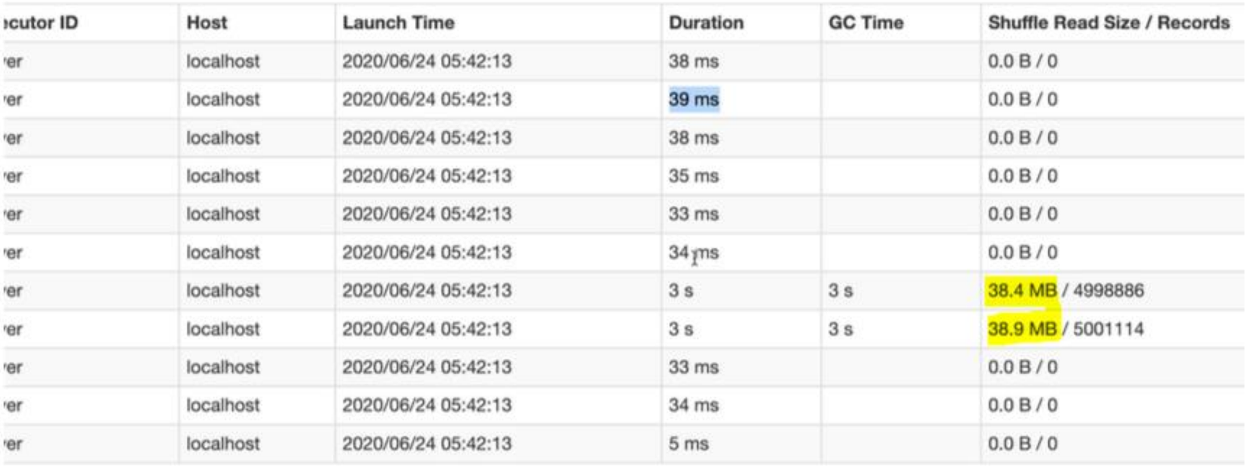


	Duration	Tasks: Succeeded/Total	Input
:42:13	10 s	11/11	
:42:10	3 s	11/11	348.7 MB

-->File compressed and sent to reducers. As we can see **only 2 reducers are doing all the work and all others are idle**. It is not an idle situation as we need to make the best use of all tasks. Due to this groupByKey() is **not recommended**



ks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records
	0	0	11	77.3 MB / 10000000



cutor ID	Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records
er	localhost	2020/06/24 05:42:13	38 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	39 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	38 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	35 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	33 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	34 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	3 s	3 s	38.4 MB / 4998886
er	localhost	2020/06/24 05:42:13	3 s	3 s	38.9 MB / 5001114
er	localhost	2020/06/24 05:42:13	33 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	34 ms		0.0 B / 0
er	localhost	2020/06/24 05:42:13	5 ms		0.0 B / 0

What is a Pair RDD:

1. **RDD with a tuple of 2 elements**
   ("vaish",10) (20,30)
2. Pair RDD is not a map as map can have only unique keys
3. It works on transformations like reduceByKey() and groupByKey()

What happens when multiple actions are called?

Val mapIn=X.map(x=>x,1)
Val redIn= mapIn.reduceByKey(x,y=>x+y)

Val resultFinal=redIn.**collect**() ==> action 1 (processes entire data)
Val finalResult=redIn.**count** ==>action2 (spark stores the results upto last transformation and uses it directly)

-->Every **action** triggers a **job** in spark
-->Every job processes the data right from beginning

-->** But spark optimizes it and **holds the results of processing done previously** which can be used for the next action directly rather than processing from the beginning.

Properties in spark:

1. **For data collections**

1.1 **sc.defaultparallelism** => It gives us the default no. of partitions created for the rdd's

1. For Hdfs data

2.1 **sc.defaultMinPartitions** => It determines min no. of partitions returned by rdd
Ex. File~ 100mb (actually 1 partition)
But due to above property it returns 2
2.2 **sc.getNumPartitions** => It gives no. of partitions for a particular rdd
This property takes precedence over defaultMinPartitions

1. **Repartition vs Coalsence**

Repartition	Coalesce
1. Used to **either increase/decrease no. of partitions for rdd's** Val newRdd=inputFile.repartition(10)	1.Used to **only decrease no. of partitions** 2.If we increase partitions no changes will be made and same old value remains

Val newRdd=inputFile.repartition(1) 2. It is a **wide transformation** as data shuffling is involved	Val newRdd=inputFile.coalesce(2)
1. Use repartition for **INCREASE** of partitions	1. Use coalesce for **DECREASE** of partitions It combines partitions on same machines due to which shuffling is less and optimized Before: Node1: p1 p2 Node 2: p3 p4 Node3: p5 p6 After: Node1: p1(p1+p2) Node2: p3(p3+p4) Node3:p5(p5+p6) 2. Don't use **REPARTIION** for DECREASE Repartition tries to **equally distribute the data** among all partitions due to

	which more shuffling is required

**

1. No. of actions = No. of jobs= No. of AM
2. No. of tasks= No. of partitions
3. 1 block size in local = 1rdd partition=32mb
4. 1 rdd= n partitions
5. 1 task = 1 partition