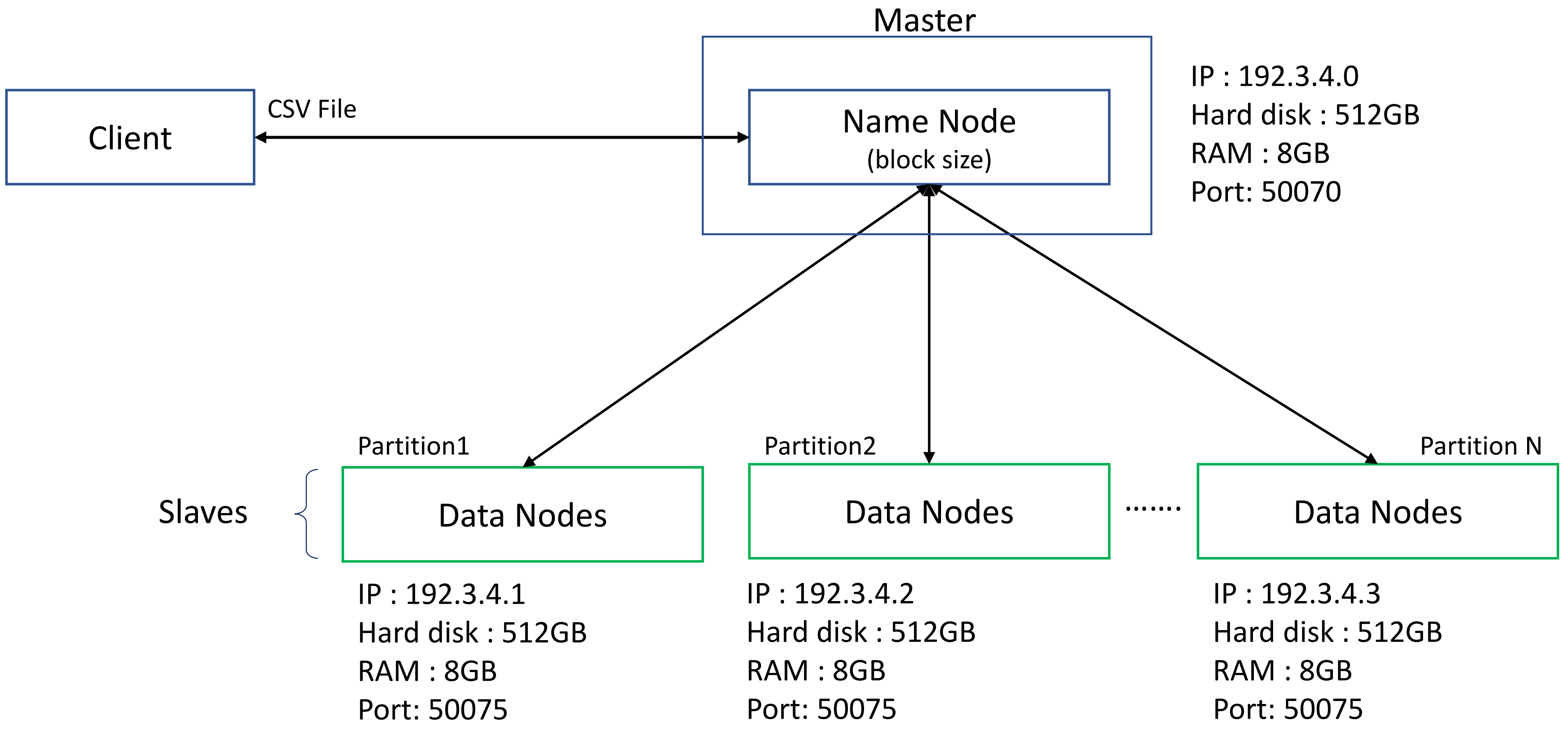
Recall HDFS



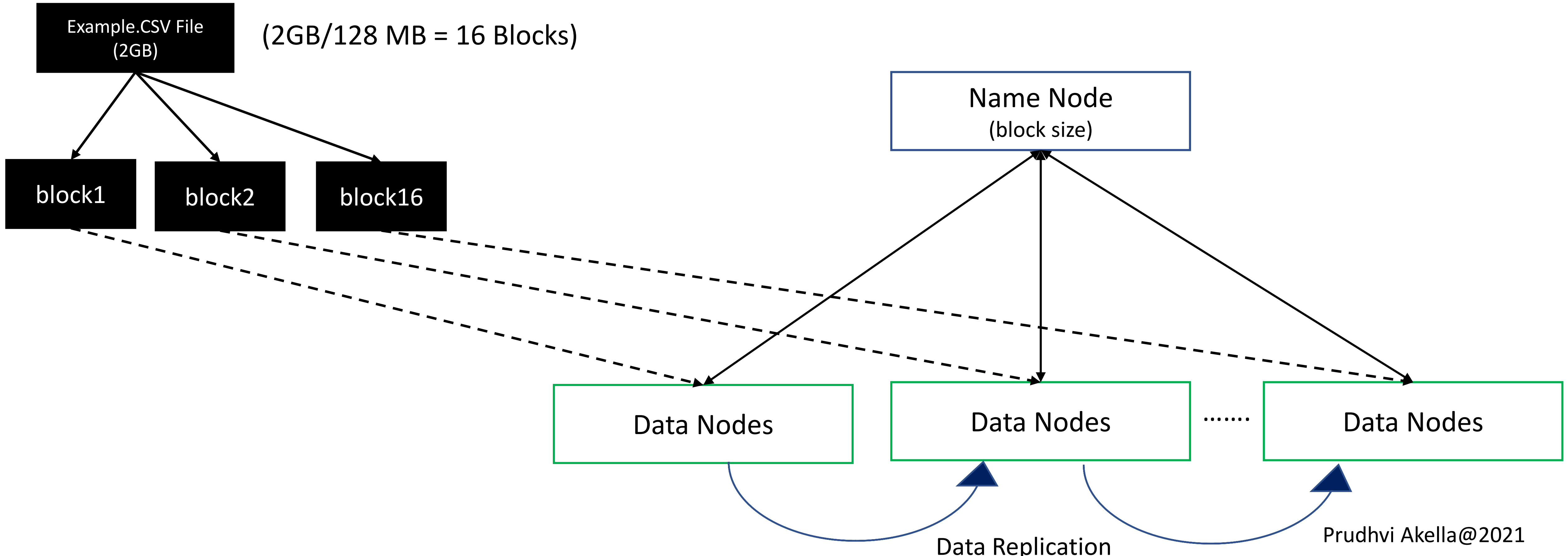
Note: Name Node and Data Node are the JVM process and they will in a dedicated server with a dedicated port number(Name Node: 50070, Data Node: 50075)

Prudhvi Akella@2021

Recall HDFS

There are three important parameters which will control the behaviour of HDFS
Block size = 128MB, Split size = 256MB(By default split size is equal to block size), Replication Factor = 3

When you try to copy the csv file of size 2GB into HDFS. The file will be splitted into blocks based on the block size and each block will be stored into each data node and its further replicated into data node to achieve the fault tolerance. The metadata information of the file is stored into NameNode



Recall HDFS

Name Node : It records the metadata of all the files stored in the cluster, such as location of blocks stored, size of the files, permissions, hierarchy, etc.

In the above example CSV file is divided into 16 blocks and each block is stored into different data nodes.
So now lets say imagine Spark or MapReduce wants to read and process that CSV file then they will reach out name node for the meta data information using that they will distribute the work to executer(spark) or mappers(MR).

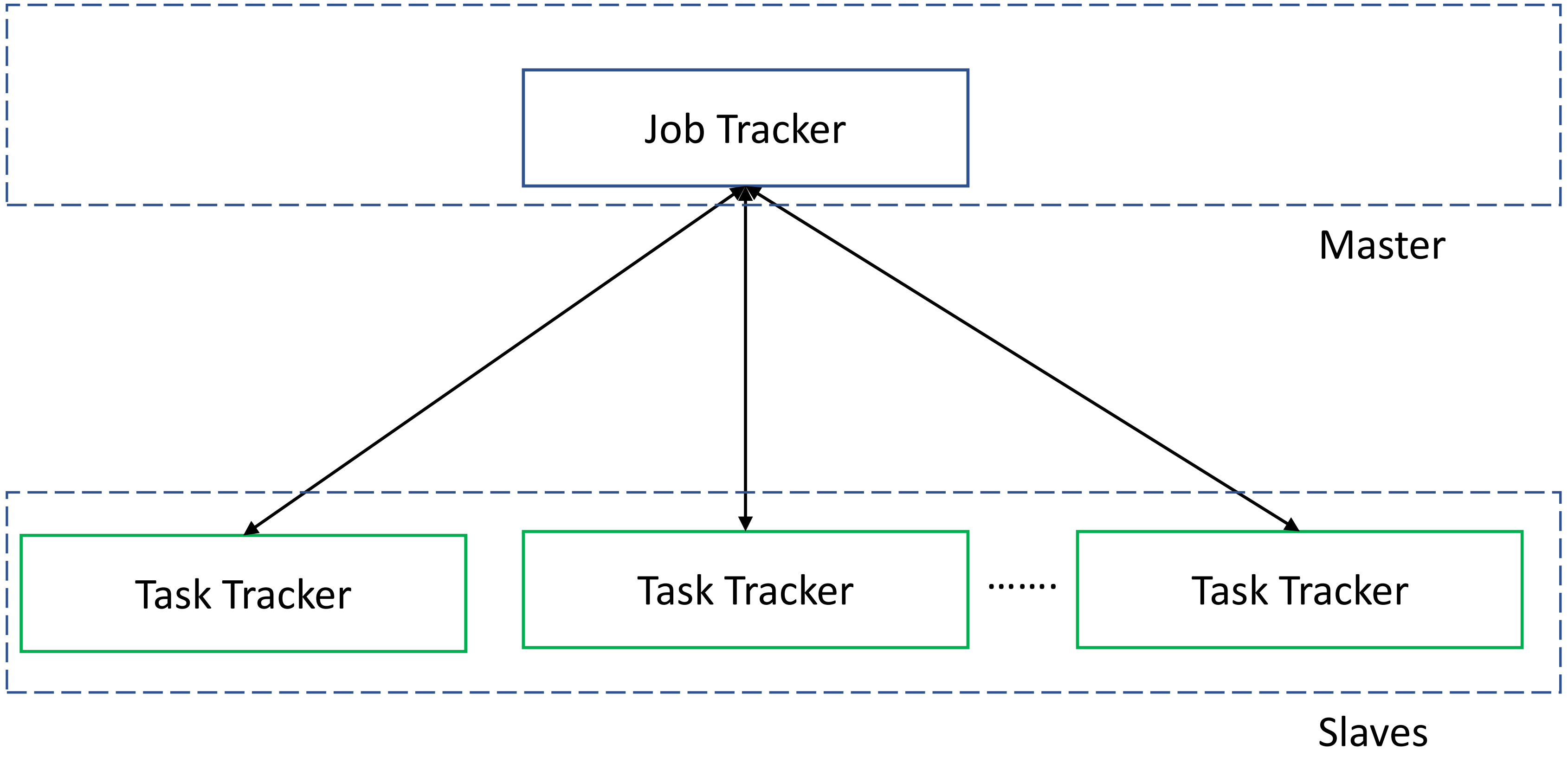
File Name	Part Files	Data Node
Example.csv	part1.csv,part2.csv,part3.csv,part4.csv	DN1
	part5.csv,part6.csv,part7.csv,part8.csv	DN2
	part9.csv,part10.csv,part11.csv,part12.csv	DN3
	part13.csv,part14.csv,part15.csv,part16.csv	DN3

Data Node IP:Port	Alias
http://192.3.4.1:50075	DN1
http://192.3.4.2:50075	DN2
http://192.3.4.3:50075	DN3

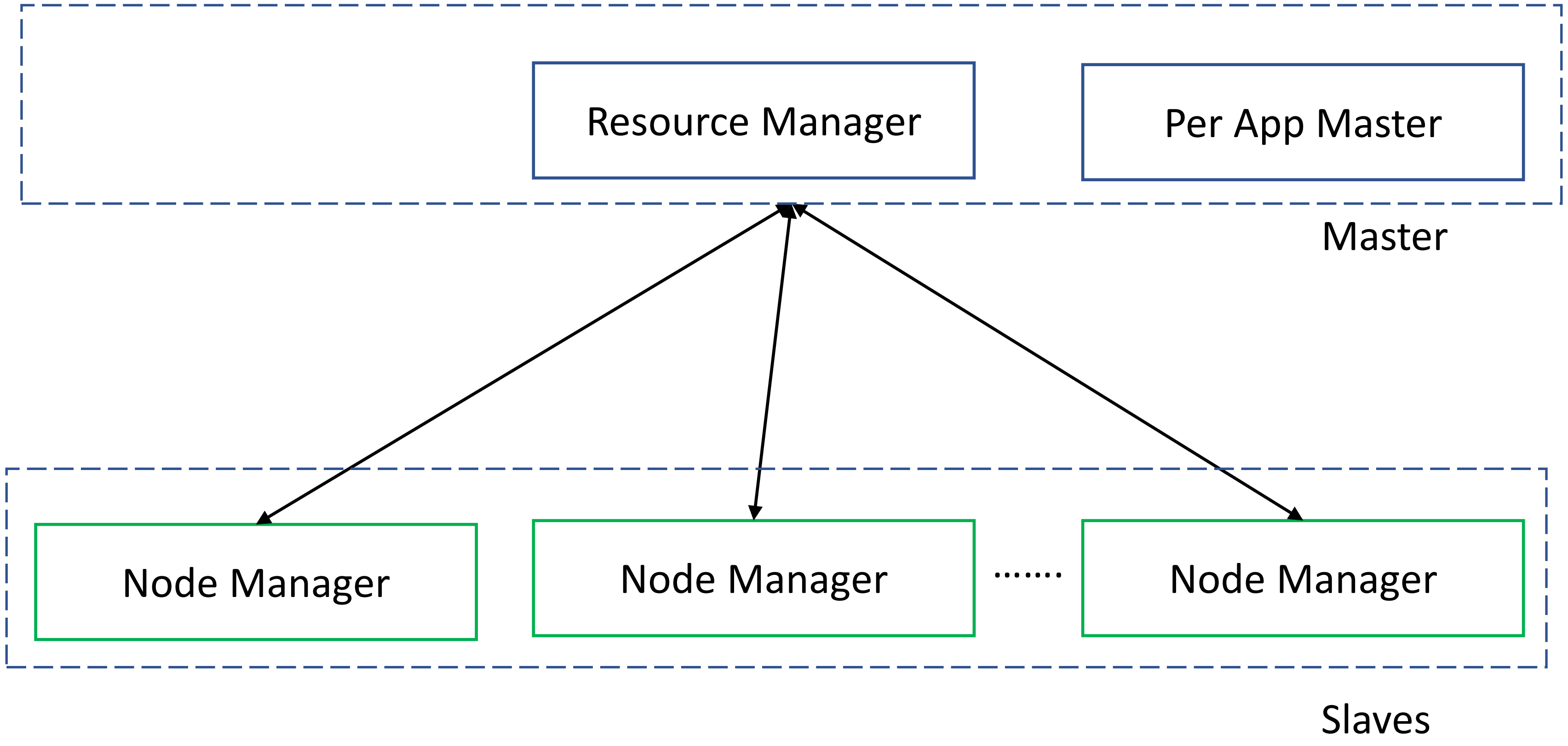
Note: For each HDFS Block one new part-<block number> will be created and data of that block will be in that part file.

For Example.csv after copying into HDFS 16 part files will be created for 16 blocks

Recall Map/Reduce 1



Recall YARN(Yet Another Resource Negotiator)(MR 2)



Map/Reduce at High level

At a high level Map-Reduce has two phases in it Mapper and Reducer . Mapper are used to perform transformations and Reducers are used to perform the aggregations.

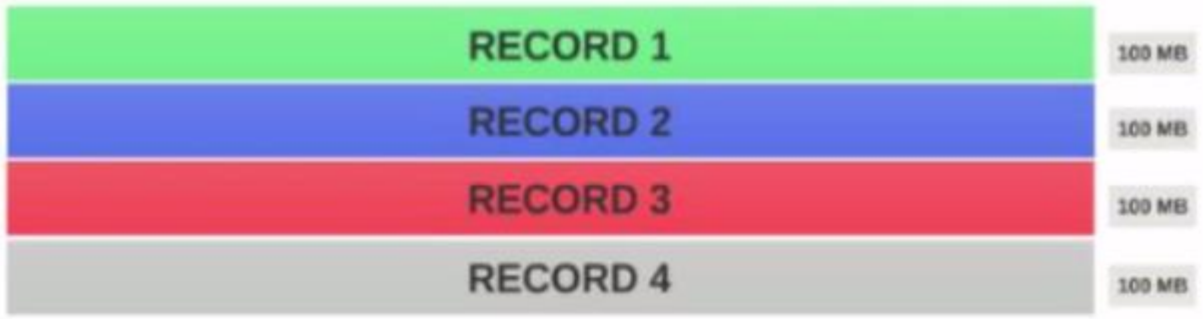
How many mapper will be launched to process on file?
Number of Mapper is completely depends on the input split size.
By defaults Input-Split size = Block Size

In CSV file case that we discussed in “Recall HDFS” created 16 blocks for 2GB file for that it creates 16 Mappers jobs. One mapper completes the transformations it will store that intermediate data into the disk by default those intermediate files are compressed using Bzip2 Codec. Those intermediate mapper output files will be input(shuffled) to the reducer job it will be decompressed and then reducer job will perform aggregation for each key.

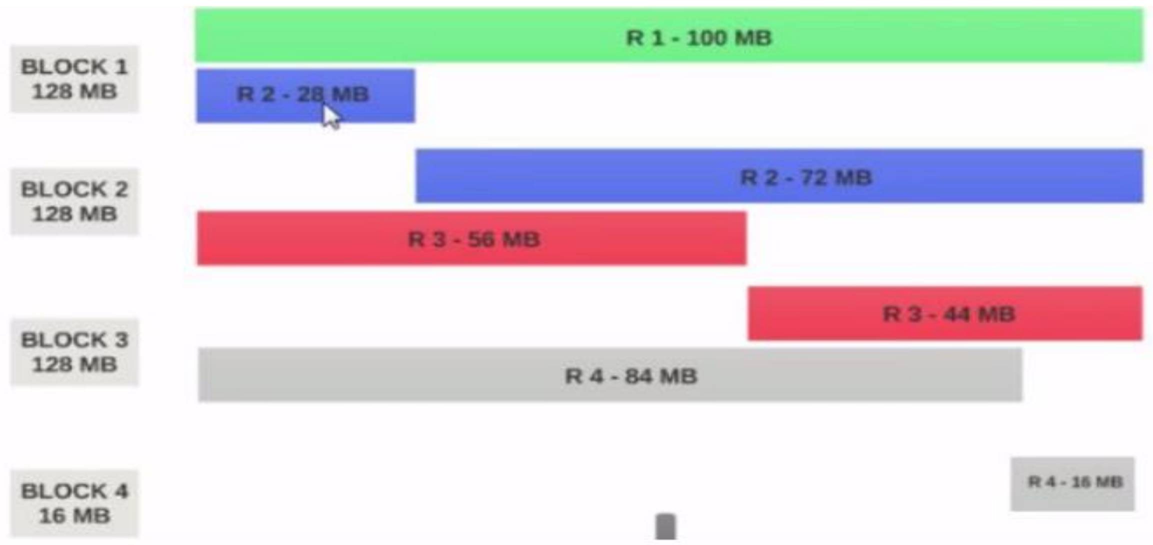
Number of Reducer job is all based on “mapred.reduce.tasks” configuration we set.

When the input split really matters

Assume we have a file of 400MB with consists of 4 records(e.g : csv file of 400MB and it has 4 rows, 100MB each)



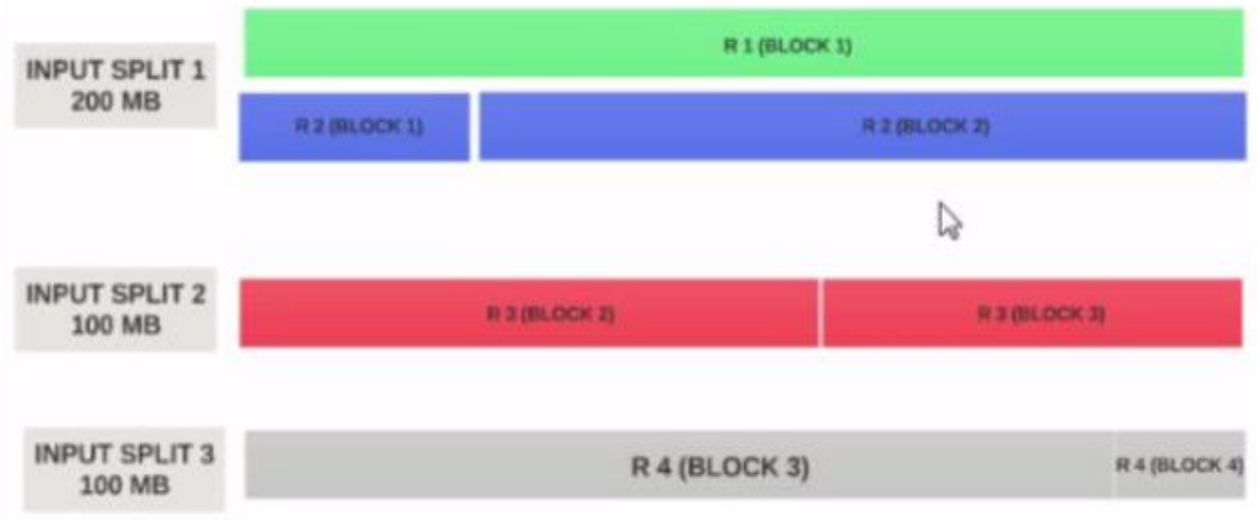
If the HDFS Block Size is configured as 128MB, then the 4 records will not be distributed among the blocks evenly. It will look like this.



•**Block 1** contains the entire first record and a 128MB chunk of the second record.
•If a mapper is to be run on **Block 1**, the mapper cannot process since it won't have the entire second record.
•This is the exact problem that **input splits** solve. **Input splits** respects logical record boundaries.

When the input split really matters

•**Lets Assume the input split size is 200MB**



•Therefore the **input split 1** should have both the record 1 and record 2. And input split 2 will not start with the record 2 since record 2 has been assigned to input split 1. Input split 2 will start with record 3.

•This is why an input split is only a **logical chunk** of data. It points to start and end locations with in blocks.

•If the input split size is n times the block size, an input split could fit multiple blocks and therefore less number of **Mappers** needed for the whole job and therefore less parallelism. (Number of mappers is the number of input splits)

Py4J

Py4J(A Bridge between Python and Java): Py4J enables Python programs running in a Python interpreter to dynamically access Java objects in a Java Virtual Machine

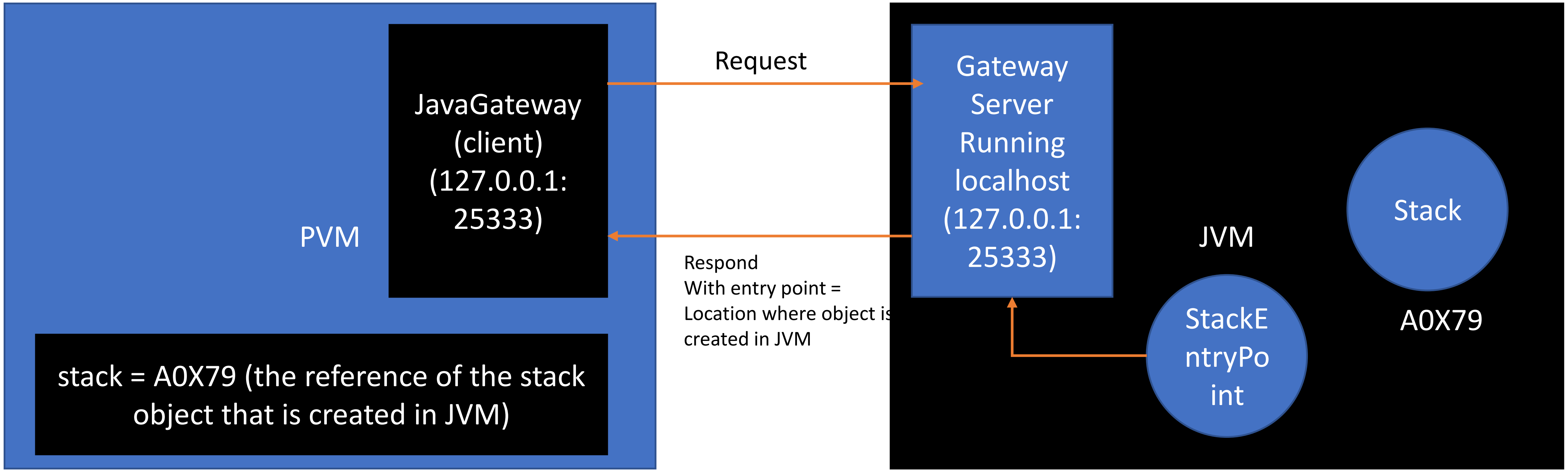
- → Java Gateway:
  - →A JavaGateway is the main interaction point between a Python VM and a JVM.
  - →By Default Python tries to connect to JVM with a gateway on localhost on port 25333.
  - →Technically, a **Gateway Server** is only responsible for accepting connection
- → Entry point
  - → Its a reference to returned objects.

JAVA:

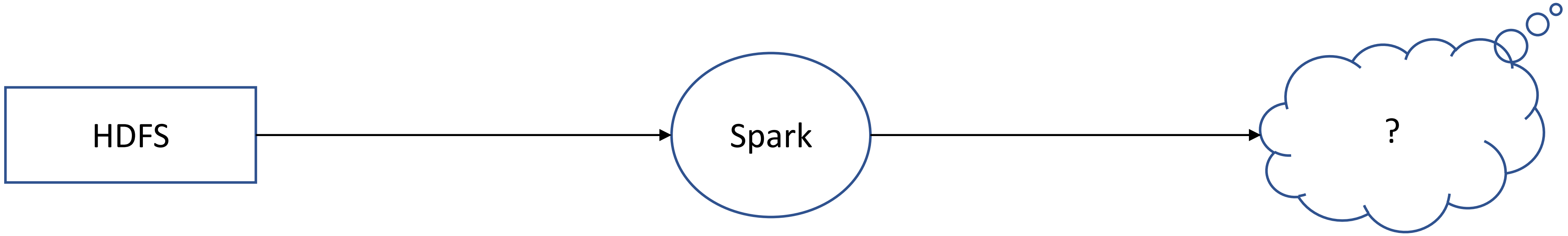
```
package py4j.examples;

import py4j.GatewayServer;

public class StackEntryPoint {

    private Stack stack;

    public StackEntryPoint() {
      stack = new Stack();
      stack.push("Initial Item");
    }

    public Stack getStack() {
        return stack;
    }

    public static void main(String[] args) {
        GatewayServer gatewayServer = new GatewayServer(new StackEntryPoint());
        gatewayServer.start();
        System.out.println("Gateway Server Started");
    }
}
```

Python:

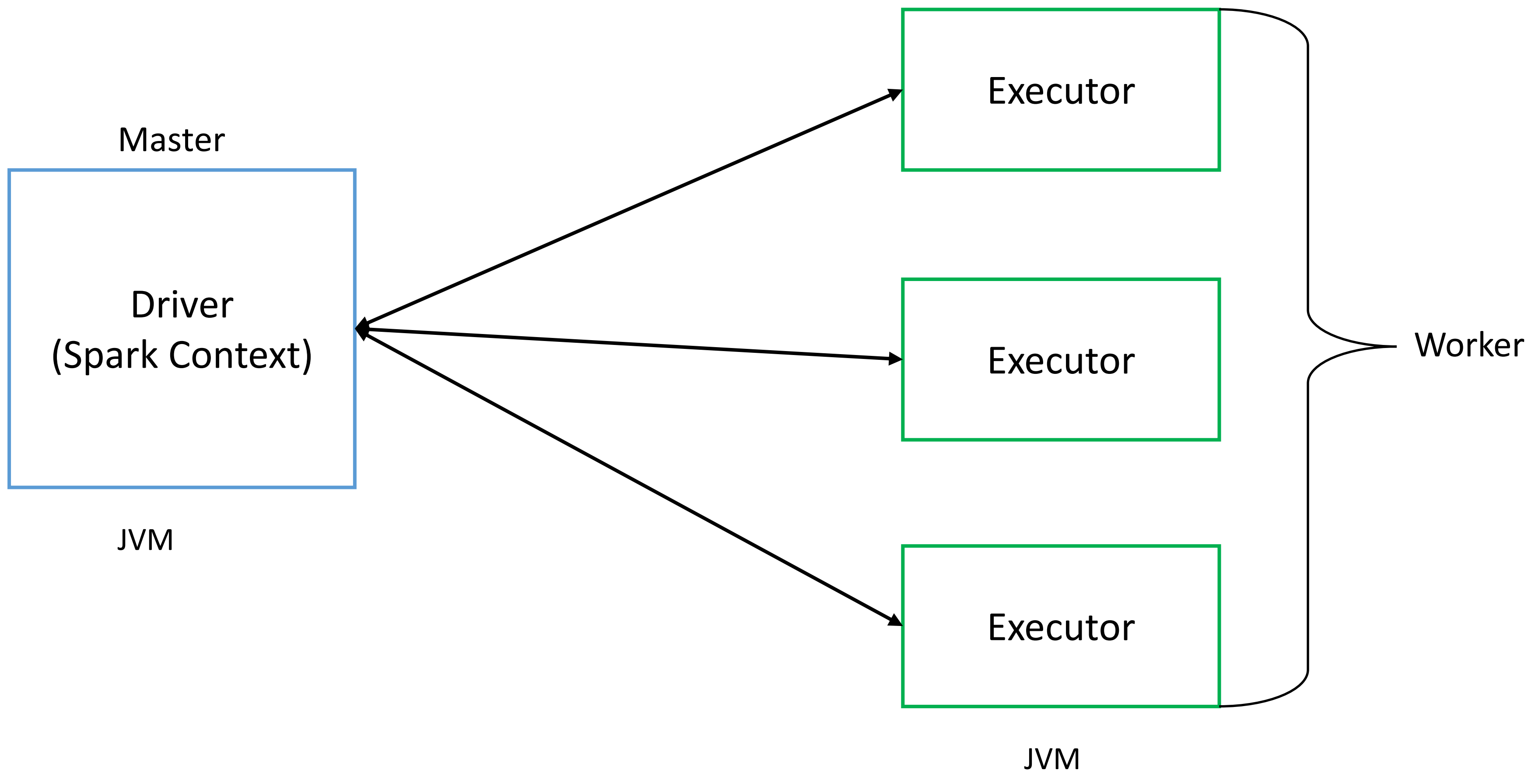
```
from py4j.java_gateway import JavaGateway
gateway = JavaGateway()
stack = gateway.entry_point.getStack()
stack.push("First %s" % ('item’))
stack.push("Second item")
print(stack.pop())
print(stack.pop())
print(stack.pop())
```


JavaGateway (client) (127.0.0.1: 25333)
PVM
stack = A0X79 (the reference of the stack object that is created in JVM)
Request
Respond
With entry point =
Location where object is
created in JVM
Gateway Server Running localhost (127.0.0.1: 25333)
JVM
Stack
A0X79
StackEntryPoint

What Happens when you try to read the file from HDFS?



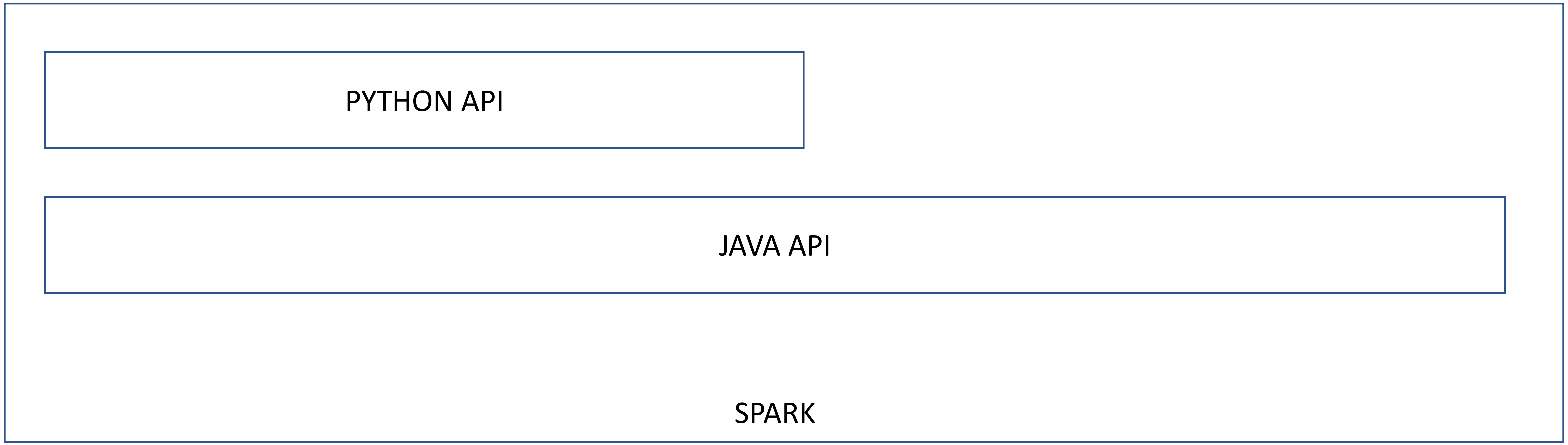
Spark Architecture



- Spark Context acts like channel where driver uses it to communicate with executor.
- Both Driver and Executors are the java process which will runs in there dedicated JVM containers

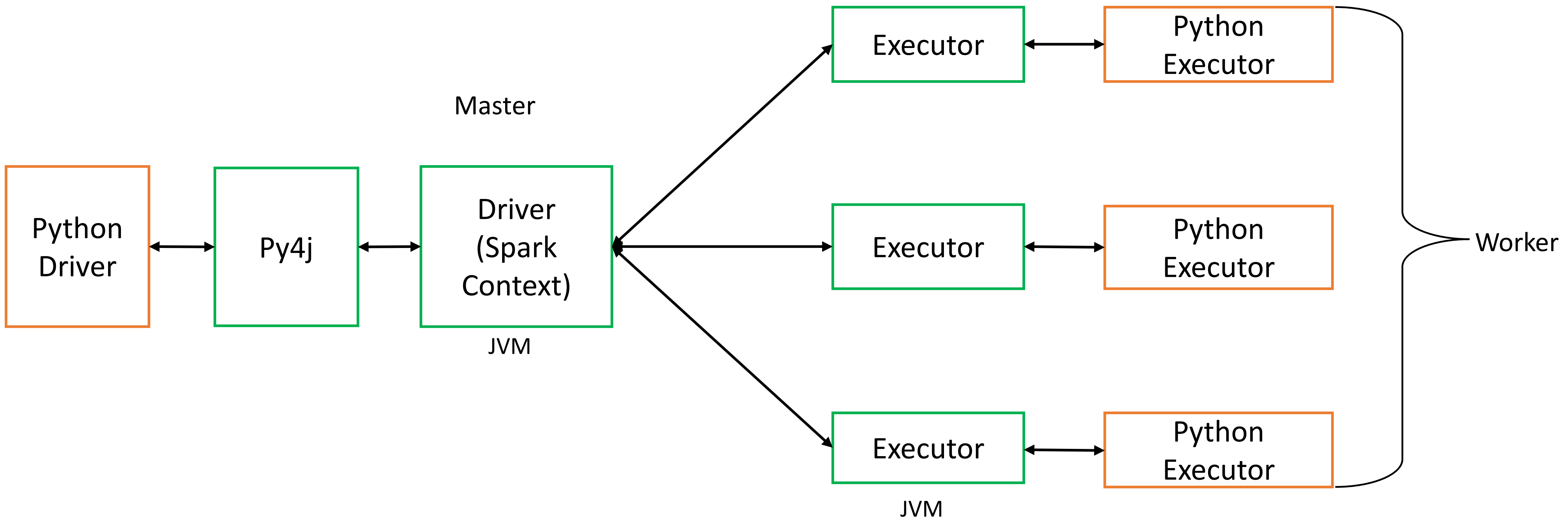
Prudhvi Akella@2021

PYSPARK API HIGH LEVEL



The PySpark API is a “very thin layer” on top of the Java API for spark, which itself is only a wrapper around the core Scala functionality.

Py-Spark Architecture

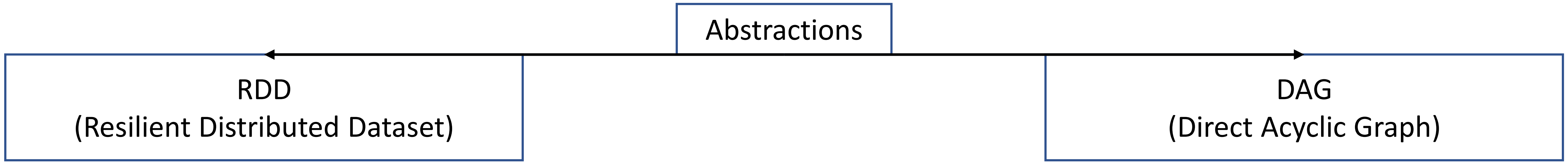


🟩 JVM Process

🟧 PVM Process

Prudhvi Akella@2021

Major Abstractions in Spark



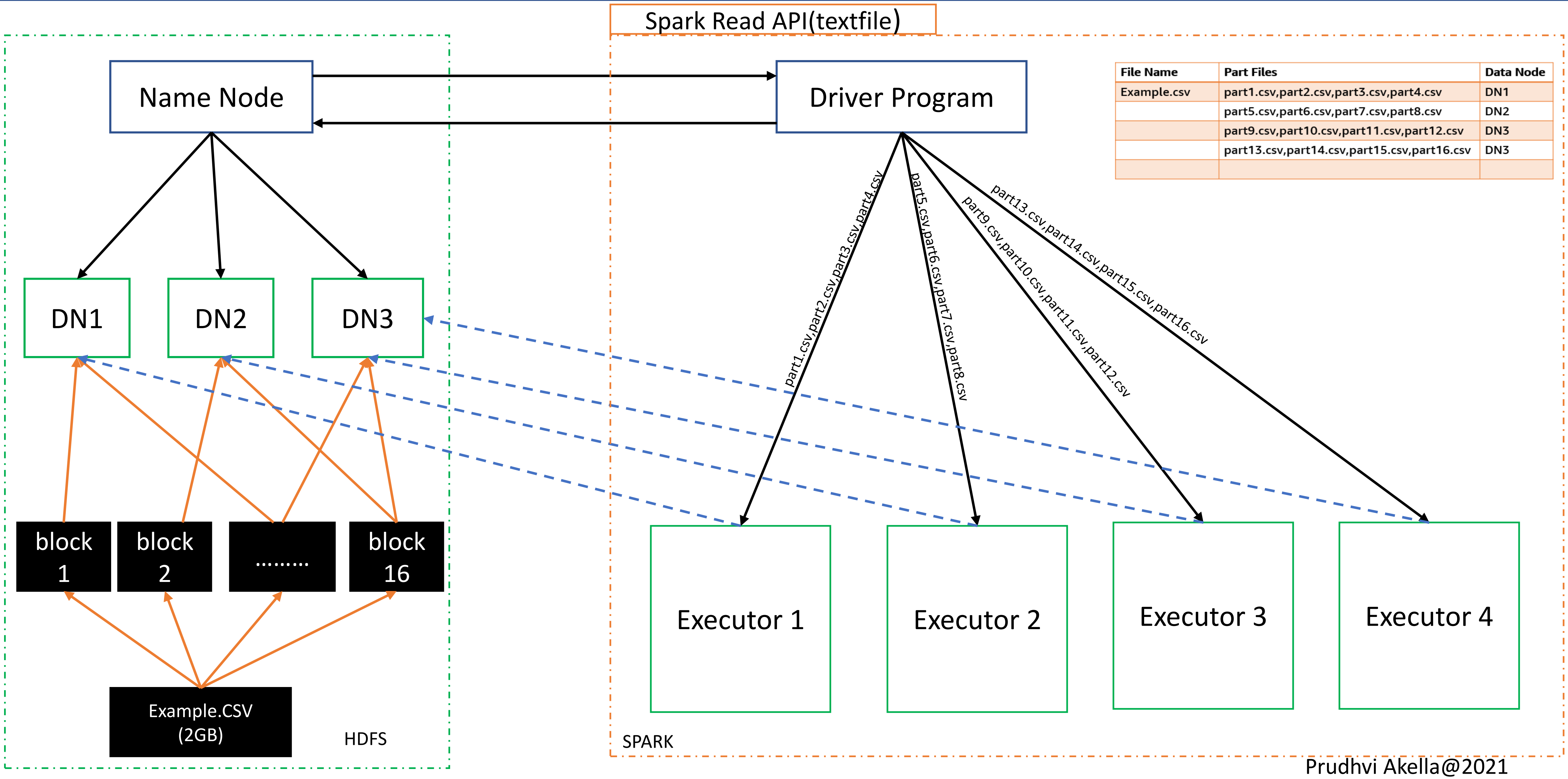
Question your self before processing the data in spark

datastore?(where data is stored)
HDFS,S3,Blob,Local Storage, NFS.

Type of Data?
.PNG, txt, csv, parquet, ORC, json, xml

Codec(compression and decompression algo)?
gzip, bzip2, lzo, snappy

High Level View of Spark Read API From HDFS


Spark Read API(textfile)
Name Node
Driver Program
DN1
DN2
DN3
block 1
block 2
....
block 16
Example.CSV (2GB)
HDFS
part1.csv,part2.csv,part3.csv,part4.csv
part5.csv,part6.csv,part7.csv,part8.csv
part9.csv,part10.csv,part11.csv,part12.csv
part13.csv,part14.csv,part15.csv,part16.csv
Executor 1
Executor 2
Executor 3
Executor 4
SPARK

File Name	Part Files	Data Node
Example.csv	part1.csv,part2.csv,part3.csv,part4.csv	DN1
	part5.csv,part6.csv,part7.csv,part8.csv	DN2
	part9.csv,part10.csv,part11.csv,part12.csv	DN3
	part13.csv,part14.csv,part15.csv,part16.csv	DN3

Prudhvi Akella@2021

High Level View of Spark Read API From HDFS

The whole idea of the spark driver is to distribute the work across the executors(distribute the data) and monitor them.

When you try to read and process a file(Example.csv) from HDFS.

→ Spark driver will make a request to Name Node to get the Metadata information of the file.

→ One the Name Node receives the request from driver it will check whether the request is authorized to serve or not. If the request is valid Name Node will respond back with the Metadata info of the file to Driver

→ One the driver gets the meta information it looks at it and see how many blocks are there for that file to process.

→ Then those blocks info are distributed across the Executors. Once the executor receives the request from driver to process a particular block(It has the address(ip : port no) of data node in which that block/part file is residing) then executor will connect to that Data Node and get the block data into its memory for processing.

Driver needs a efficient way to Monitor, distribute and compute on the distributed data.

RDD(Resilient Distributed Dataset) : Basic Unit Of Parallelism

Resilient : Fault Tolerant and Ability to capable of rebuild state on failure.

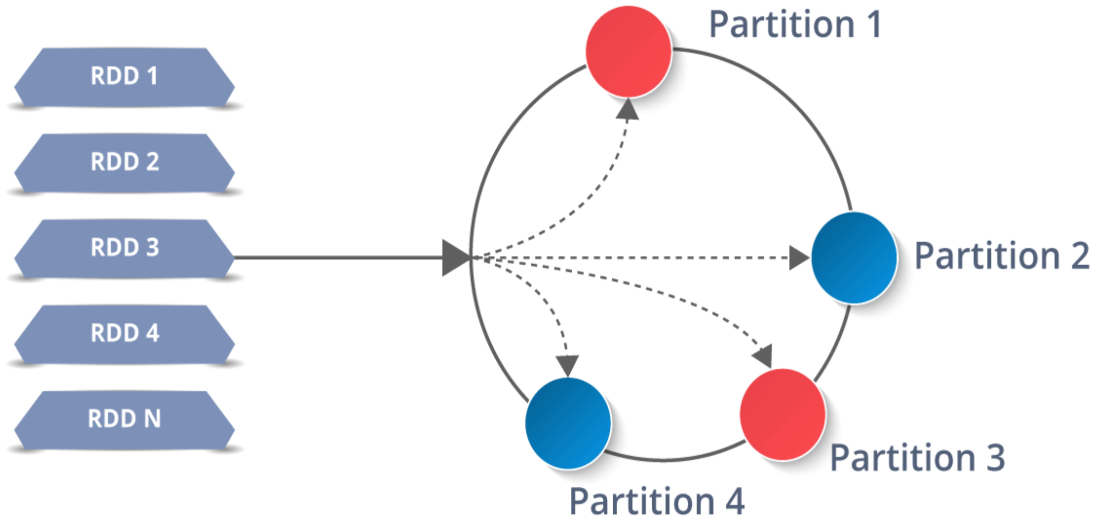
Distributed : Distributed the Data among multiple nodes in the cluster

Dataset : Collection of partitioned data with values

<u>Points to Remember:</u>

→Data in RDD is split of chunk's and each chunk is a partition. RDD's are highly resilient, Even in case of executor fails it has a ability to re-construct back by looking at the lineage.

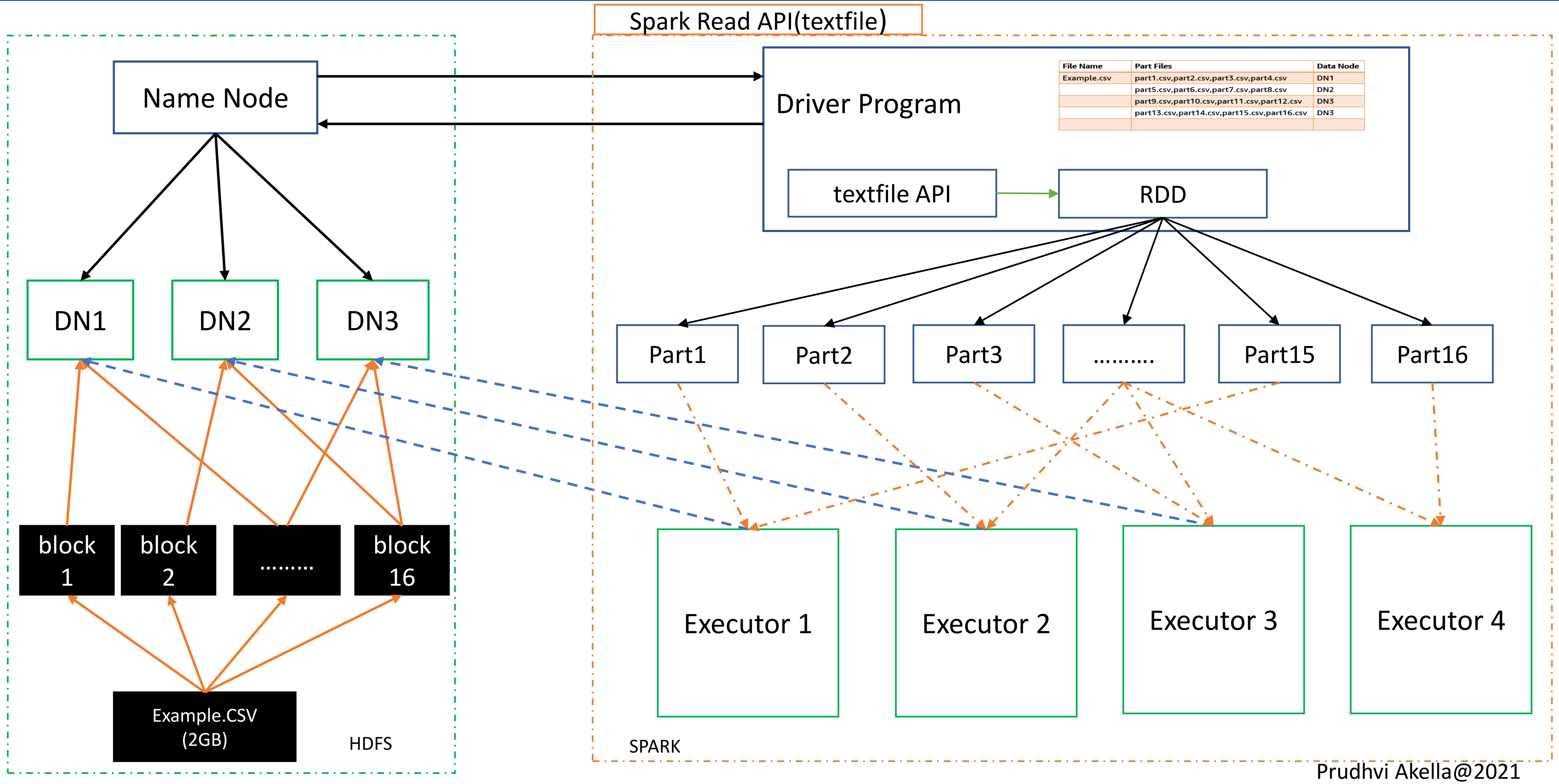
→RDD's are immutable that's means once object is created whose stage cannot be changed but can be transformed.



How many partitions each RDD's will have?
Its depends on the source where we are reading the data from.
Let's say if we reading the file from HDFS
Number of HDFS file Blocks = Number of RDD Paritions

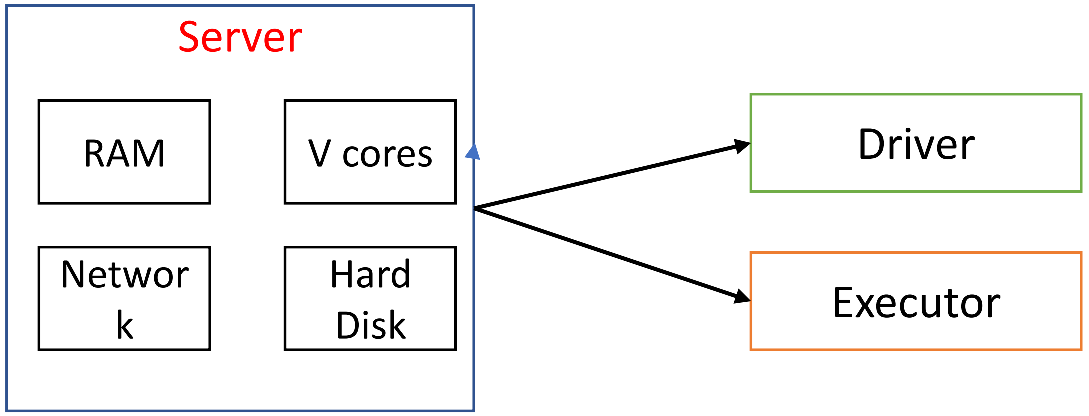
Prudhvi Akella@2021

RDD(Resilient Distributed Dataset)



File Name	Part Files	Data Node
Example.csv	part1.csv,part2.csv,part3.csv,part4.csv	DN1
	part5.csv,part6.csv,part7.csv,part8.csv	DN2
	part9.csv,part10.csv,part11.csv,part12.csv	DN3
	part13.csv,part14.csv,part15.csv,part16.csv	DN3

Prudhvi Akella@2021

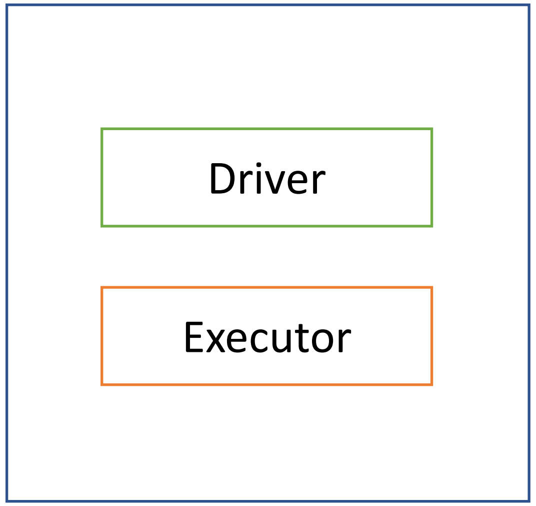


Driver and Executors are the JVM Processes which will be launched in JVM Container . Each JVM container is allocated with some part of RAM, Vcores and Hard Disk in a server.

By default Driver and Executor are allocated with 1GB of the RAM In cluster mode. In local mode driver memory is 1GB(1024MB) and Executor memory is 380Mb .Then can changed and we will discuss in spark memory management section.

Till now we looked at how spark reads the data from HDFS.
From now we will experience the spark in local mode then we will look at cluster mode.

Local Mode

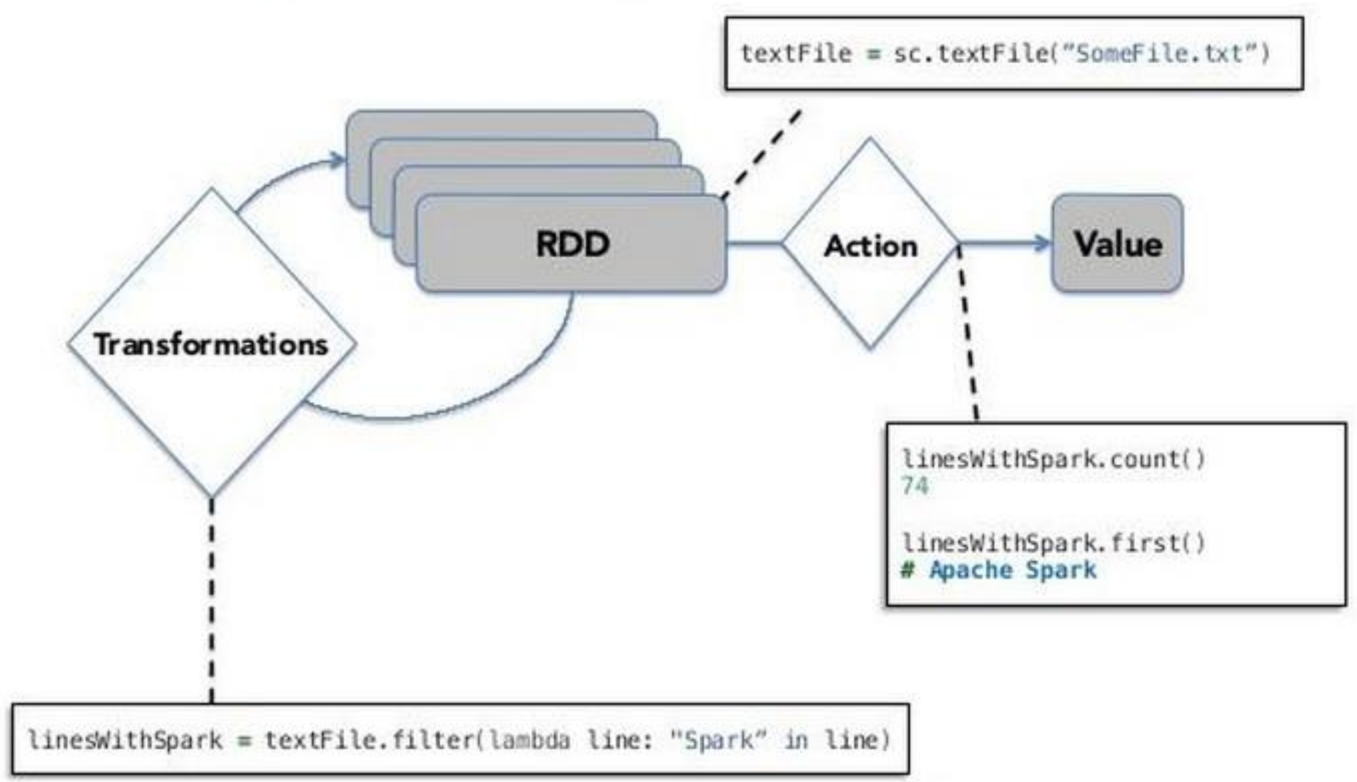


JVM Container

In local mode both driver and executor runs in a same jvm container and you can launch only one executor in local mode.
By Default:
Driver memory = 1GB
Executor Memory is 380Mb

Note: one partition → one task → one core(For every partition as task will be created and gets executed by one VCore

Transformations And Actions



```
textFile = sc.textFile("SomeFile.txt")
```

```
linesWithSpark.count()
74

linesWithSpark.first()
# Apache Spark
```

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

→ Transformations: Takes RDD as an input and returns transformed RDD as an Output.

→ Actions: Takes RDD as an input returns back computed value to the driver.

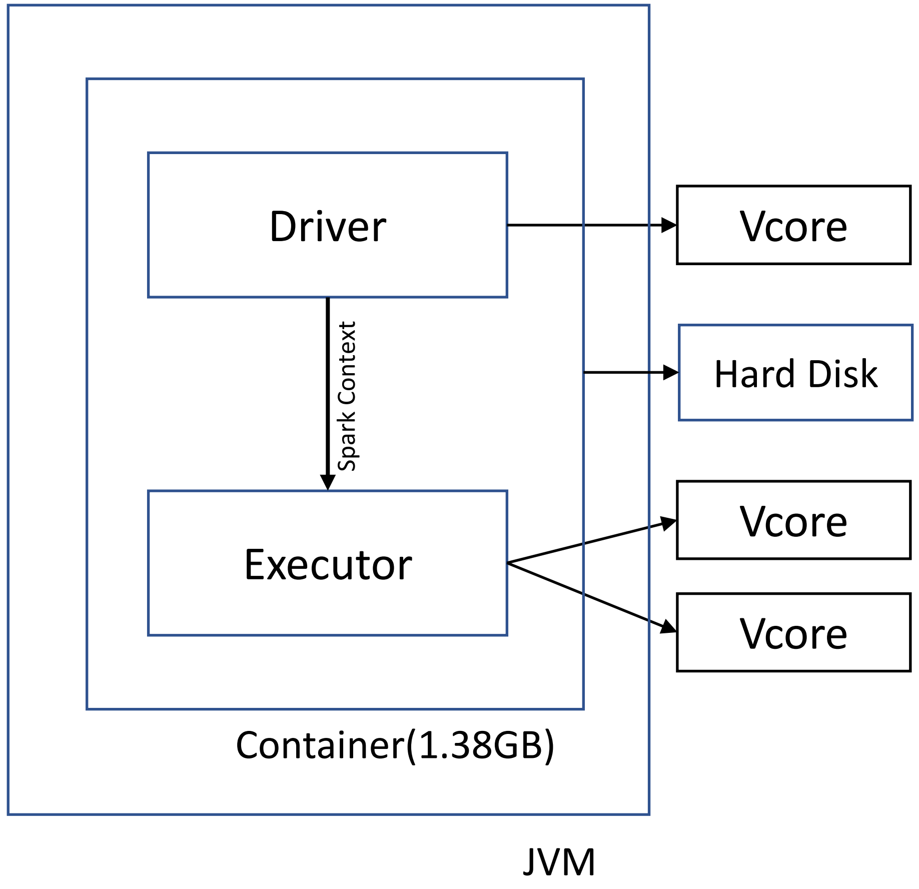
→ Transformations are lazy until and unless an action is performed it will not be evaluated/computed.

https://spark.apache.org/docs/latest/api/scala/org/apache/spark/SparkContext.html#wholeTextFiles(path:String,minPartitions:Int):org.apache.spark.rdd.RDD[(String,String)]

Example1:Word Count

sc = SparkContext("local[2]", "Word Analysis")

SparkContext is a class and its constructor accepts 3 arguments
1) master = local[2]
2) appname = “Word Analysis”
3) Sparkconf = None(Default)



At this line in local the container will created with Driver and Executor process with 1.38 GB of the Memory and 3 Vcores
1 is for Driver and 2 are for Executor. Why only 2? because after local in the square brackets what ever number you see it talks about Vcores.

Example:

local -> 1core
local[2] -> 2Cores
local[*] -> All Available cores.

default.parallelism: defaults to the number of all cores on all machines. The parallelize api has no parent RDD to determine the number of partitions.

defaultMinPartitions: **math.min(defaultParallelism, 2)**. Decides the number of partitions if MinPartitions are not set for Read API’s(textfile , wholetextfile). This means if you don’t set how many partitions you want at the time of reading the file it will always be 2.

Example:
When master = local, default.parallelism = 1,defaultMinPartitions: 1
When master = local[2], default.parallelism = 2, defaultMinPartitions: 2
When master = local[*], default.parallelism = 8, defaultMinPartitions: 2

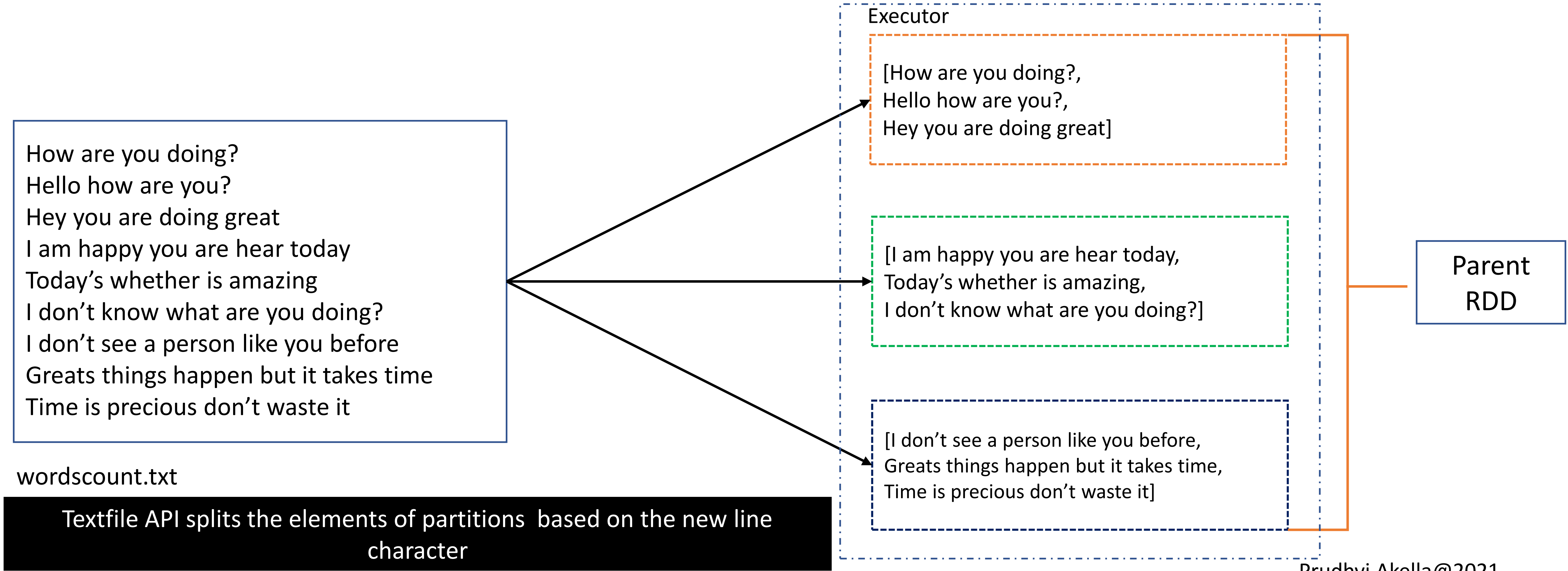
Example1:Word Count

trans1 = sc.textFile("wordscount.txt“, 3)

Arguments
1) Path of the file to process 2) MinPartitions

Type of RDD:

PVM : pyspark.rdd.RDD | **JVM** : MapPartitionsRDD


How are you doing?
Hello how are you?
Hey you are doing great
I am happy you are hear today
Today’s whether is amazing
I don’t know what are you doing?
I don’t see a person like you before
Greats things happen but it takes time
Time is precious don’t waste it
wordscount.txt
Executor
[How are you doing?,
Hello how are you?,
Hey you are doing great]
[I am happy you are hear today,
Today’s whether is amazing,
I don’t know what are you doing?]
[I don’t see a person like you before,
Greats things happen but it takes time,
Time is precious don’t waste it]
Parent RDD

Textfile API splits the elements of partitions based on the new line character

Example1:Word Count

trans3 = trans1.map(removeunicode_splitwords)

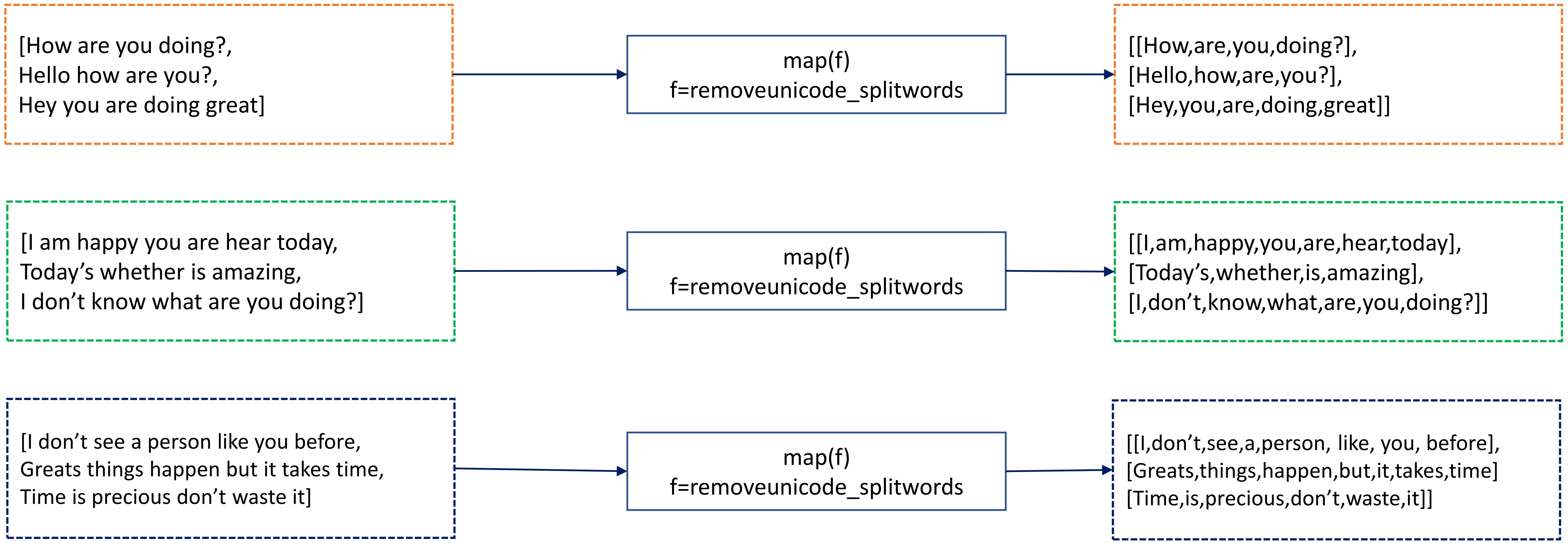
Arguments

1) Python UDF / Lambda Function

Type of RDD:

PVM : pyspark.rdd.PipelinedRDD

JVM : PythonRDD



[How are you doing?,
Hello how are you?,
Hey you are doing great]

map(f)
f=removeunicode_splitwords

[[How,are,you,doing?],
[Hello,how,are,you?],
[Hey,you,are,doing,great]]

[I am happy you are hear today,
Today’s whether is amazing,
I don’t know what are you doing?]

map(f)
f=removeunicode_splitwords

[[I,am,happy,you,are,hear,today],
[Today’s,whether,is,amazing],
[I,don’t,know,what,are,you,doing?]]

[I don’t see a person like you before,
Greats things happen but it takes time,
Time is precious don’t waste it]

map(f)
f=removeunicode_splitwords

[[I,don’t,see,a,person, like, you, before],
[Greats,things,happen,but,it,takes,time]
[Time,is,precious,don’t,waste,it]]

Note: map is a transformation. it’s an one to one mapping means for every element in partitions it will apply function and returns the output for that record.

Note: The UDF of map functions reference should always be define with one parameter

```
def removeunicode_splitwords(x):
    # Removing non-ascii charecters
    ascii_str = str(x).encode("ascii", "ignore").decode()
    return ascii_str.split(" ")
```

[[How,are,you,doing?],
[Hello,how,are,you?],
[Hey,you,are,doing,great]]

map(f)
f=removeunicode_splitwords

[[How,are,you,doing?],
[Hello,how,are,you?],
[Hey,you,are,doing,great]]

```
Output_list = []
Element 1 →
            x = How are you doing?
            ascii_str = How are you doing?
            ascii_str.split(" ") → Output_list .append([“How”, “are”, “you”, ”doing?”])
Element 2 →
           x = Hello how are you?
            ascii_str = Hello how are you?
            ascii_str.split(" ") → Output_list .append([“Hello”, “how”, “are”,“you”, ”doing?”])
Element3 ->
           x = “Hey you are doing great”
            ascii_str = Hey you are doing great
            ascii_str.split(" ") → Output_list .append([“Hey”,”you”,”are”,”doing”,great])
return Output_list
```

Prudhvi Akella@2021

Example1:Word Count

trans4 = trans3.flatMap(lambda x:x)

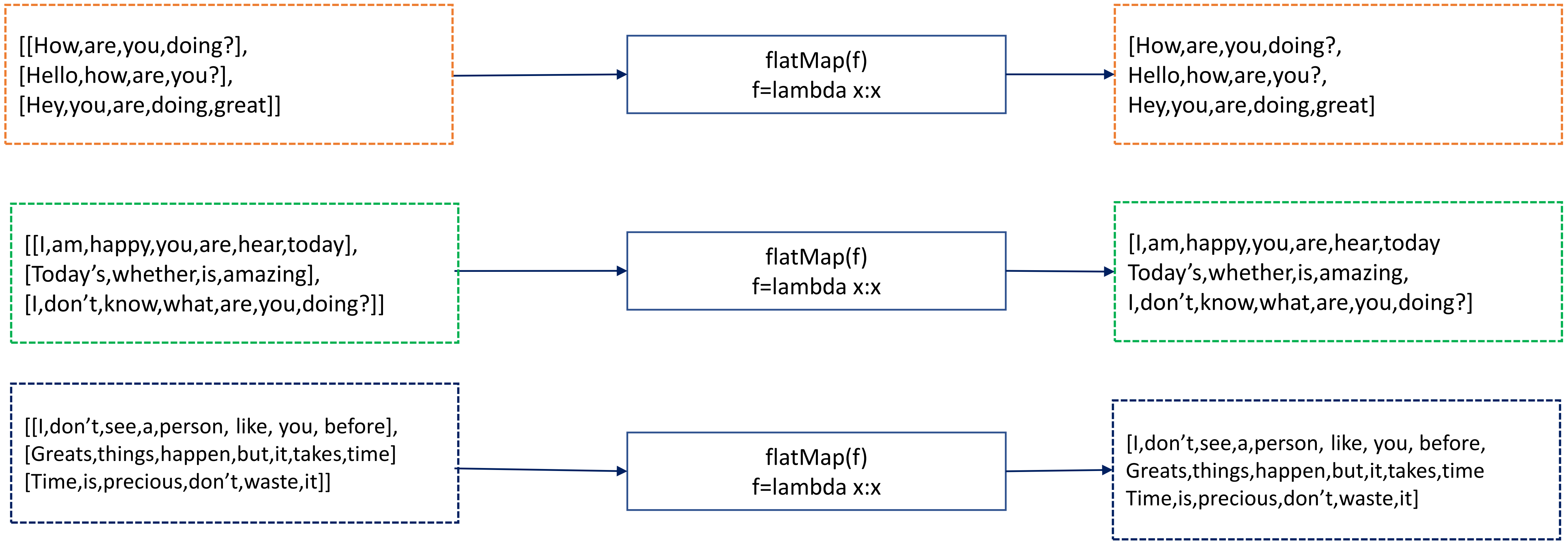
Arguments

1) Python function/ Lambda Function

Type of RDD:

PVM : pyspark.rdd.PipelinedRDD

JVM : PythonRDD



[[How,are,you,doing?],
[Hello,how,are,you?],
[Hey,you,are,doing,great]]

flatMap(f)
f=lambda x:x

[How,are,you,doing?,
Hello,how,are,you?,
Hey,you,are,doing,great]

[[I,am,happy,you,are,hear,today],
[Today’s,whether,is,amazing],
[I,don’t,know,what,are,you,doing?]]

flatMap(f)
f=lambda x:x

[I,am,happy,you,are,hear,today
Today’s,whether,is,amazing,
I,don’t,know,what,are,you,doing?]

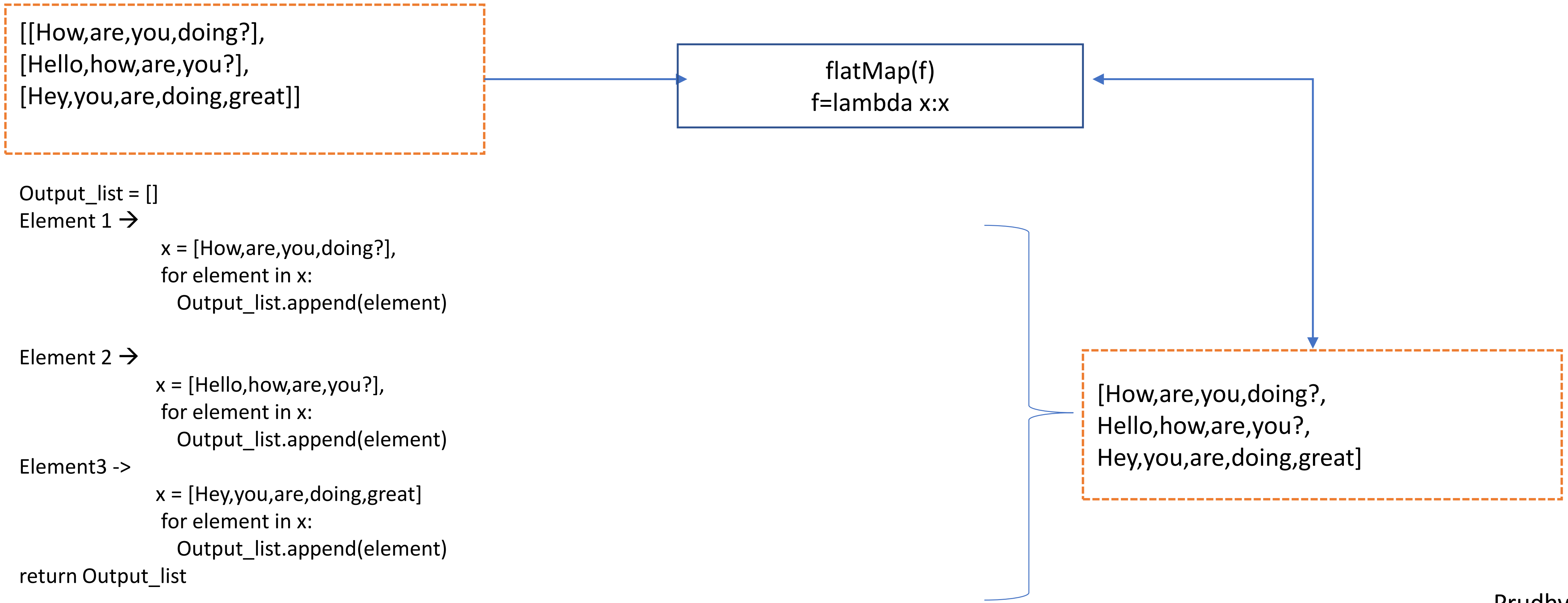
[[I,don’t,see,a,person, like, you, before],
[Greats,things,happen,but,it,takes,time]
[Time,is,precious,don’t,waste,it]]

flatMap(f)
f=lambda x:x

[I,don’t,see,a,person, like, you, before,
Greats,things,happen,but,it,takes,time
Time,is,precious,don’t,waste,it]

Note: map is a transformation. Input to it is nested RDD elements and apply the function and returns the flattened data structure as the output. In our case It flattened the Nested list into single list.

Note: The UDF of flatmap functions reference should always be define with one parameter



[[How,are,you,doing?],
[Hello,how,are,you?],
[Hey,you,are,doing,great]]

flatMap(f)
f=lambda x:x

```
Output_list = []
Element 1 →
                x = [How,are,you,doing?],
                for element in x:
                  Output_list.append(element)

Element 2 →
               x = [Hello,how,are,you?],
                for element in x:
                  Output_list.append(element)
Element3 ->
               x = [Hey,you,are,doing,great]
                for element in x:
                  Output_list.append(element)
return Output_list
```

[How,are,you,doing?,
Hello,how,are,you?,
Hey,you,are,doing,great]

Prudhvi Akella@2021

Example1:Word Count

trans5 = trans4.map(makepairedRDD)

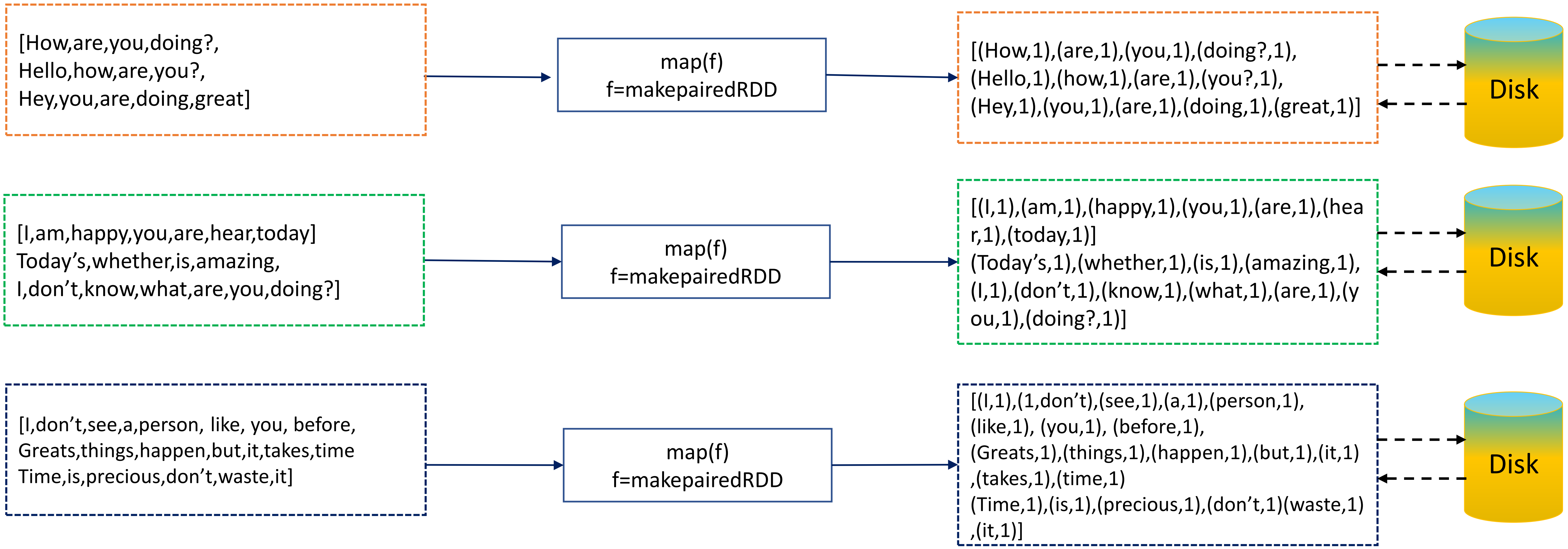
Arguments

1) Python function/ Lambda Function

Type of RDD:

PVM : pyspark.rdd.PipelinedRDD

JVM : PythonRDD



[How,are,you,doing?,
Hello,how,are,you?,
Hey,you,are,doing,great]

[(How,1),(are,1),(you,1),(doing?,1),
(Hello,1),(how,1),(are,1),(you?,1),
(Hey,1),(you,1),(are,1),(doing,1),(great,1)]

[I,am,happy,you,are,hear,today]
Today's,whether,is,amazing,
I,don't,know,what,are,you,doing?]

[(I,1),(am,1),(happy,1),(you,1),(are,1),(hea
r,1),(today,1)]
(Today's,1),(whether,1),(is,1),(amazing,1),
(I,1),(don't,1),(know,1),(what,1),(are,1),(y
ou,1),(doing?,1)]

[I,don't,see,a,person, like, you, before,
Greats,things,happen,but,it,takes,time
Time,is,precious,don't,waste,it]

[(I,1),(1,don't),(see,1),(a,1),(person,1),
(like,1), (you,1), (before,1),
(Greats,1),(things,1),(happen,1),(but,1),(it,1)
,(takes,1),(time,1)
(Time,1),(is,1),(precious,1),(don't,1)(waste,1)
,(it,1)]

Prudhvi Akella@2021

Example1:Word Count

map(makepairedRDD) → makepairedRDD function pairs each word with 1. where the output for each RDD element is going to be Tupple2[(String, Int)]

For Example :
How, are, you, doing → (How,1),(are,1),(you,1),(doing?,1)

If you closely observe the output of each element (How,1) has two things (key, value) where key = “How” value = 1

→ output of map(makepairedRDD) can also be called as pariedRDD.

Why PariedRDD’s are required?

If you want to perform any aggregations based on key then spark provides some key based API’s/ transformations on RDD and they expect PariedRDD as an input. These transformation are also called as wide transformations and these will lead to shuffling phase.

Examples of pariedRDD.
GroupbyKey
ReducebyKey

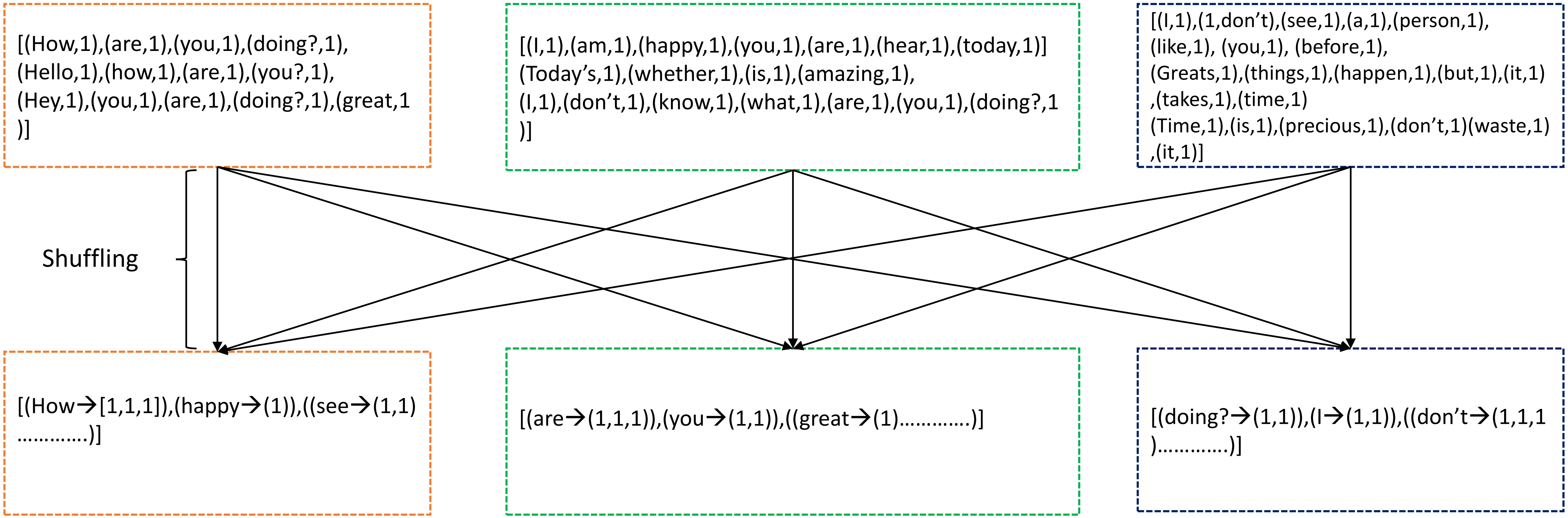
Example1:Word Count

trans6 = trans5.groupByKey()

It ensure that all the keys are grouped with there value in one partition.

Type of RDD:

PVM : pyspark.rdd.PipelinedRDD	**JVM** : PythonRDD



[(How,1),(are,1),(you,1),(doing?,1),(Hello,1),(how,1),(are,1),(you?,1),(Hey,1),(you,1),(are,1),(doing?,1),(great,1)]

[(I,1),(am,1),(happy,1),(you,1),(are,1),(hear,1),(today,1)] (Today's,1),(whether,1),(is,1),(amazing,1),(I,1),(don't,1),(know,1),(what,1),(are,1),(you,1),(doing?,1)]

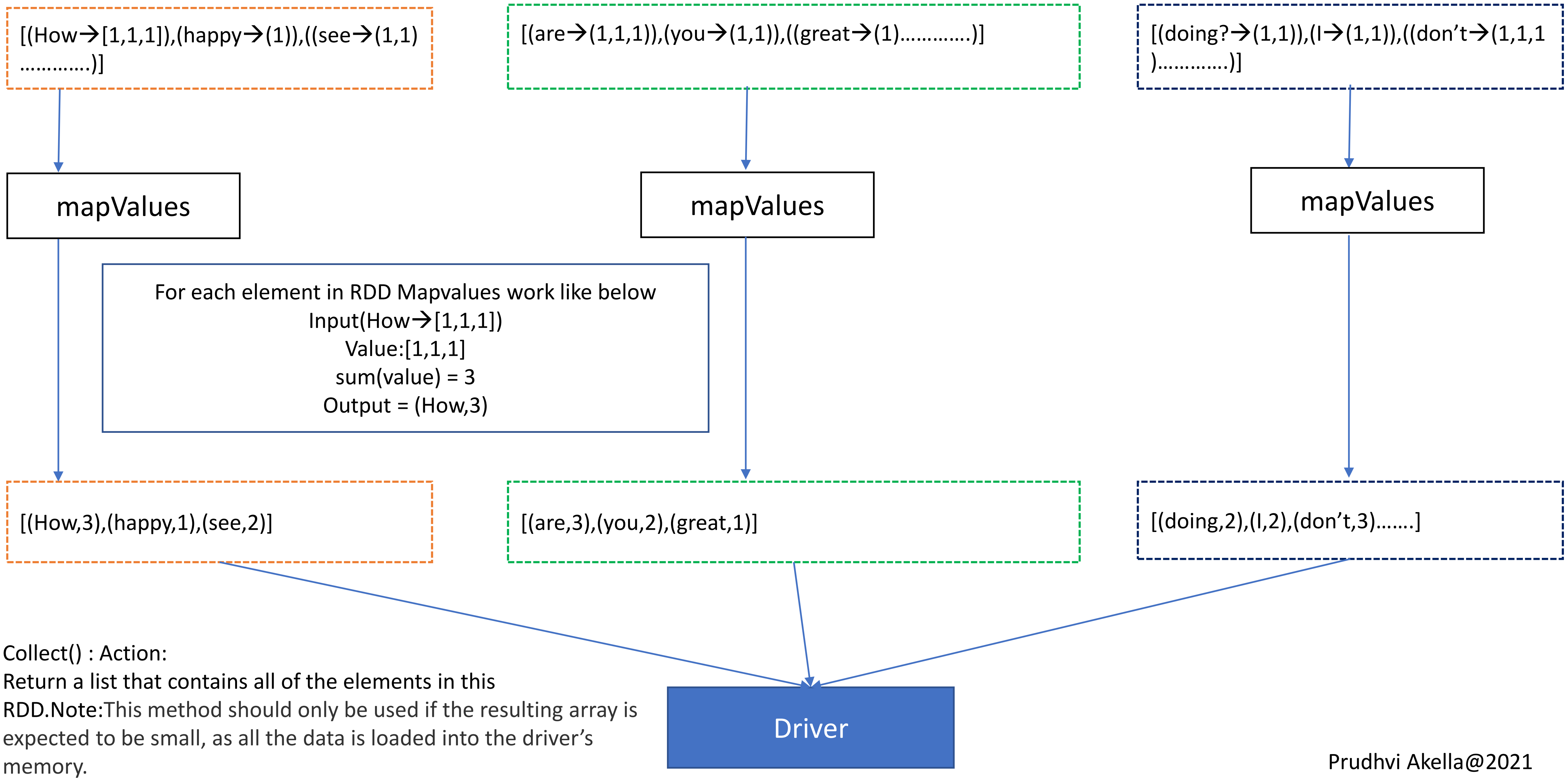
[(I,1),(1,don't),(see,1),(a,1),(person,1),(like,1), (you,1), (before,1),(Greats,1),(things,1),(happen,1),(but,1),(it,1),(takes,1),(time,1) (Time,1),(is,1),(precious,1),(don't,1)(waste,1),(it,1)]

[(How→[1,1,1]),(happy→(1)),((see→(1,1)…………)]

[(are→(1,1,1)),(you→(1,1)),((great→(1)…………)]

[(doing?→(1,1)),(I→(1,1)),((don't→(1,1,1)…………)]

Prudhvi Akella@2021


[(How→[1,1,1]),(happy→(1)),((see→(1,1) …………)]
[(are→(1,1,1)),(you→(1,1)),((great→(1)…………)]
[(doing?→(1,1)),(I→(1,1)),((don't→(1,1,1)…………)]
mapValues
mapValues
mapValues
[(How,3),(happy,1),(see,2)]
[(are,3),(you,2),(great,1)]
[(doing,2),(I,2),(don't,3)…….]
Driver

For each element in RDD Mapvalues work like below
Input(How→[1,1,1])
Value:[1,1,1]
sum(value) = 3
Output = (How,3)

Collect() : Action:
Return a list that contains all of the elements in this RDD.Note:This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

Prudhvi Akella@2021

Hash and Range Partitioner, Custom Partitioner in Spark.

Join or Any sort of key based transformation

Hash Partitioner

    hash(key) % number of partition => to which partition this key should shuffle to.

Example1:Word Count

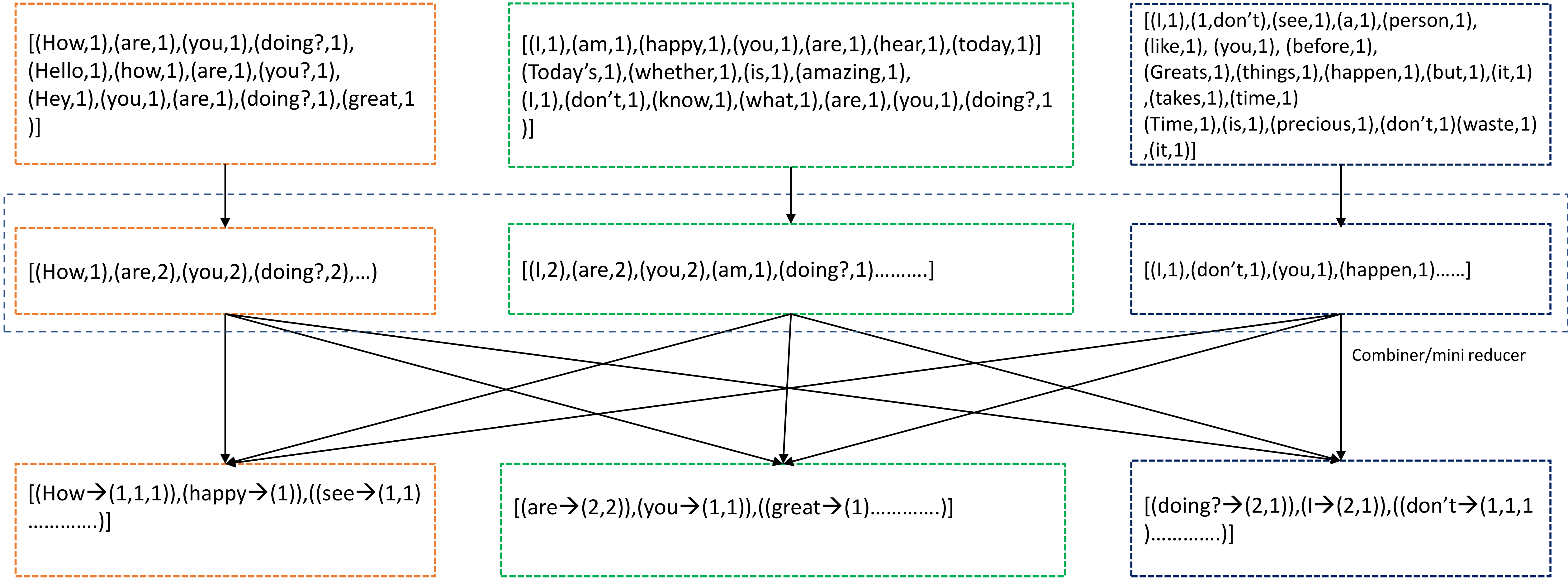
trans7 = trans5.reduceByKey(lambda x,y:x+y)

Arguments :

Python function which takes two inputs or lambda with two inputs

Type of RDD:

PVM : pyspark.rdd.PipelinedRDD	**JVM** : PythonRDD



[(How,1),(are,1),(you,1),(doing?,1),(Hello,1),(how,1),(are,1),(you?,1),(Hey,1),(you,1),(are,1),(doing?,1),(great,1)]

[(I,1),(am,1),(happy,1),(you,1),(are,1),(hear,1),(today,1)] (Today's,1),(whether,1),(is,1),(amazing,1),(I,1),(don't,1),(know,1),(what,1),(are,1),(you,1),(doing?,1)]

[(I,1),(1,don't),(see,1),(a,1),(person,1),(like,1), (you,1), (before,1),(Greats,1),(things,1),(happen,1),(but,1),(it,1),(takes,1),(time,1)(Time,1),(is,1),(precious,1),(don't,1)(waste,1),(it,1)]

[(How,1),(are,2),(you,2),(doing?,2),…)

[(I,2),(are,2),(you,2),(am,1),(doing?,1)……….]

[(I,1),(don't,1),(you,1),(happen,1)……]

[(How→(1,1,1)),(happy→(1)),((see→(1,1)………….)]

[(are→(2,2)),(you→(1,1)),((great→(1)………….)]

[(doing?→(2,1)),(I→(2,1)),((don't→(1,1,1)………….)]

Example1:Word Count

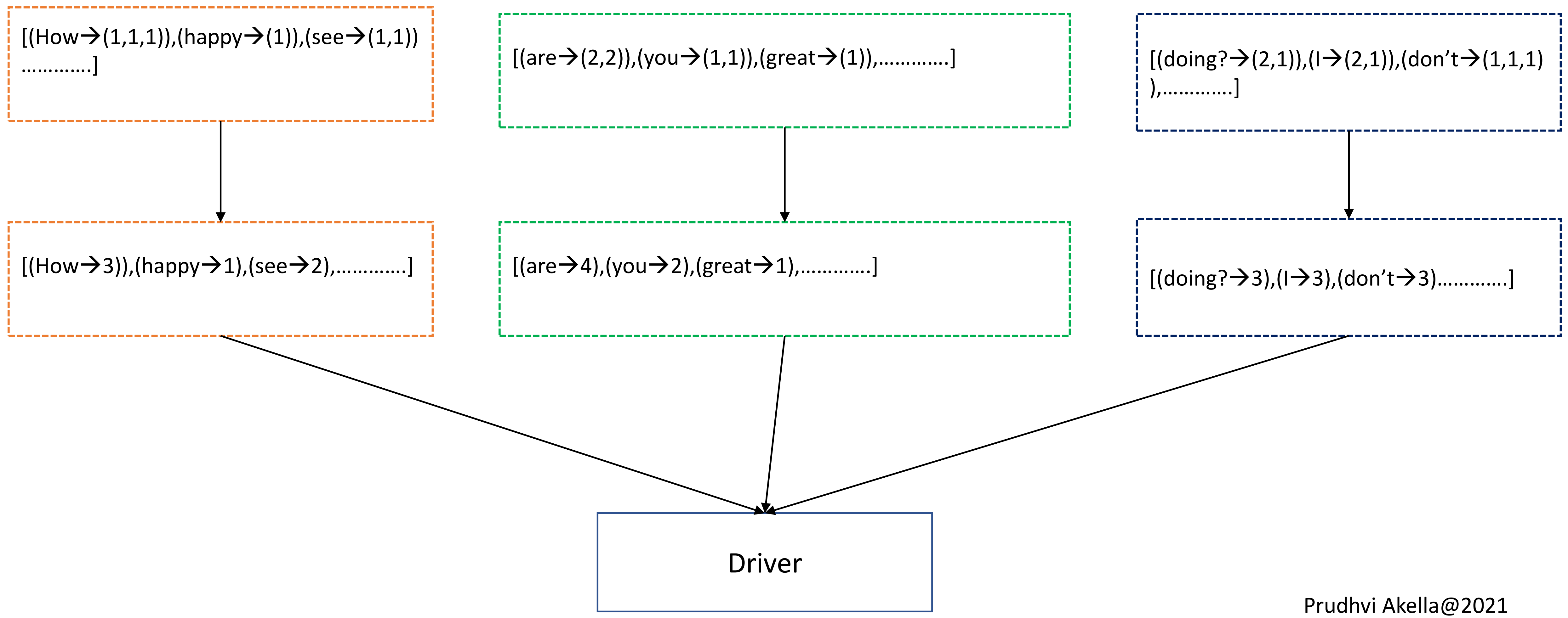
trans7.take(100)

Take is an action which will collect first 100 records from the parititons and send back to driver

<u>Type of RDD</u>:

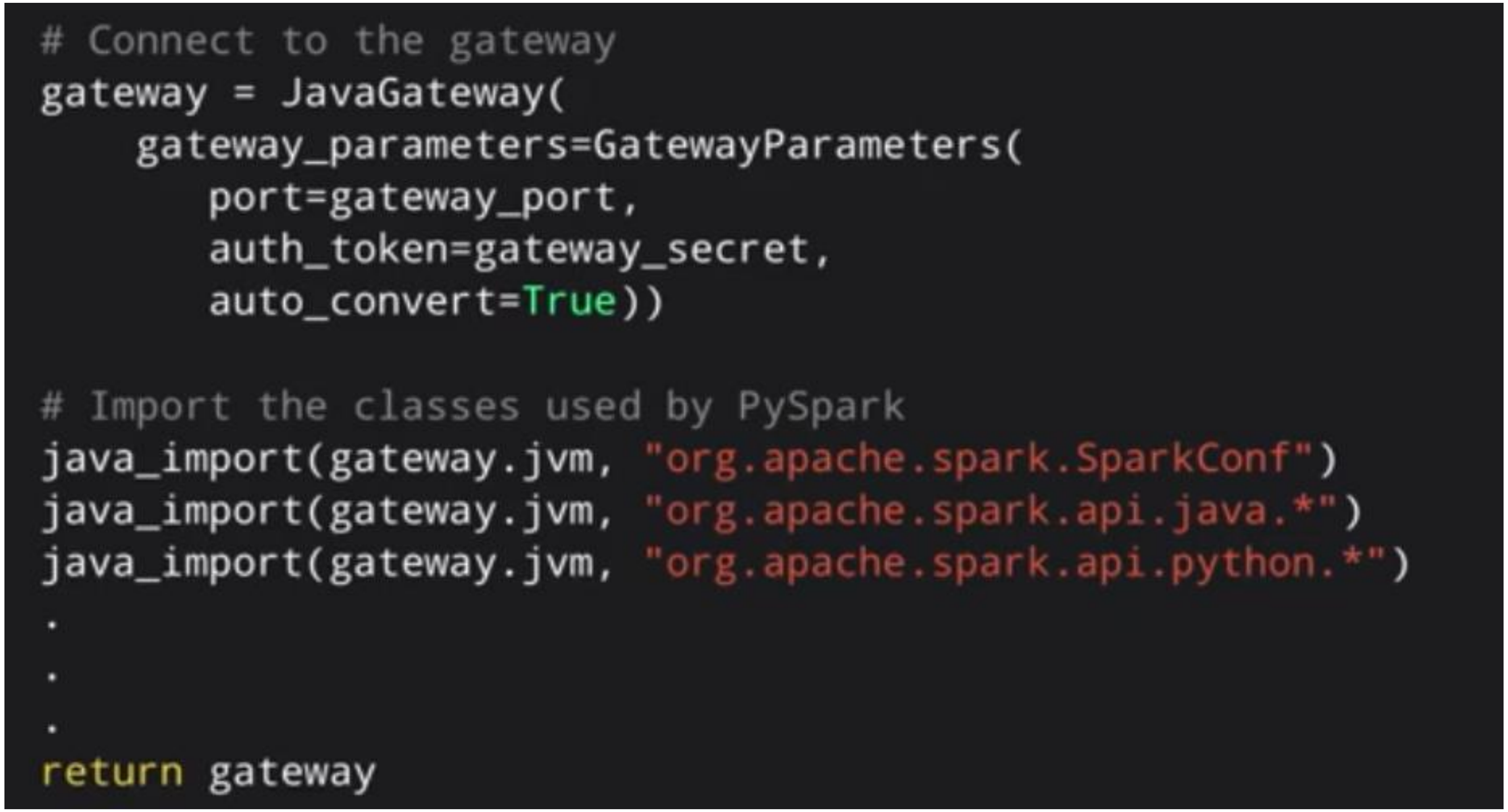
PVM : Respective Values type

JVM : : Respective Values type


[(How→(1,1,1)),(happy→(1)),(see→(1,1)) ………….]
[(are→(2,2)),(you→(1,1)),(great→(1)),………….]
[(doing?→(2,1)),(I→(2,1)),(don't→(1,1,1)),………….]
[(How→3)),(happy→1),(see→2),………….]
[(are→4),(you→2),(great→1),………….]
[(doing?→3),(I→3),(don't→3)………….]
Driver

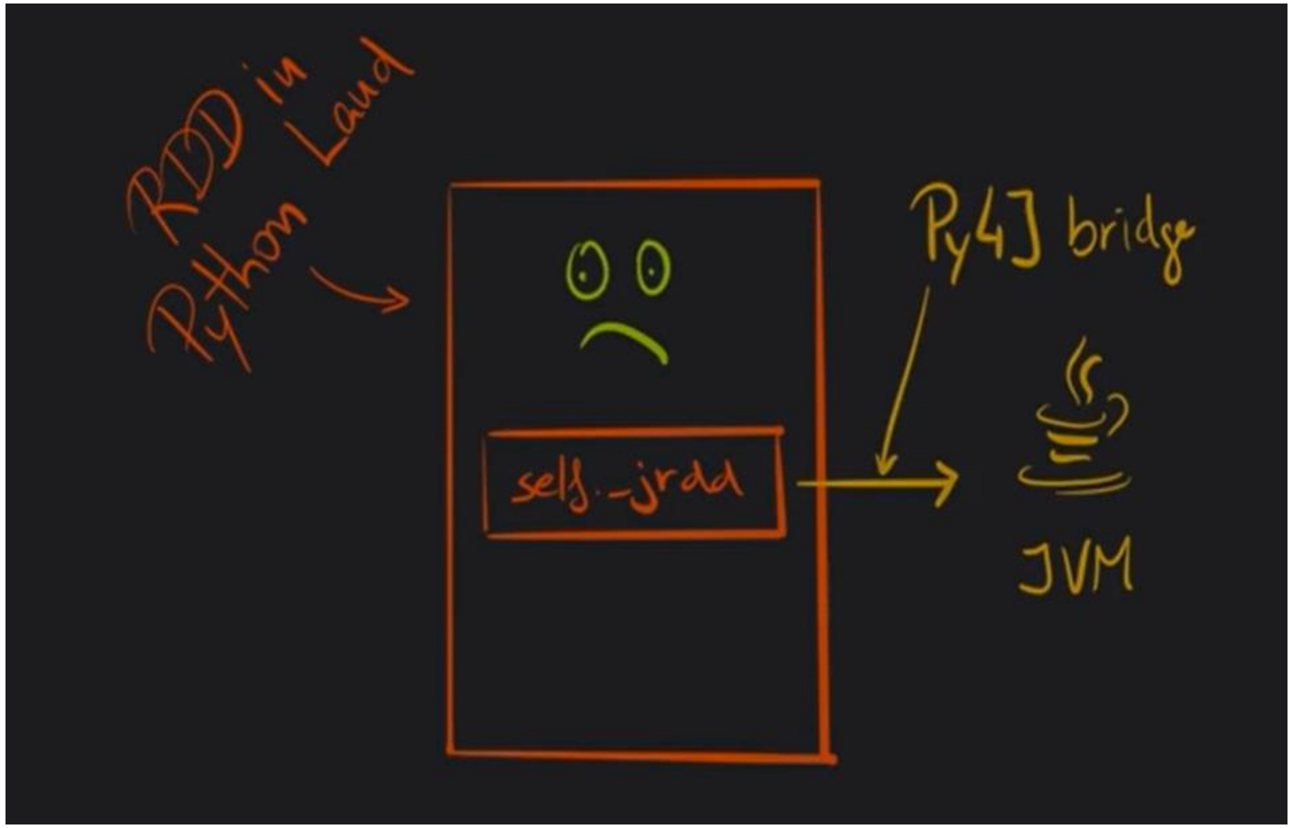
Prudhvi Akella@2021

Internals of RDD in PySpark



```
# Connect to the gateway
gateway = JavaGateway(
    gateway_parameters=GatewayParameters(
        port=gateway_port,
        auth_token=gateway_secret,
        auto_convert=True))

# Import the classes used by PySpark
java_import(gateway.jvm, "org.apache.spark.SparkConf")
java_import(gateway.jvm, "org.apache.spark.api.java.*")
java_import(gateway.jvm, "org.apache.spark.api.python.*")
.
.
.
return gateway
```

Internals of RDD in Pyspark



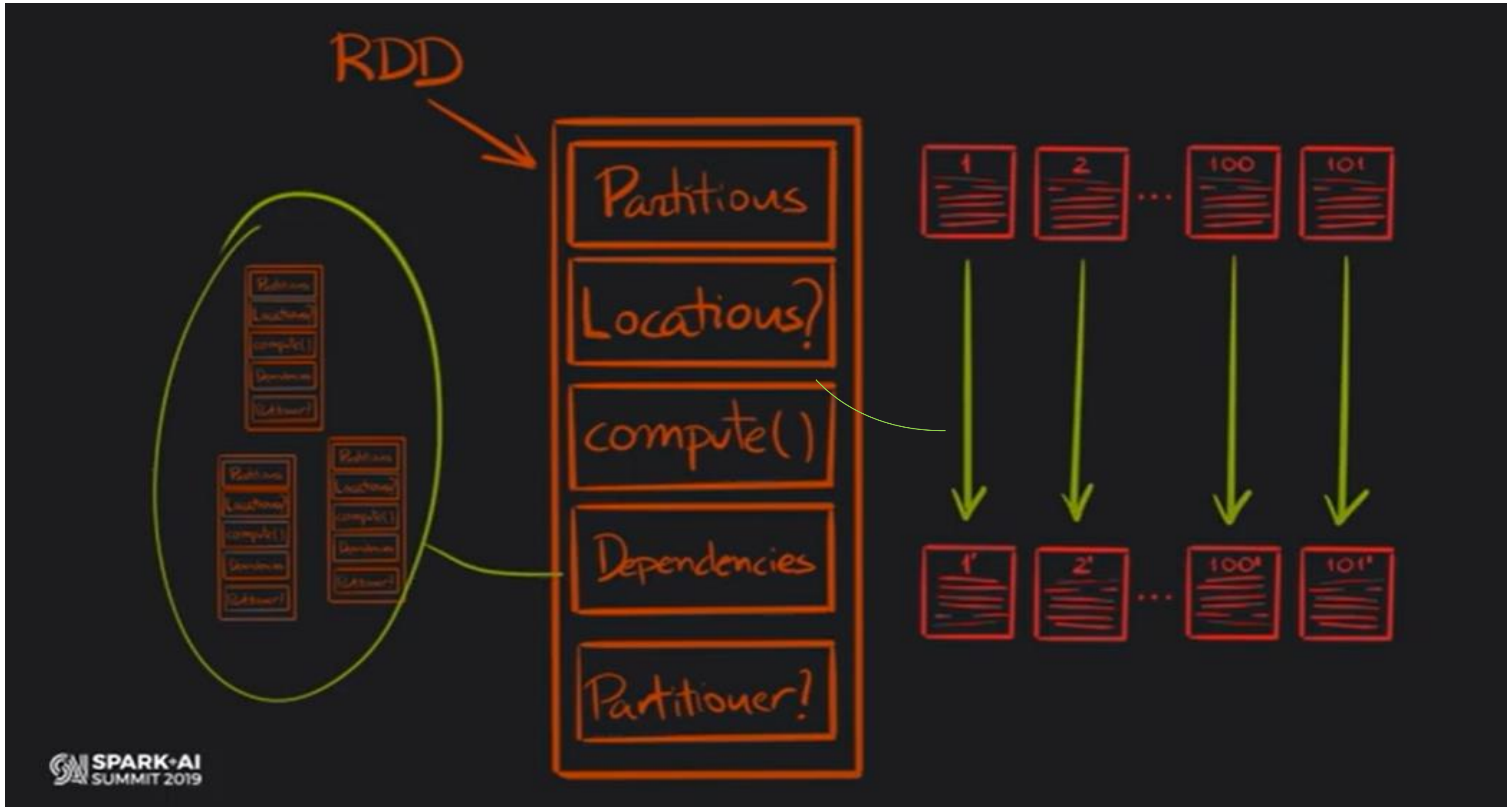
If you clearly observe RDD/Pipelined RDD class in python has a private member called _jrdd which holds the reference of the RDD object that is created in JVM.

Note:

This means every time you create an RDD in Python using Pyspark API's it will create an respective RDD objects in JVM and the reference of that JVM's object will be stored into _jrdd of PVM's RDD object.

Internals of RDD in PySpark

Important Properties of RDD(RDD is a class)

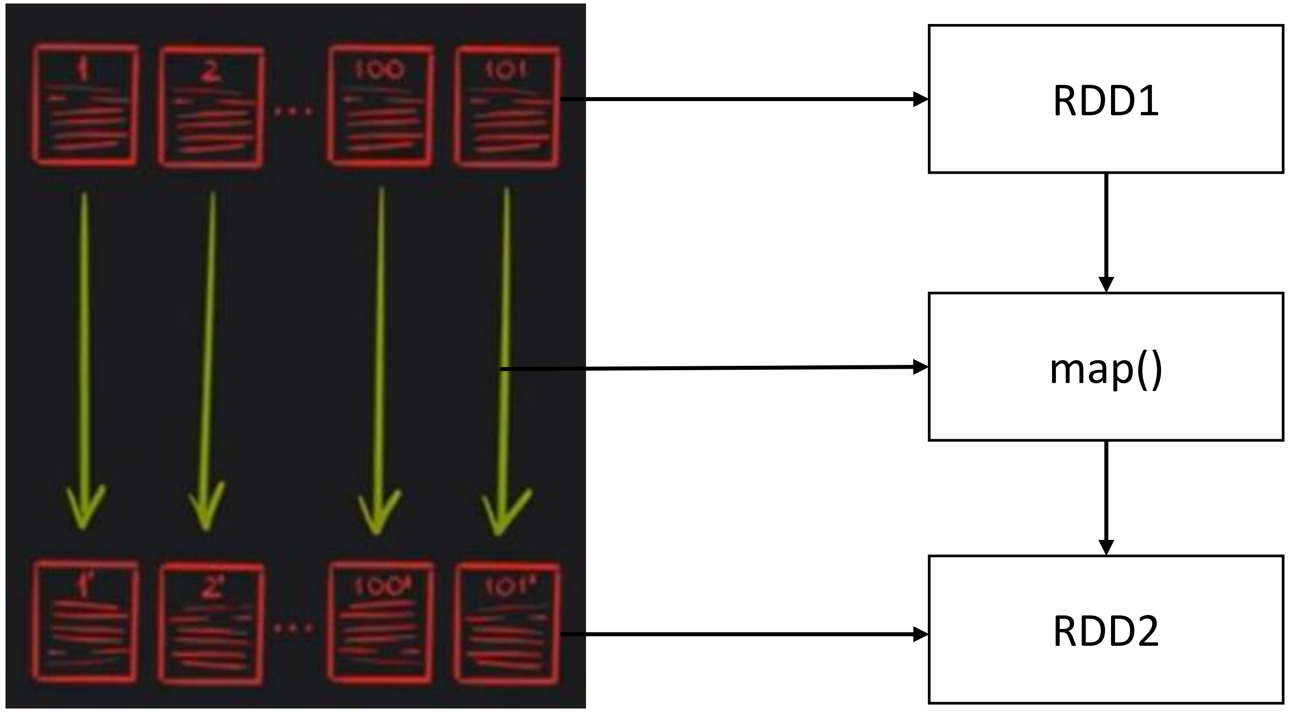


Internals of RDD in PySpark

Partitions: We already discussed about this

Locations: Spark decides the location of the executor based on the data locality. So that its easy to load the data into executor.

compute() : Every RDD has a compute method. Once this method is called computations(transformation/Actions)(map) will applied to the all the partitions of that RDD (RDD1). in executors and creates the new partitions as output of a new RDD(RDD2).



Prudhvi Akella@2021

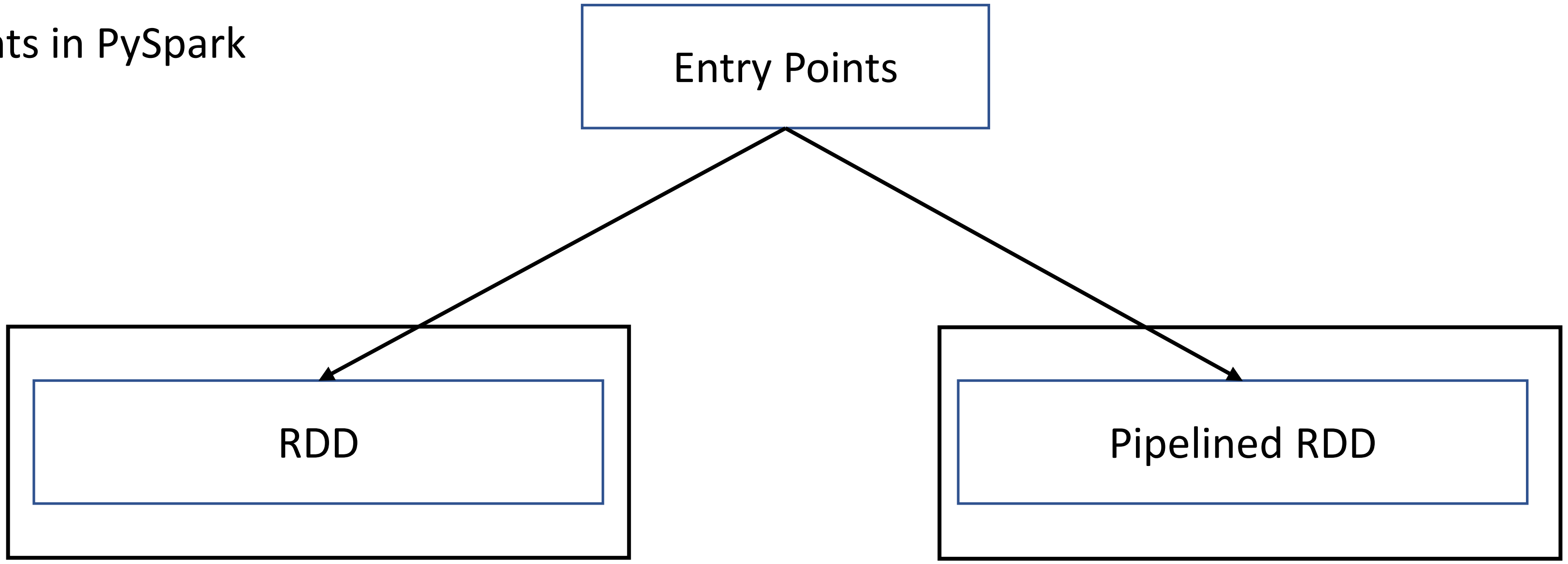
Important Properties of RDD

<u>Dependencies</u>: This is the key point of RDD when one of the partition disappears because the machine that was holding it disappears because of the internet being last. you can recompute only the partitions because you can trace back through compute and the dependence's where this coming from so eventually you can go back to the first original data source and compute just only that. Its more over less talking about the RDD lineage.

<u>Partitioner</u> : Its used to distribute the partitions across the different machine. This plays a very important role with wide transformations.

Internals of RDD in PySpark

RDD Entry Points in PySpark



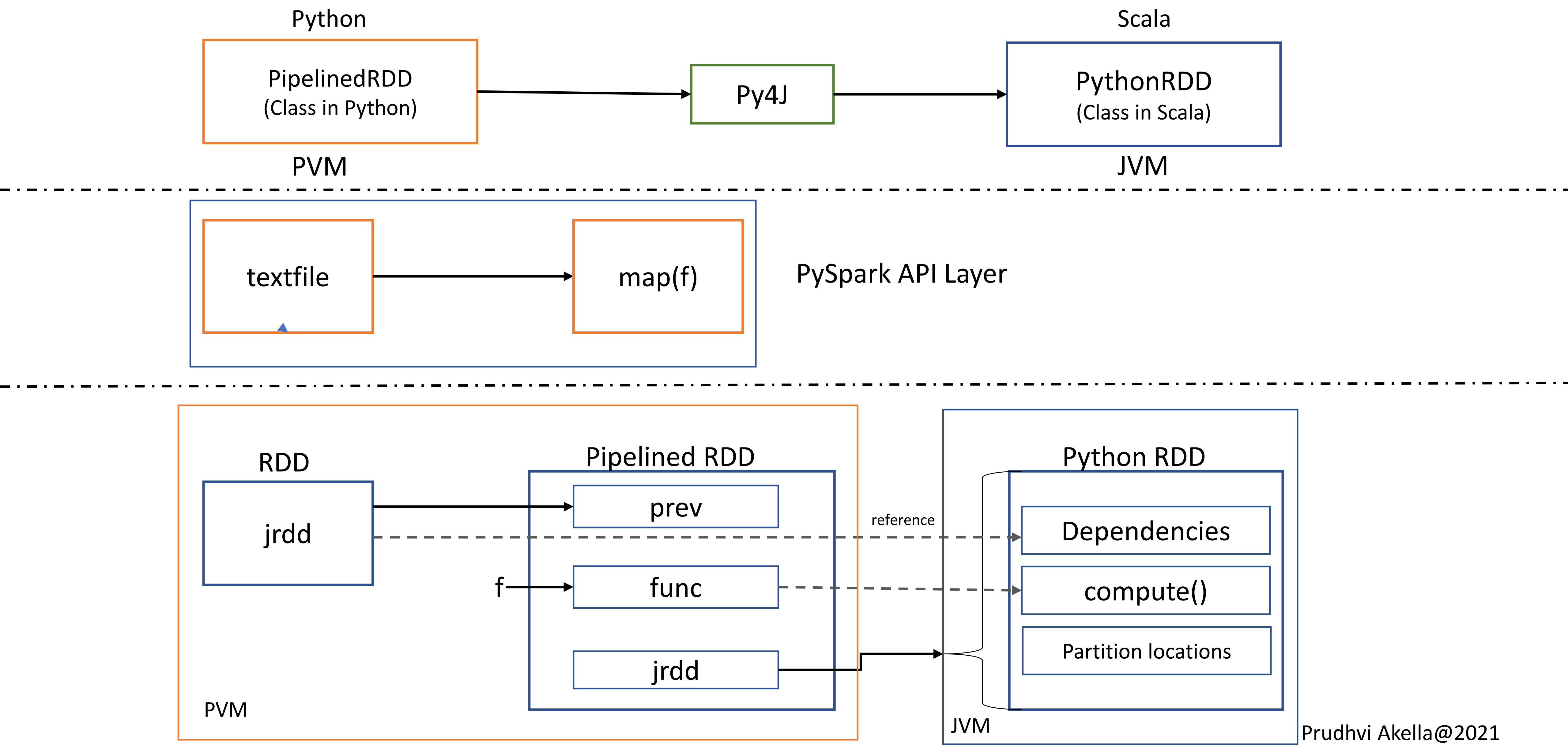
RDD object will be created in PVM when you read the data into Spark using PySpark Read API(textfile, wholeTextfile).

This the main RDD which does all the Work. When ever you try to map the partitions this RDD will be created.

Which create PythonRDD object in JVM(Scala Side) and the reference of it is stored into _jrdd of Pipelined RDD in PVM

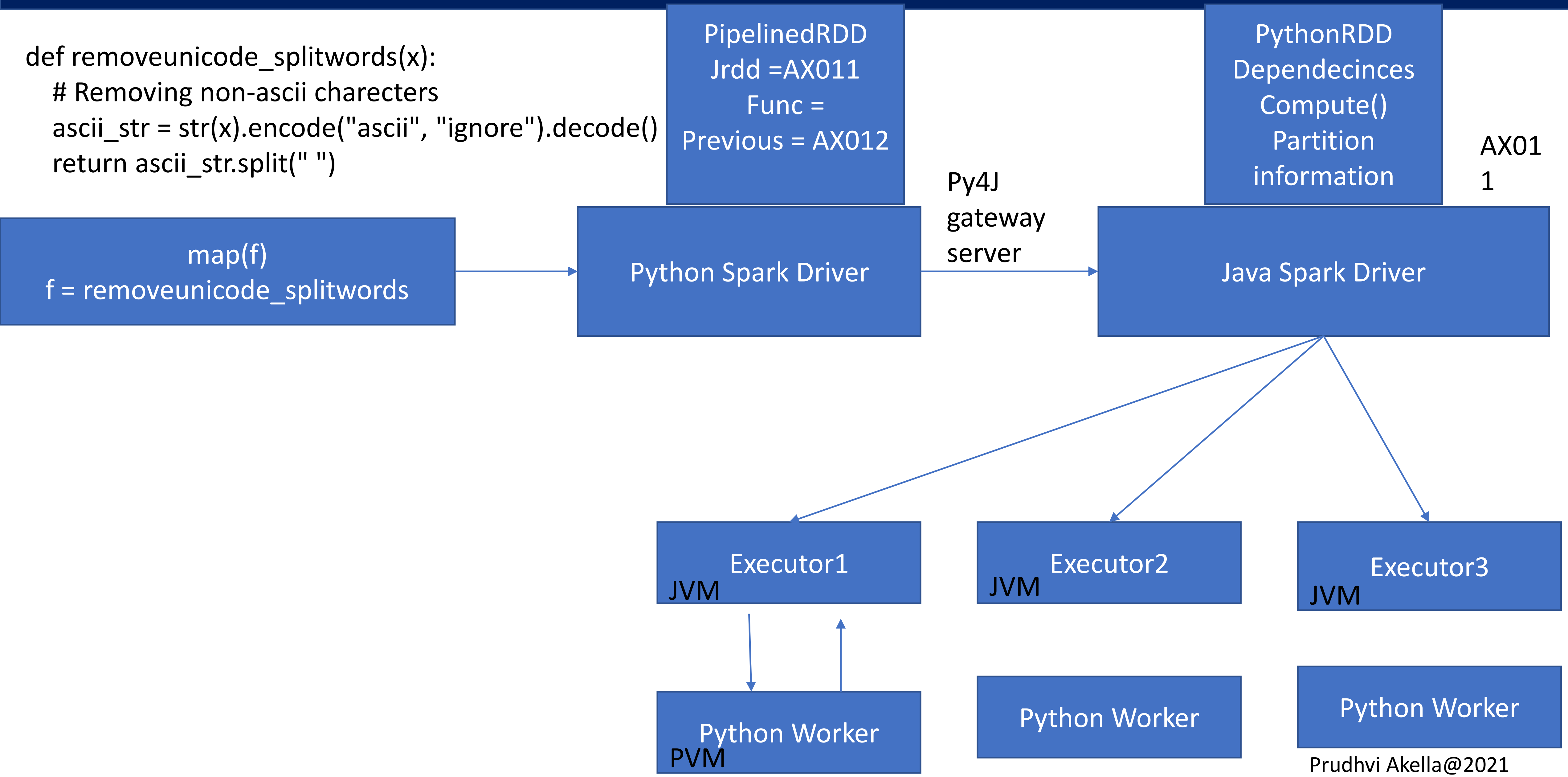
Prudhvi Akella@2021

Internals of RDD in PySpark



Prudhvi Akella@2021

```
def removeunicode_splitwords(x):
    # Removing non-ascii charecters
    ascii_str = str(x).encode("ascii", "ignore").decode()
    return ascii_str.split(" ")
```



Internals of RDD in PySpark

Lets understand by an example

```
trans1 = sc.textFile("C:/MyStuff/Systemdesign_Expert/Datasets/covid/wordscount.txt")
type(trans1)
```

```
pyspark.rdd.RDD
```

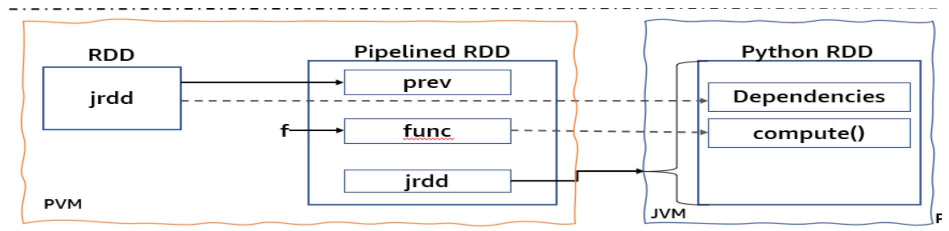
textFile API returns RDD Object which is created in PVM and its equivalent HadoopRDD is created in JVM. The reference of HadoopRDD object is stored in _jrdd of RDD

Lets assume jrdd = AX0101

```
def removeunicode_splitwords(x):
    ascii_str = str(x).encode("ascii", "ignore").decode()
    return ascii_str.split(" ")
```

```
trans3 = trans1.map(removeunicode_splitwords)
type(trans3)
```

```
pyspark.rdd.PipelinedRDD
```



Map transformation will create **PipelinedRDD** in **PVM** and respective **PythonRDD** object **in** JVM and reference of it is stored in **_jrdd** of PipelinedRDD that is created in PVM.

Now lets Assume that _jrdd of **PipelinedRDD = AX0102**

Pipelined RDD :

1) **prev** : previous RDD jrdd address in this case **AX0101**
2) **Func** : **map(f). Where f = removeunicode_splitword**
3) **jrdd**: PythonRDD objects reference that is created in JVM say in our case **AX0102**

1) **PythonRDD**: It has two Important things:
2) **Dependencies**: Previous RDD's Address in this case RDD._jrdd address

3) **compute**() : is magic method it executes map function(in this case **removeunicode_splitwords** function) by creating executors

Prudhvi Akella@2021

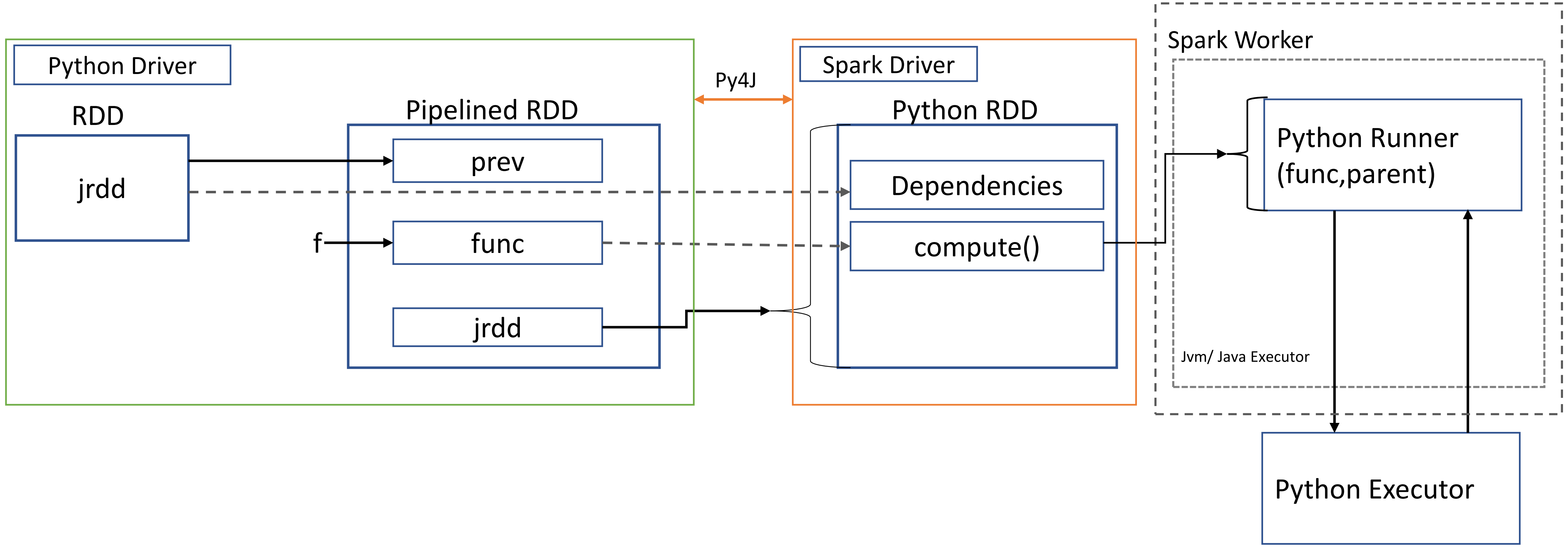
removeunicode_splitword is a Python can it be executed in JVM?

compute() is the Magic

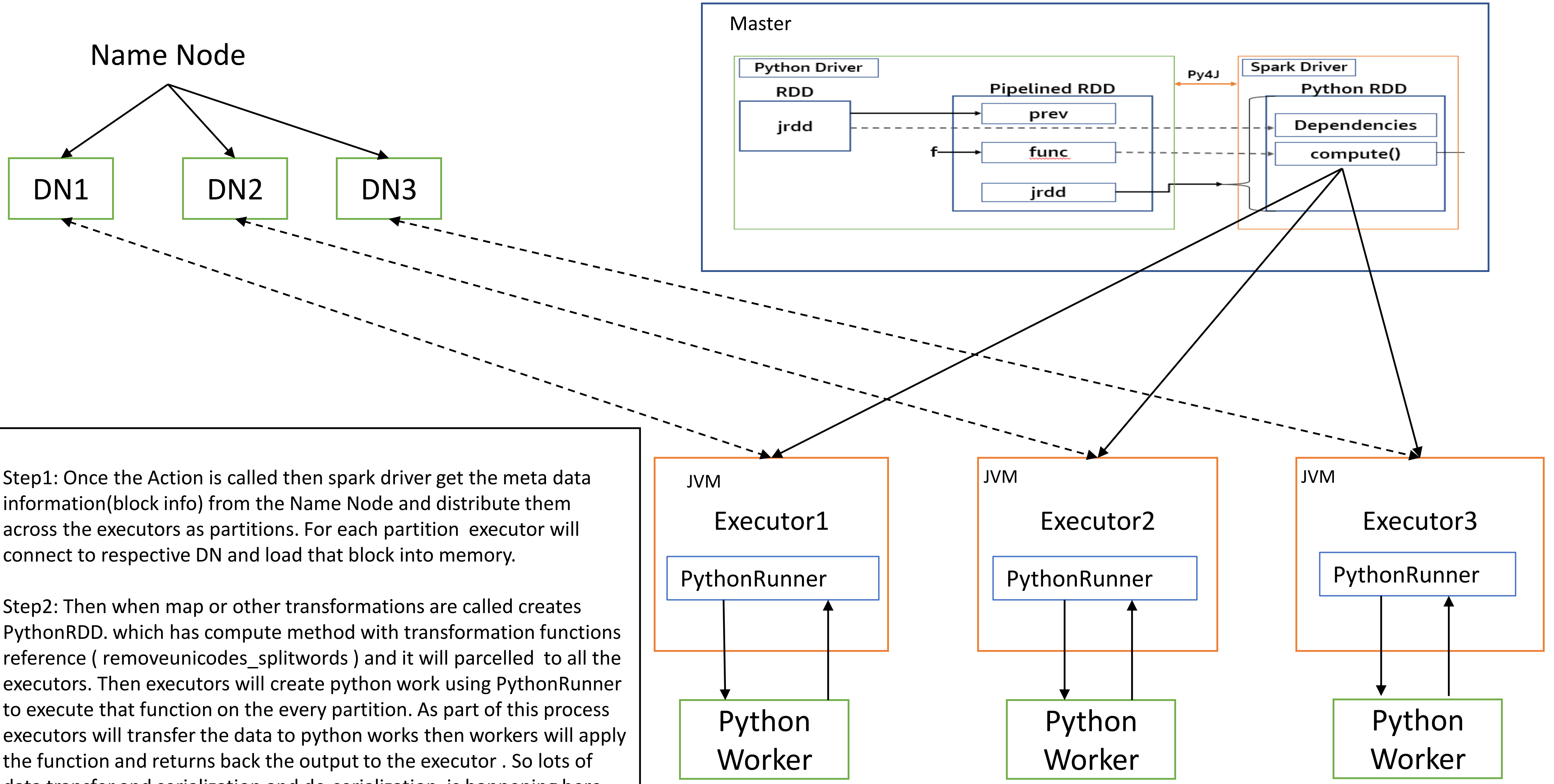
compute() runs on each EXECUTOR and start PYTHON WORKER VIA PythonRunner.

Why python worker is required?

As f in map(f)[map(removeunicodes_splitwords)] is an python function/lambda which cannot be executed in executor which is a JVM process so it launches python worker which is an python processes



Prudhvi Akella@2021



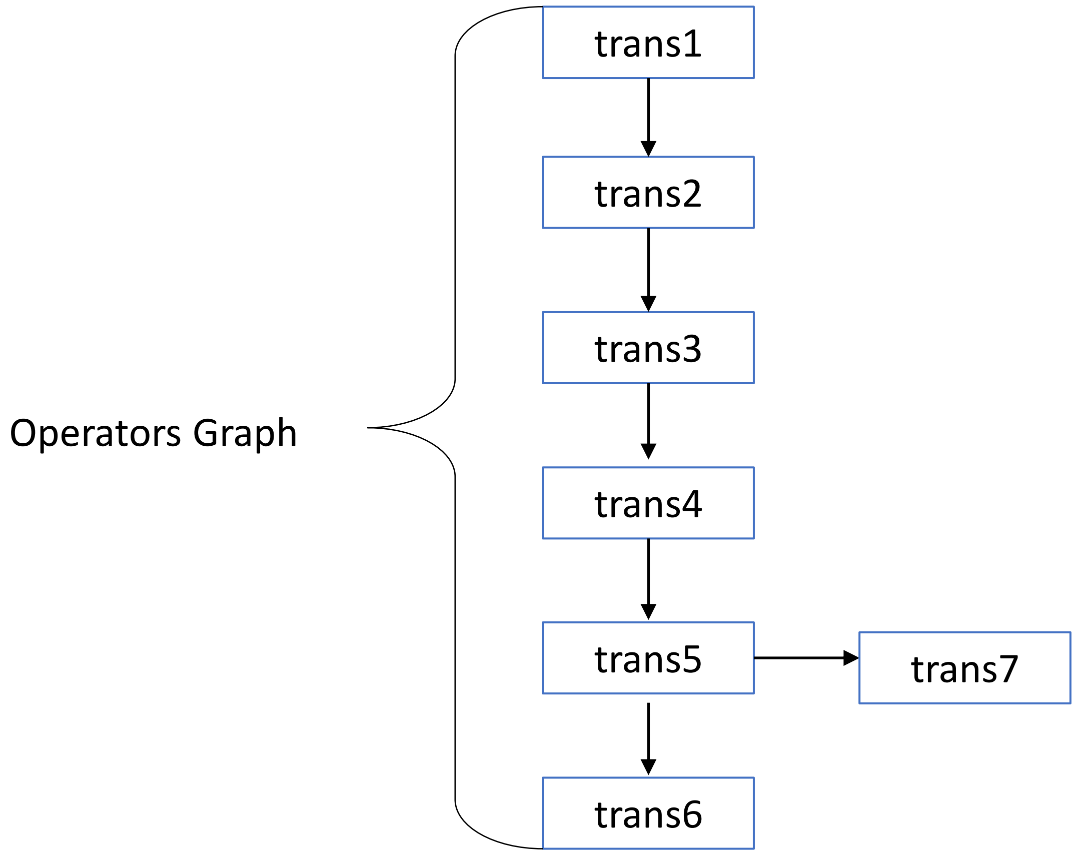
Step1: Once the Action is called then spark driver get the meta data information(block info) from the Name Node and distribute them across the executors as partitions. For each partition executor will connect to respective DN and load that block into memory.

Step2: Then when map or other transformations are called creates PythonRDD. which has compute method with transformation functions reference (removeunicodes_splitwords) and it will parcelled to all the executors. Then executors will create python work using PythonRunner to execute that function on the every partition. As part of this process executors will transfer the data to python works then workers will apply the function and returns back the output to the executor . So lots of data transfer and serialization and de-serialization is happening here.

Prudhvi Akella@2021

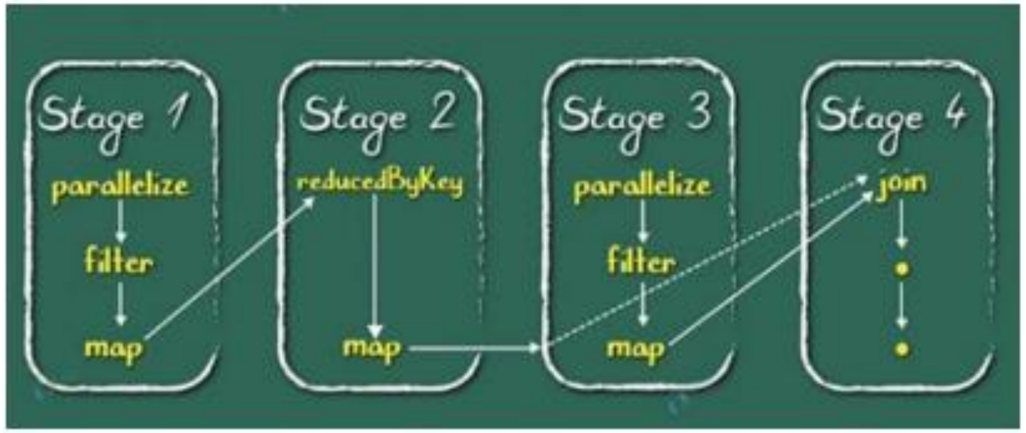
Operators Graph/ RDD Lineage

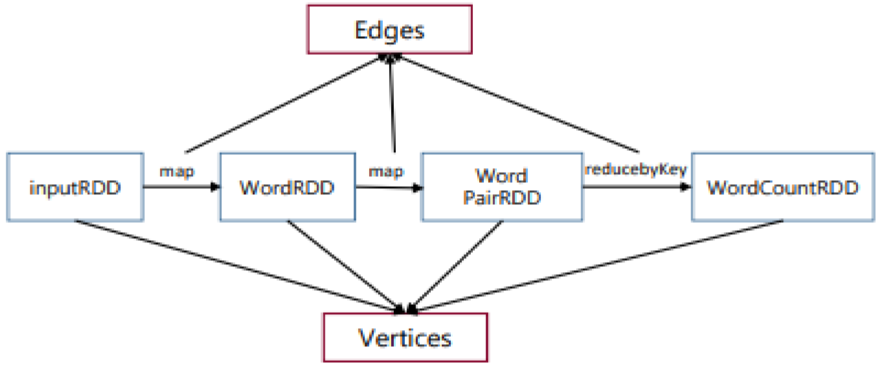
- → As we discussed Spark transformations are lazy (means spark will not execute transformation until a action is performed) .
- → Instead each RDD maintains a pointer to one or more parent RDD's along with metadata about what type of relationship it has with the Parent RDD. Its called a Lineage.
- → Lineage is created for each transformation.
- → A Lineage will keep track of what all transformations has to be applied on the RDD, including the location from where it has to read the data.
- → RDD lineage is used to re compute the data if there any faults as it contains the pattern of the computation.



DAG(Direct Acyclic Graph)

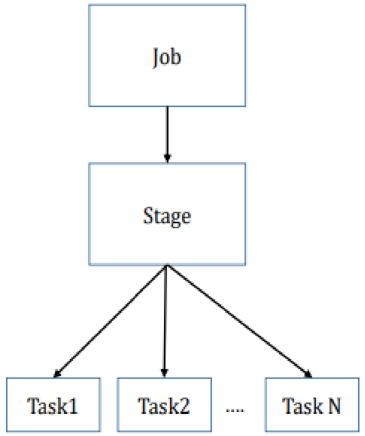
→ (Directed Acyclic Graph) DAG in Apache Spark is a set of Vertices and Edges
→ Vertices represent the RDDs and the edges represent the Operation to be applied on RDD.
→ In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task.



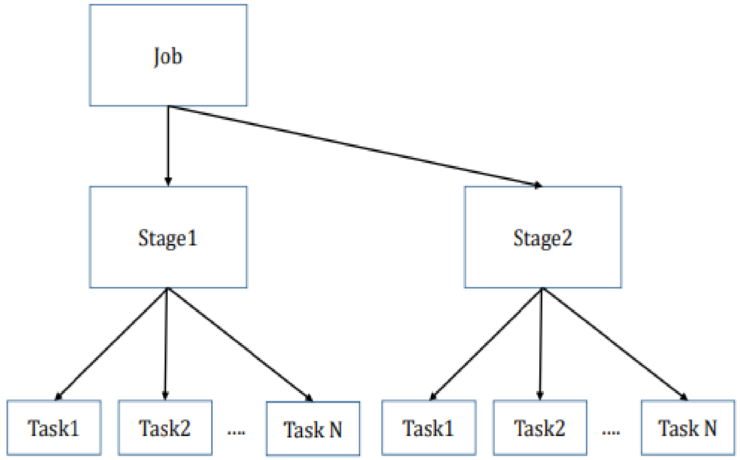


DAG(Direct Acyclic Graph)

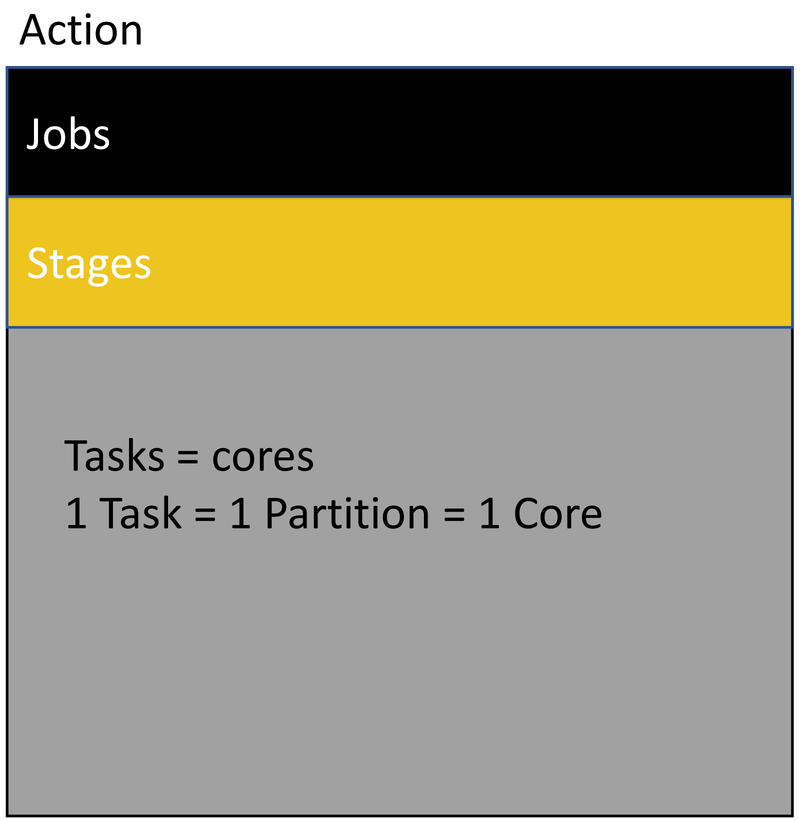
→ For every action one new job will be created by spark application.
→Job is further split into stages based on the transformations(For every wide transformation one new stage will be added to DAG)
→ Each stage is further split into tasks



Narrow Transformations



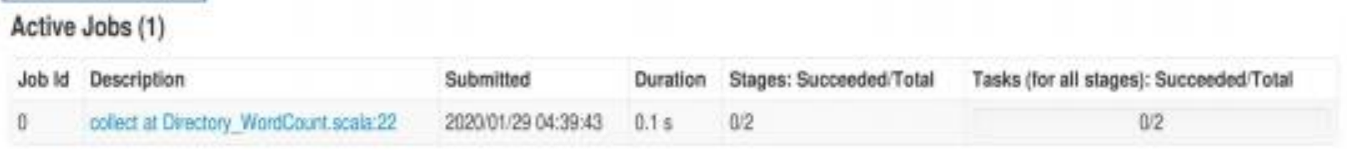
Wide Transformations



Tasks = cores
1 Task = 1 Partition = 1 Core

DAG(Direct Acyclic Graph)

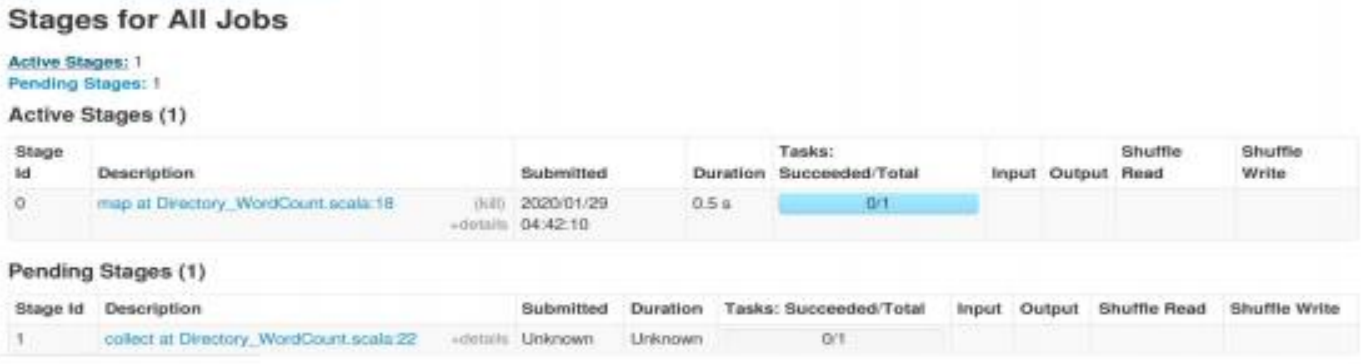
Job View



Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at Directory_WordCount.scala:22	2020/01/29 04:39:43	0.1 s	0/2	0/2

Detail Stage View



Stages for All Jobs

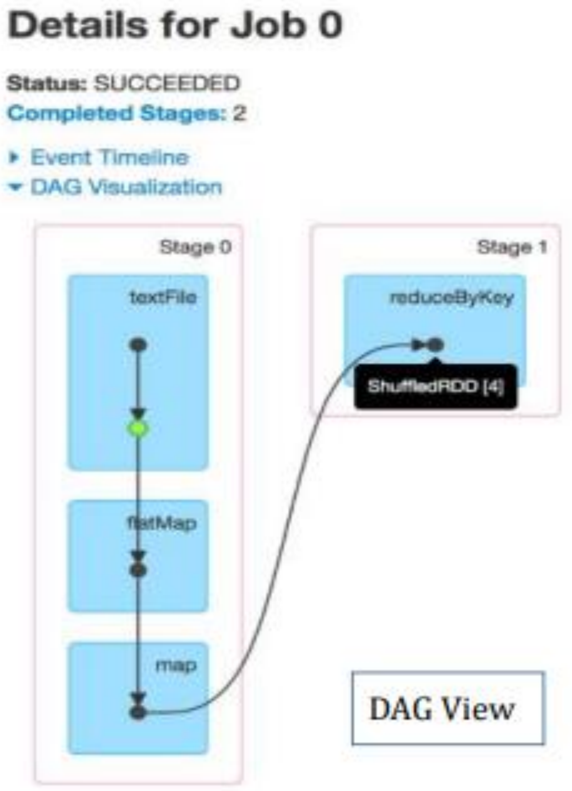
Active Stages: 1
Pending Stages: 1

Active Stages (1)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	map at Directory_WordCount.scala:18	(kill) +details	2020/01/29 04:42:10	0.5 s	0/1				

Pending Stages (1)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at Directory_WordCount.scala:22	+details	Unknown	Unknown	0/1				



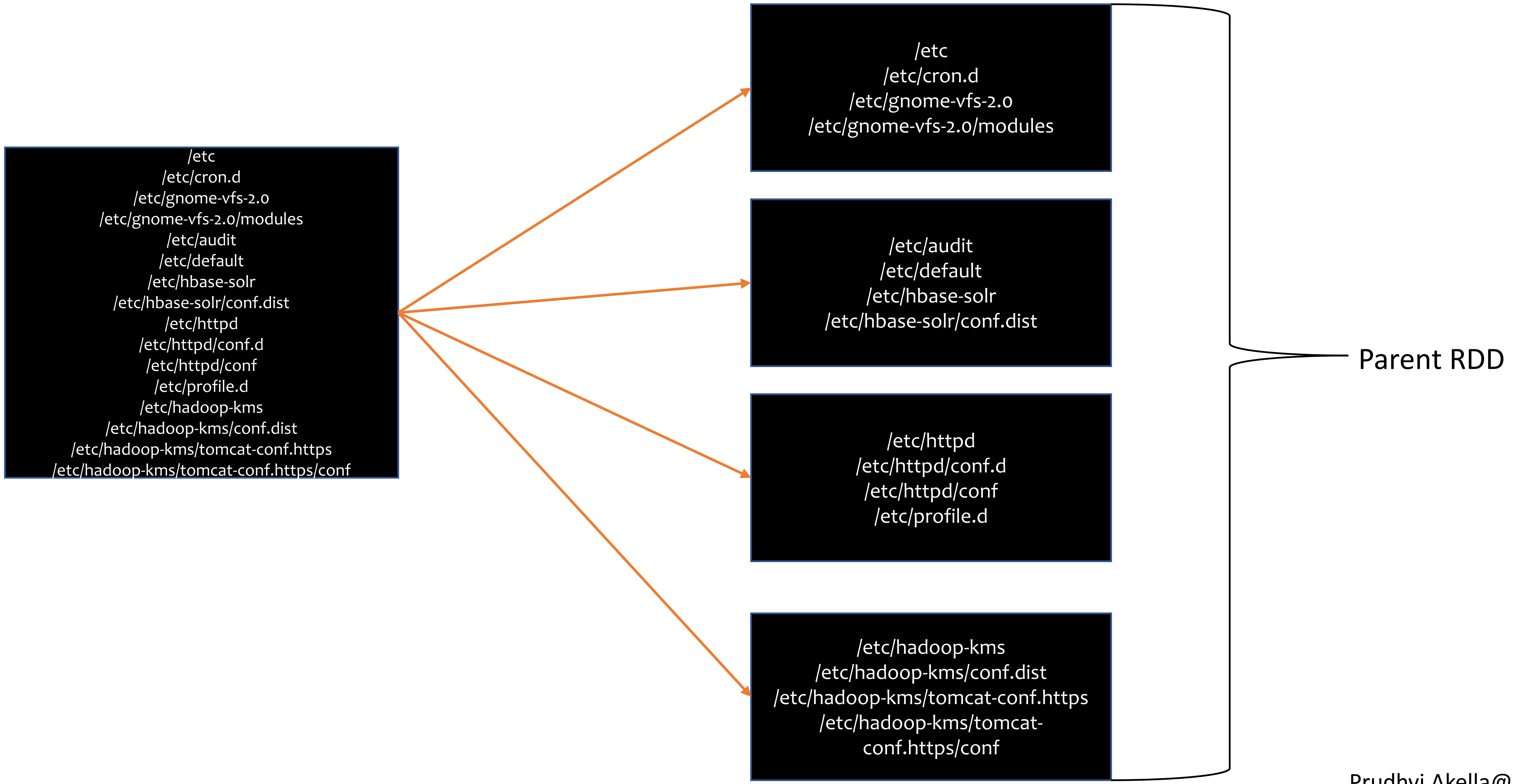
Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

DAG View

Example2:


/etc
/etc/cron.d
/etc/gnome-vfs-2.0
/etc/gnome-vfs-2.0/modules
/etc/audit
/etc/default
/etc/hbase-solr
/etc/hbase-solr/conf.dist
/etc/httpd
/etc/httpd/conf.d
/etc/httpd/conf
/etc/profile.d
/etc/hadoop-kms
/etc/hadoop-kms/conf.dist
/etc/hadoop-kms/tomcat-conf.https
/etc/hadoop-kms/tomcat-conf.https/conf
/etc
/etc/cron.d
/etc/gnome-vfs-2.0
/etc/gnome-vfs-2.0/modules
/etc/audit
/etc/default
/etc/hbase-solr
/etc/hbase-solr/conf.dist
/etc/httpd
/etc/httpd/conf.d
/etc/httpd/conf
/etc/profile.d
/etc/hadoop-kms
/etc/hadoop-kms/conf.dist
/etc/hadoop-kms/tomcat-conf.https
/etc/hadoop-kms/tomcat-conf.https/conf
Parent RDD

Example2:

/etc
/etc/cron.d
/etc/gnome-vfs-2.0
/etc/gnome-vfs-2.0/modules

/etc/audit
/etc/default
/etc/hbase-solr
/etc/hbase-solr/conf.dist

/etc/httpd
/etc/httpd/conf.d
/etc/httpd/conf
/etc/profile.d

/etc/hadoop-kms
/etc/hadoop-kms/conf.dist
/etc/hadoop-kms/tomcat-conf.https
/etc/hadoop-kms/tomcat-conf.https/conf

```
sc.textFile("C:/MyStuff/System design_Expert/Spark/directories1.txt",4)
```

[etc] [other words]
[etc] [other words]
[etc] [other words]
[etc] [other words]

[etc] [other words]
[etc] [other words]
[etc] [other words]
[etc] [other words]

[etc] [other words]
[etc] [other words]
[etc] [other words]
[etc] [other words]

[etc] [other words]
[etc] [other words]
[etc] [other words]
[etc] [other words]

```
inputRDD.map(removeunicode)
UnicodeRDD.map(split_words).flatMap(lambda x: x)
```

Narrow Transformations

(etc,1)
(etc,1)
(cton.d,1)
(fnome-vfs-2.0,1)….

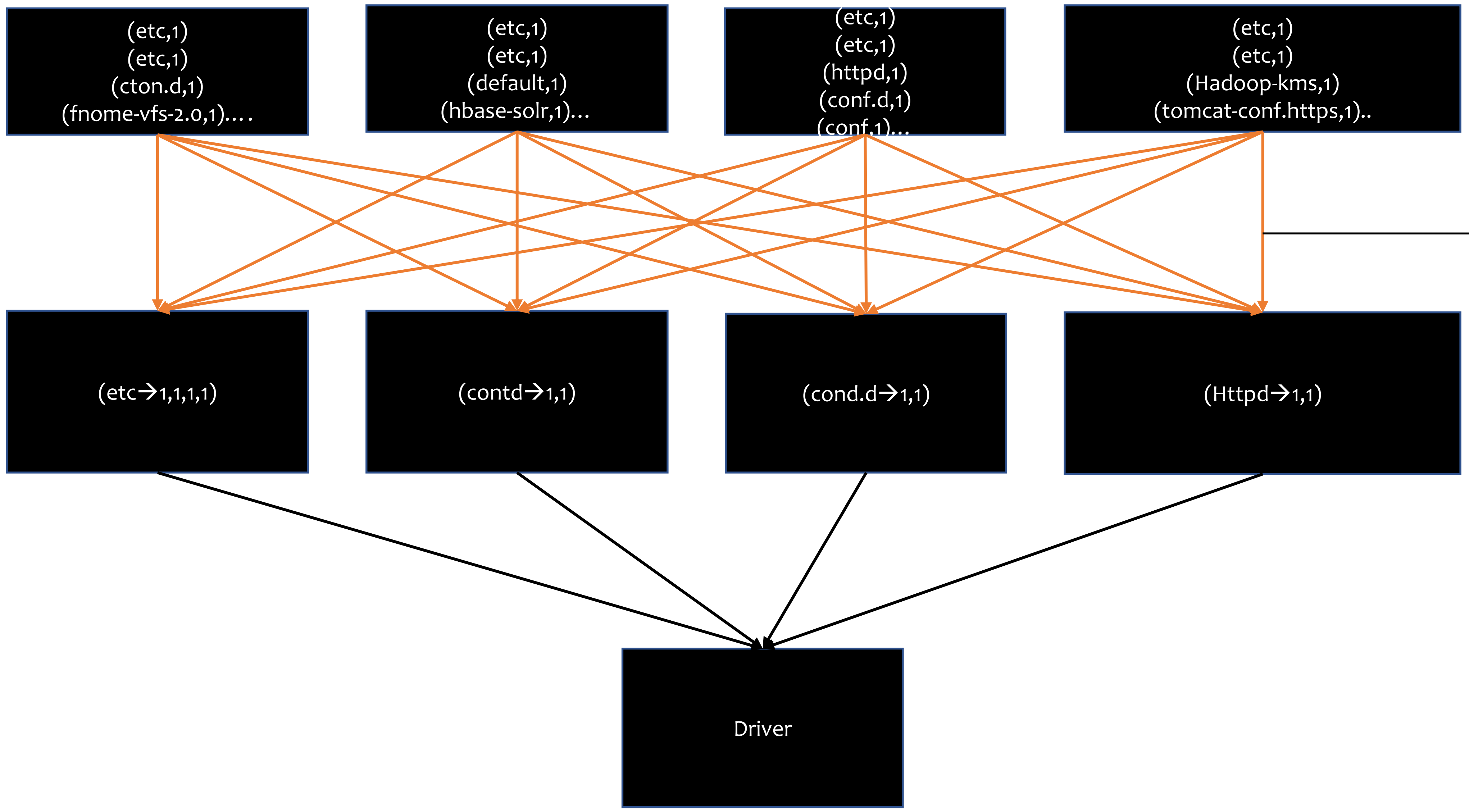
(etc,1)
(etc,1)
(default,1)
(hbase-solr,1)…

(etc,1)
(etc,1)
(httpd,1)
(conf.d,1)
(conf,1)…

(etc,1)
(etc,1)
(Hadoop-kms,1)
(tomcat-conf.https,1)..

```
WordRDD.map(lambda word: (word,1))
```

Narrow Transformation



Shuffling

wordPairRDD.reduceByKey(lambda x,y:x+y)
Wide Transformation

wordcountRDD.take(100)

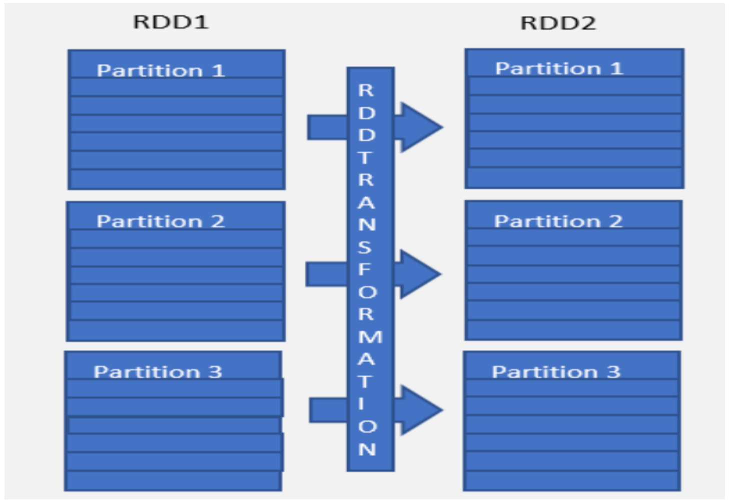
Transformations

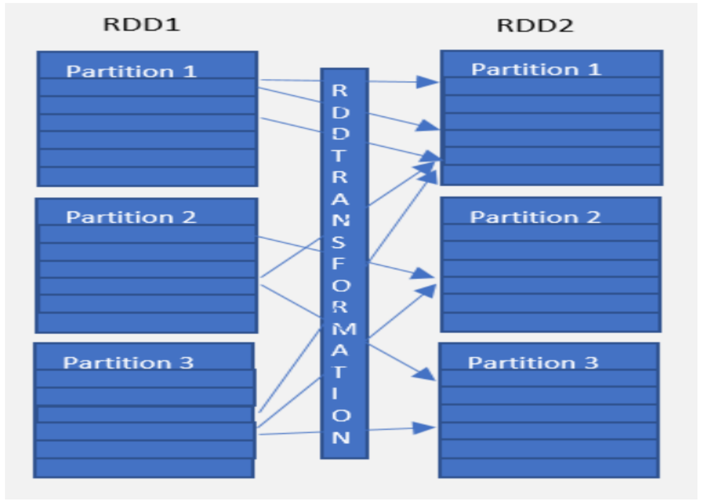
Transformations

Narrow:
These types of transformations convert each input partition to only one output partition. When each partition at the parent RDD is used by at most one partition of the child RDD or when each partition from child produced or dependent on single parent RDD.
Example: Map and Filter

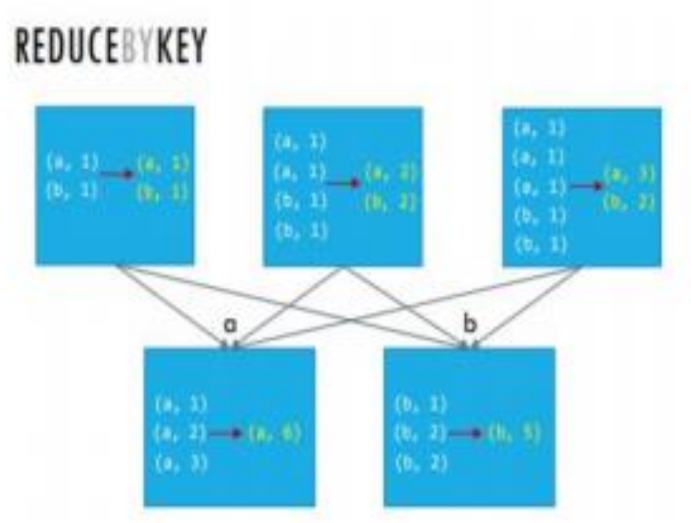
Wide:
This type of transformation will have input partitions contributing to many output partitions. When each partition at the parent RDD is used by multiple partitions of the child RDD or when each partition from child produced or dependent on multiple parent RDD.

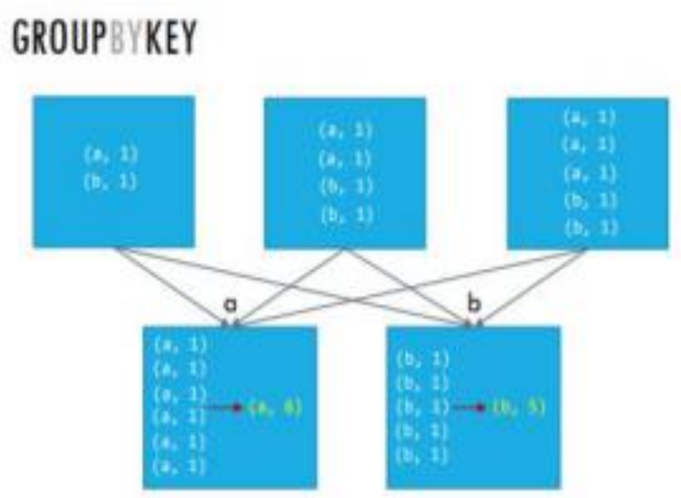
- **Might Require data shuffling over the cluster network or no data movement.**
- **Examples: groupByKey(), aggregateByKey(), aggregate(), join(), repartition()**





Transformations





reduceBykey		groupByKey
Uses Combiner		Do not uses Combiner
Take one parameter as function – for seqOp and combOp		No parameters as functions. Generally followed by map or flatMap
Implicit Combiner		No combiner
Performance is high for aggregations		Relatively slow for aggregations

- → Both wide Transformations has to be performed on paired RDD only
- → Before Shuffling the data in the same machine ReducebyKey will perform first level transformations then the data will be sent over to the reducers for second level of transformation. However coming to GroupbyKey no first level transformation will be applied because of that all the data will be aggregated on key basis in reducer because of it lots of data will be exchanges b/w nodes through Network it causes the network congestion or increases network traffic.
- → groupbyKey + foldleftofValues = reducebyKey so when ever you want to use groupbyKey with map operation use ReducebyKey for better performance.

Data Compression

What is the Compression Ratio or Compression Power?

The term compression ratio refers to a fraction, percentage or ratio that expresses the difference between the size of a file before it was compressed and after the compression process is complete.

Data compression ratio is defined as the ratio between the uncompressed size and compressed size.

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

Sometimes the space saving is given instead, which is defined as the reduction in size relative to the uncompressed size.

$$\text{Space Saving} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}$$

These ratios are usually either expressed using qualitative terms or in a 1:10 type format. As it happens, compression rates below **1:10** are considered reasonable or **good**, while ones higher than 1:10, such as **1:12** are instead considered **excellent**. The other big factor when it comes to the compression ratio is whether or not a compression algorithm is **lossy or lossless.**

1 offers the fastest compression speed but at a lower ratio, and 10 offers the highest compression ratio but at a lower speed

Data Compression

Lossles: Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data.

Example:
- It can be advantageous to make a master lossless file which can then be used to produce additional copies from.

Lossy: lossy compression or **irreversible compression** is the class of data encoding methods that uses inexact approximations and partial data discarding to represent the content. These techniques are used to reduce data size for storing, handling, and transmitting content.

Example:
- Some times you may observer the quality of the image is compromised after high compression.
- Lossy compression is most commonly used to compress multimedia data (audio, video, and images), especially in applications such as streaming media and internet telephony.

Data Compression

Lossles: Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data.

Example:

- It can be advantageous to make a master lossless file which can then be used to produce additional copies from.

Lossy: lossy compression or **irreversible compression** is the class of data encoding methods that uses inexact approximations and partial data discarding to represent the content. These techniques are used to reduce data size for storing, handling, and transmitting content.

Example:

- Some times you may observer the quality of the image is compromised after high compression.
- Lossy compression is most commonly used to compress multimedia data (audio, video, and images), especially in applications such as streaming media and internet telephony.

Data Compression

<u>Codec:</u>

- A codec, which is a shortened form of compressor/decompressor, is technology (software or hardware, or both) for compressing and decompressing data; it’s the implementation of a compression/decompression algorithm.

<u>Splitable</u>

- You need to know that some codecs support something called **splitable compression** and that codecs differ in both the speed with which they can compress and decompress data and the degree to which they can compress it.

- Splitable compression is an important concept in a Hadoop context. The way Hadoop works is that files are split if they’re larger than the file’s block size setting, and individual file splits can be processed in parallel by different mappers.

<u>Resources:</u>

- You can measure the performance of the codec by looking at the usage of Disk and CPU.
- The most common trade-off is between compression ratios (the degree to which a file is compressed) and compress/decompress speeds.

<u>Advantages</u>

- If the input file to a spark job contains compressed data, the time that is needed to read that data from HDFS is reduced and job performance is enhanced. The input data is decompressed automatically when it is being read by executer.

Compression Formats

Compressi on format	extension name	Multi-file	Support sliced	Top compression ratio	Decompression speed Ranking	tool	hadoop comes
gzip	.gz	no	no	2	3	gzip	Yes
bzip2	.bz2	Yes	Yes	1	4	bzip2	Yes
lzo	.lzo	no	Yes	3	2	lzop	no
snappy	.snappy	no	no	4	1	no	no

Compression format	Compression ratio	Compression rate	Decompression rate
gzip/deflate	13.4%	21 MB/s	118 MB/s
bzip2	13.2%	2.4MB/s	9.5MB/s
lzo	20.5%	135 MB/s	410 MB/s
snappy	22.2%	172 MB/s	409 MB/s

Compression ratio: BZip2> GZip> Lzo> Snappy **Compression rate**: Snappy> Lzo> GZIp> BZip2

Compression Formats

gzip:

Applies to the processing of compressed file size within 120M (the standard block size of haoop2 is 120M), which can effectively improve the concurrency of reading.

Storage Space: Medium
CPU Usage: Medium
Splittable: No

GZip is often a good choice for **cold data**, which is accessed infrequently.

--

bzip2 :

Due to splitable and multi file behaviour it can be applied on the bigger file.
Storage Space: Low
CPU Usage: Medium
Splittable: No

not suitable for speeder access.

bzip2 is often a good choice for **cold data**, which is accessed infrequently.

Compression Formats

<u>lzo:</u>
The compression/decompression speed is also faster and the compression ratio is reasonable.

Storage Space: High
CPU Usage: Low
Splittable: Yes

lzo is often a good choice for **hot data**, which is accessed frequently.

<u>Snappy :</u>
High compression speed and reasonable compression ratio.

Storage Space: High
CPU Usage: Low
Splittable: No

not suitable for speeder access.

bzip2 is often a good choice for **hot data**, which is accessed infrequently.

How many Tasks?

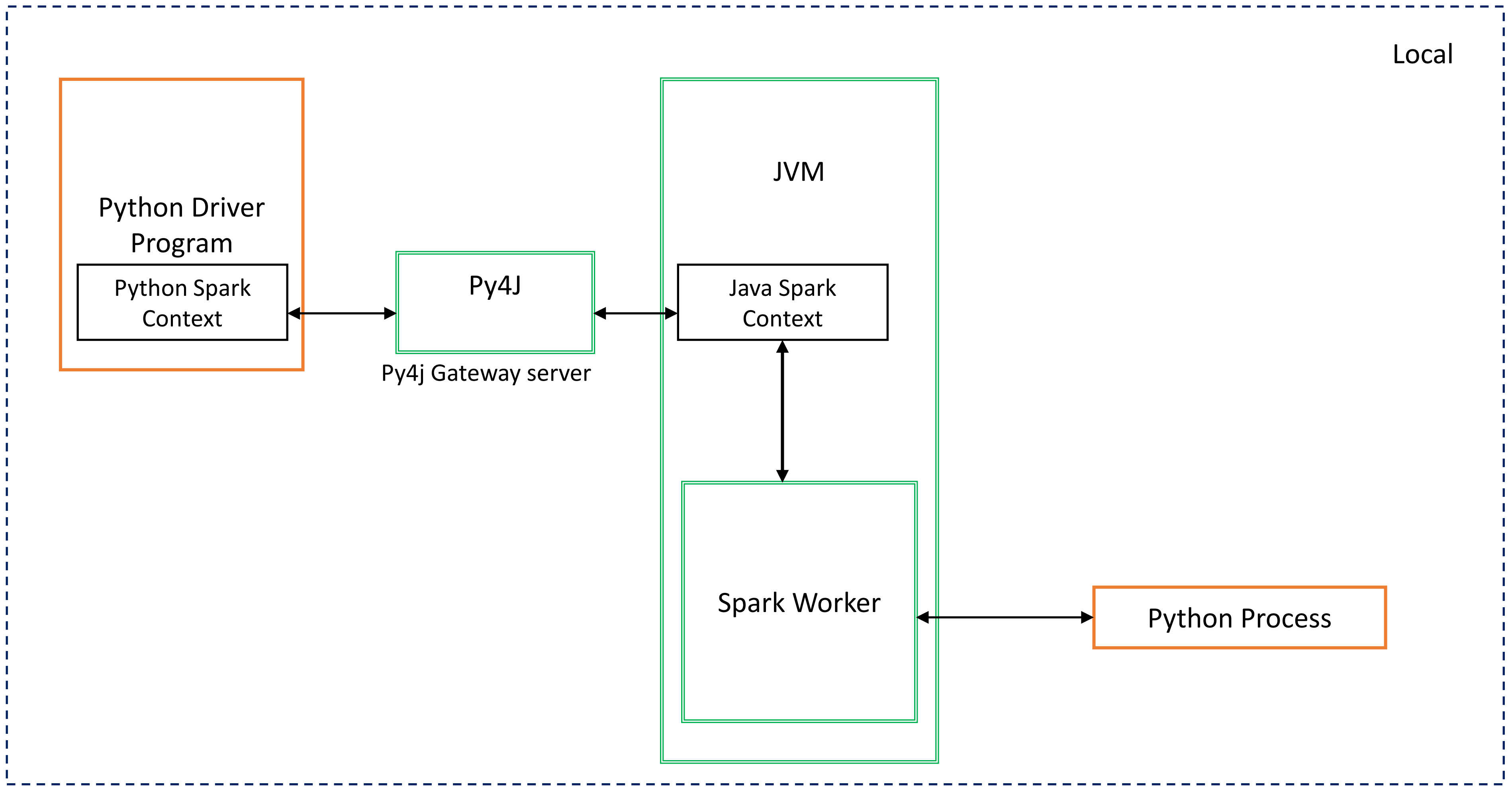
We Understand that Number of Stages depends on types of Transformations used in the Job

What about Tasks?

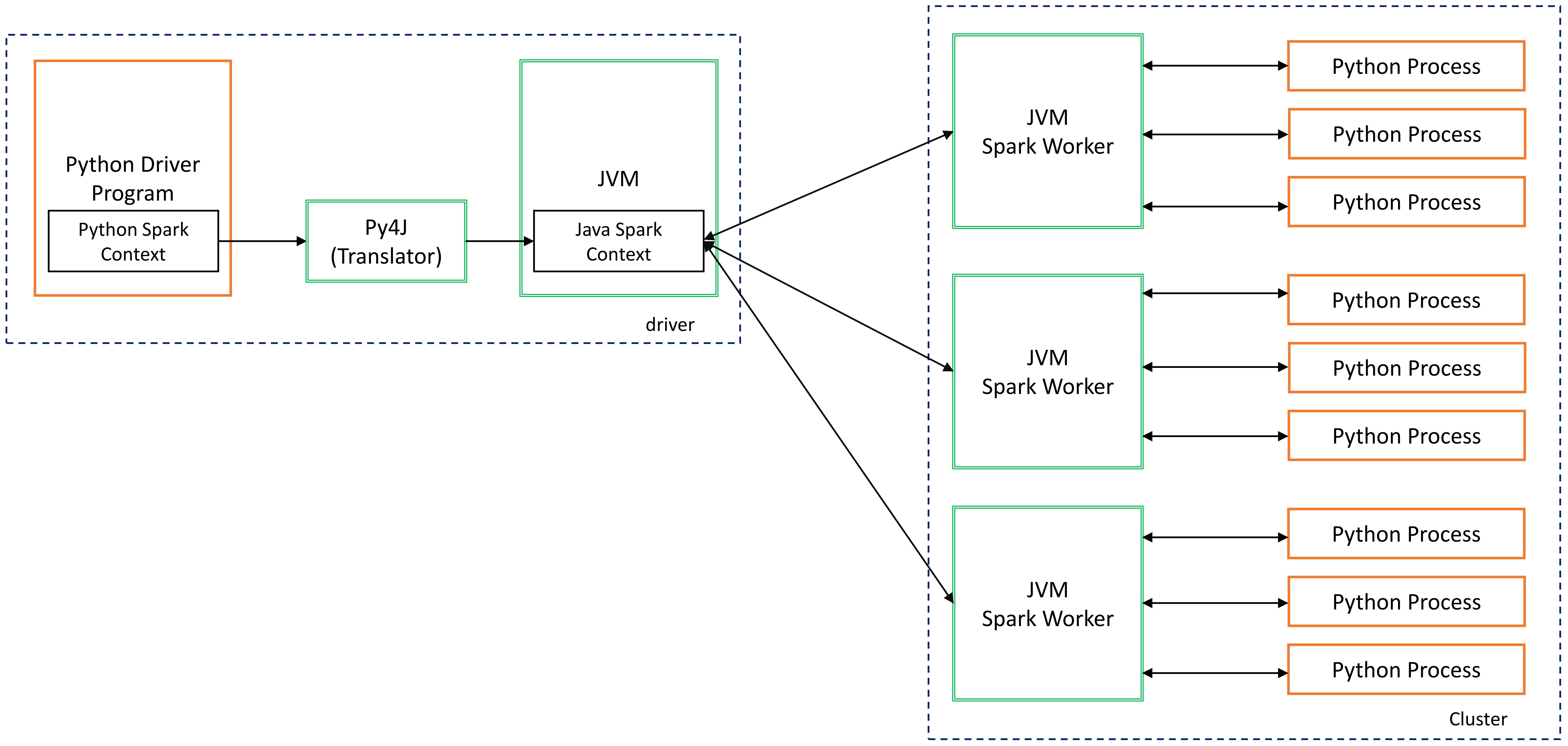
Number of Partitions = Number of Tasks. Lets Understand a bit about this

Example2:

What happens when pyspark job is submitted in local?



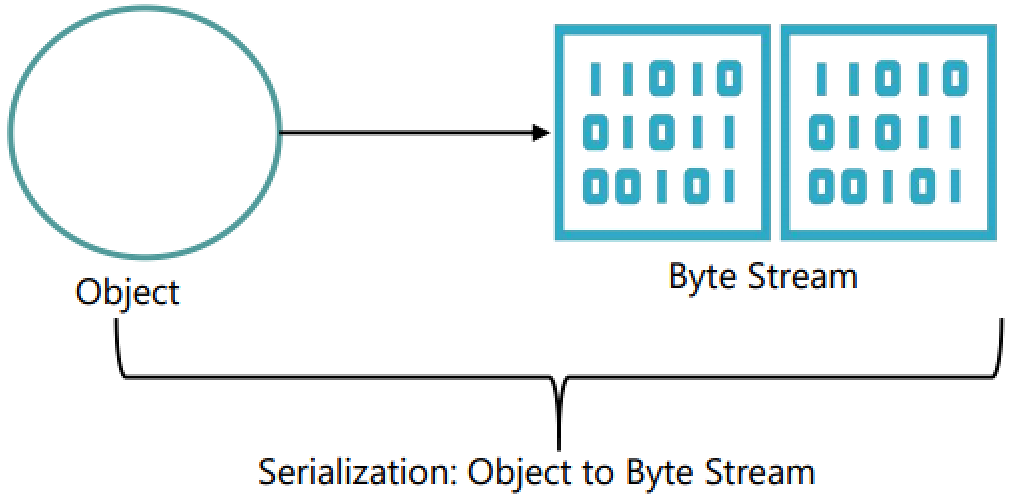
What happens when pyspark job is submitted in cluster?

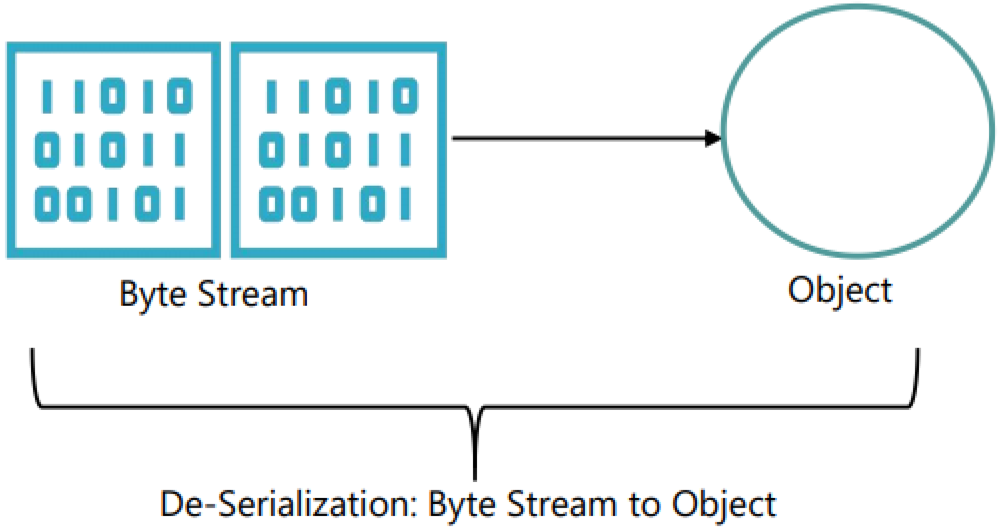


Prudhvi Akella@2021

Python Serializer and De-Serializer

Serialization and De- Serialization

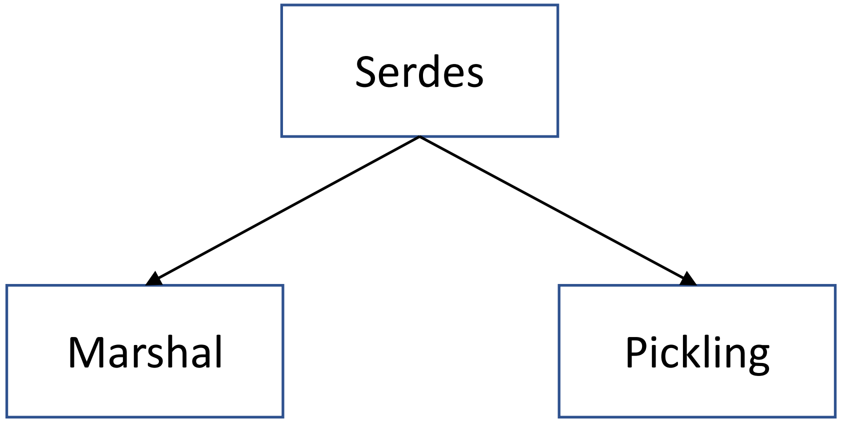




When is it Required?

usually used when the need arises to
→send your data over network
→stored in files.

Types of Serializations in Pyspark



Marshal

→ **The marshal module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of .pyc files.**

→"This module doesn't support all Python object types(Custom Types)"

→ Supported Python types:
booleans, integers, floating point numbers, complex numbers, strings, bytes, bytearrays, tuples, lists, sets, frozensets, dictionaries, and code objects

→Marshal Methods:

marshal.version :
It indicates the format used by the module.

→Version 0 – Historical format
→Version 1 – Shares interned strings
→Version 2 – Uses a binary format for floating point numbers
→Version 3 – Support for object instancing and recursion
→Version 4 – Current Version

marshal.dumps(value[, version]) : The function returns the bytes object that would be written to a file by dump(value, file). The version argument indicates the data format that dumps should use. A ValueError exception is raised if the value has (or contains an object that has) an unsupported type.

marshal.loads(bytes) : This function can reconstruct the data by converting the bytes-like object to a value. EOFError, ValueError or TypeError is raised if no value is found.

Pickle

Pickling: Serialization Un-Pickling: De- Serialization

→ dumps() – This function is called to serialize an object hierarchy.
→ loads() – This function is called to de-serialize a data stream.

For more control over serialization and de-serialization, Pickler or Unpickler objects are created respectively.

Caching

Caching or persistence are optimisation techniques for (iterative and interactive) Spark computations. They help saving interim partial results so they can be reused in subsequent stages. These interim results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated.

RDDs can be cached using **cache** operation. They can also be persisted using **persist** operation.

The difference between cache and persist operations is purely syntactic. cache is a synonym of persist or persist(MEMORY_ONLY), i.e. cache is merely persist with the default storage level MEMORY_ONLY.

getStorageLevel(): Returns storagelevel information of an RDD.

Example: trans1.getStorageLevel()
Output: StorageLevel(False, False, False, False, 1)

<u>StorageLevel class definition:</u>
StorageLevel(disk=false, memory=false, offheap=false, deserialized=false, replication=1)

Caching: StorageLevel

StorageLevel describes how an RDD is persisted (and addresses the following concerns):

- Does RDD use disk?
- How much of RDD is in memory?
- Does RDD use off-heap memory?
- Should an RDD be serialized (while persisting)?
- How many replicas (default: 1) to use (can only be less than 40)?

Caching: StorageLevel

There are the following StorageLevel (number _2 in the name denotes 2 replicas):

- NONE (default)
- DISK_ONLY
- DISK_ONLY_2
- MEMORY_ONLY (default for cache operation for RDDs)
- MEMORY_ONLY_2
- MEMORY_ONLY_SER
- MEMORY_ONLY_SER_2
- MEMORY_AND_DISK
- MEMORY_AND_DISK_2
- MEMORY_AND_DISK_SER
- MEMORY_AND_DISK_SER_2
- OFF_HEAP

MEMORY_ONLY:
Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.

MEMORY_AND_DISK:
Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.

MEMORY_ONLY_SER:
Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.

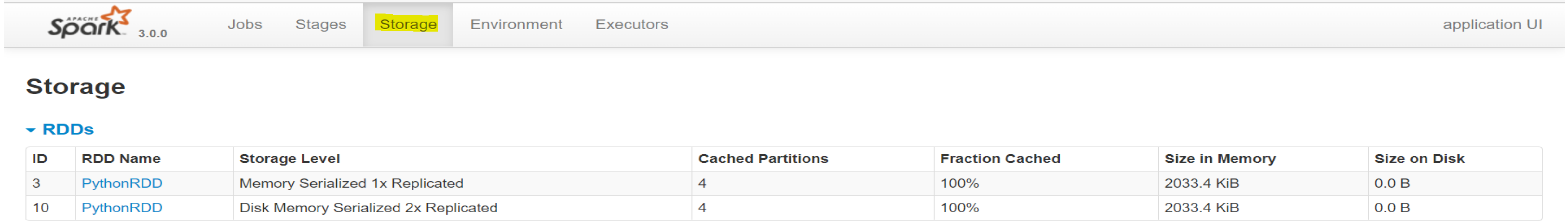
MEMORY_AND_DISK_SER:
Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.

DISK_ONLY: Store the RDD partitions only on disk.

MEMORY_ONLY_2, MEMORY_AND_DISK_2: Same as the levels above, but replicate each partition on two cluster nodes.

Caching: StorageLevel

```
Storage Level     Space used  CPU time  In memory  On-disk  Serialized   Recompute some partitions
----------------------------------------------------------------------------------------------------
MEMORY_ONLY          High        Low       Y          N        N          Y
MEMORY_ONLY_SER      Low         High      Y          N        Y          Y
MEMORY_AND_DISK      High        Medium    Some       Some     Some       N
MEMORY_AND_DISK_SER  Low         High      Some       Some     Y          N
DISK_ONLY            Low         High      N          Y        Y          N
```

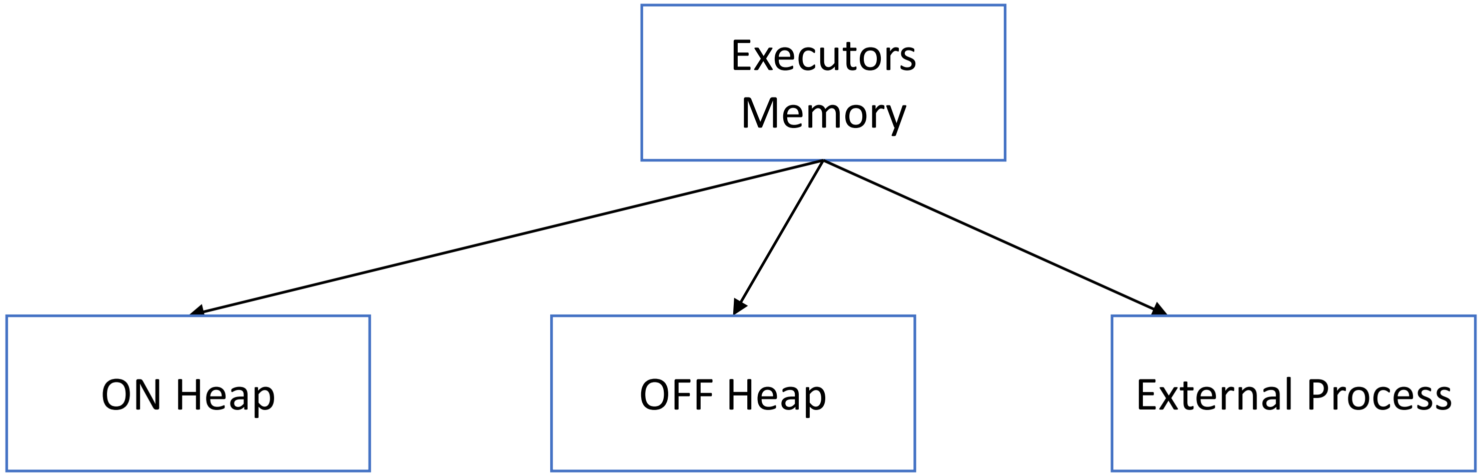


Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
3	PythonRDD	Memory Serialized 1x Replicated	4	100%	2033.4 KiB	0.0 B
10	PythonRDD	Disk Memory Serialized 2x Replicated	4	100%	2033.4 KiB	0.0 B

Spark Memory Management

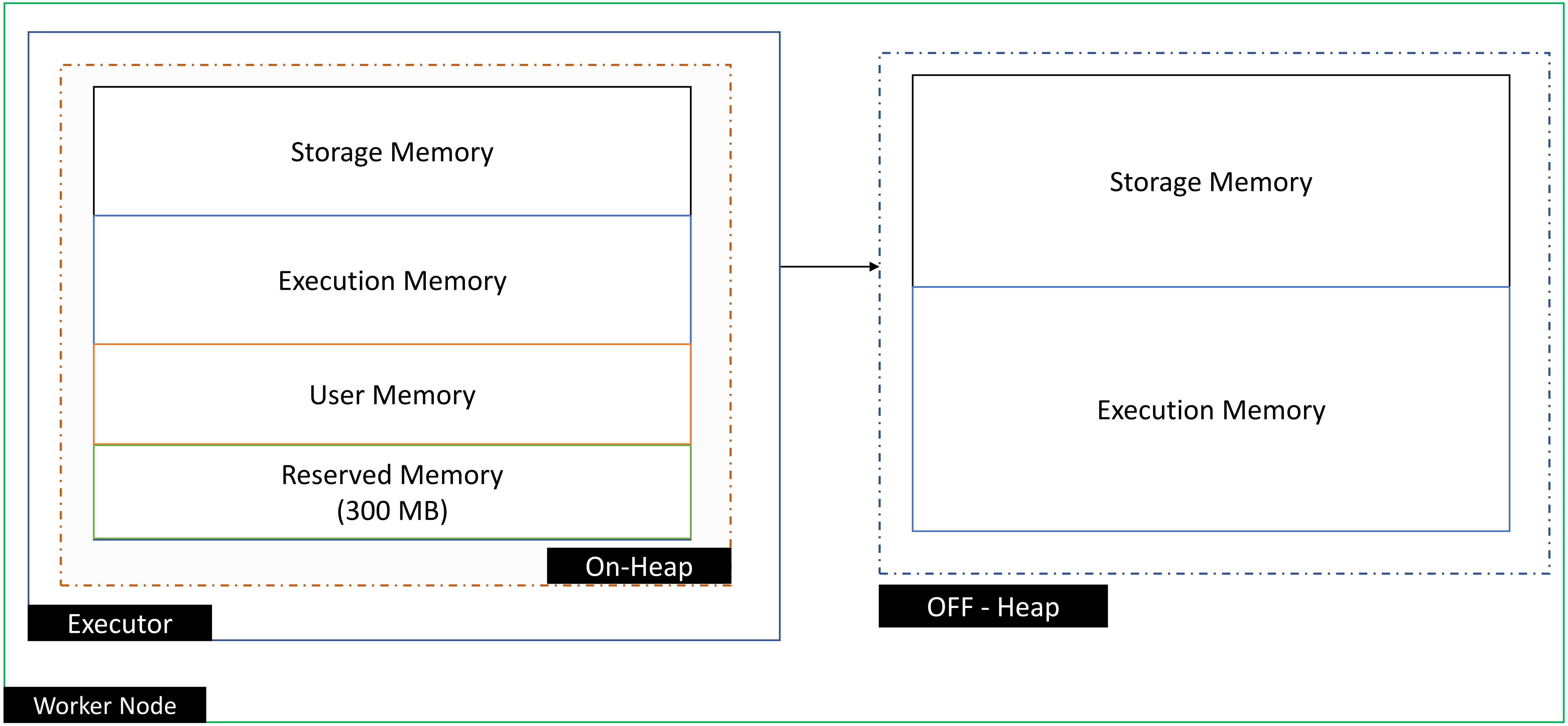


ON Heap: Objects are created on JVM heap will be controlled by Garbage collector.

OFF Heap : Objects are serialized and stored outside the JVM's memory. These objects are not bounded to GC. Application is responsible of de-allocating Memory. The advantages of using OFF Heap memory is to avoid the frequent GC cycles.

External Process : Specific to PySpark and SparkR, this is the memory used by the python/R process which resides outside of the JVM.

Spark Memory Management



Prudhvi Akella@2021

Spark Memory Management: ON-Heap

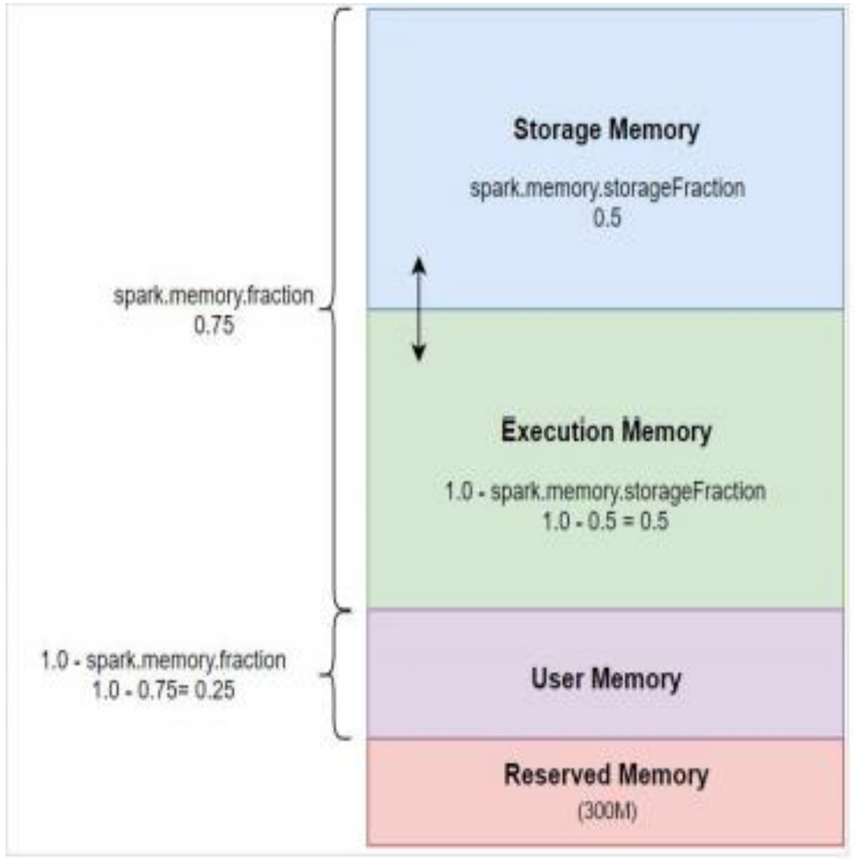
On-Heap Memory is divided into 4 parts.

Storage Memory: Its mainly used to store the spark cache data such as RDD cache, Broadcast variables.

Execution Memory: Its mainly used to store the temporary data in the calculation process of Shuffles(ByKey Transformations, Joins), Sorts ect.

User Memory: Its Mainly used to store the data needed for RDD conversion operations, such as RDD dependencies

Reserved Memory: The memory is reserved for the system and its used to store the spark's Internal objects



Spark Memory Management: OFF-Heap

spark.memory.offHeap.use : false (default)
spark.memory.offHeap.size : The amount of off-heap memory used by Spark to store actual data frames/RDD

Spark uses off-heap memory for two purposes:

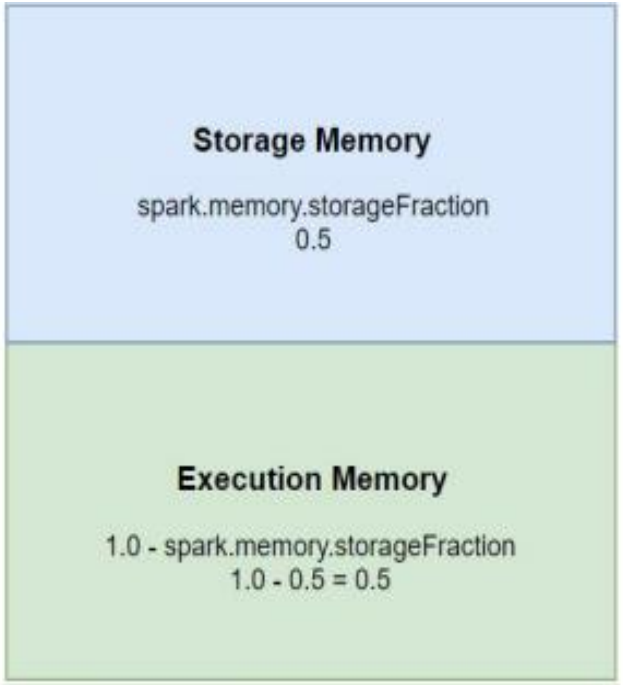
- A part of off-heap memory is used by Java internally for purposes like String interning and JVM overheads.
- Off-Heap memory can also be used by Spark explicitly for storing its data as part of Project Tungsten.

Note:
<u>Till Spark 2.4.5:</u>

The total off-heap memory for a Spark executor is controlled by spark.executor.memoryOverhead. The default value for this is 10% of executor memory subject to a minimum of 384MB. This means, even if the user does not explicitly set this parameter, Spark would set aside 10% of executor memory(or 384MB whichever is higher) for VM overheads.

<u>From Spark 3.0 </u>:

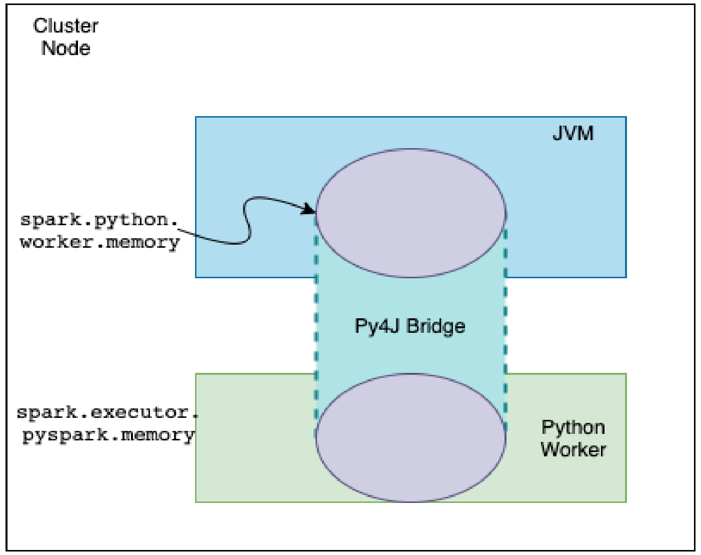
Spark 3.0 makes the Spark off-heap a separate entity from the memoryOverhead, so users do not have to account for it explicitly during setting the executor memoryOverhead.


Storage Memory
spark.memory.storageFraction
0.5
Execution Memory
1.0 - spark.memory.storageFraction
1.0 - 0.5 = 0.5

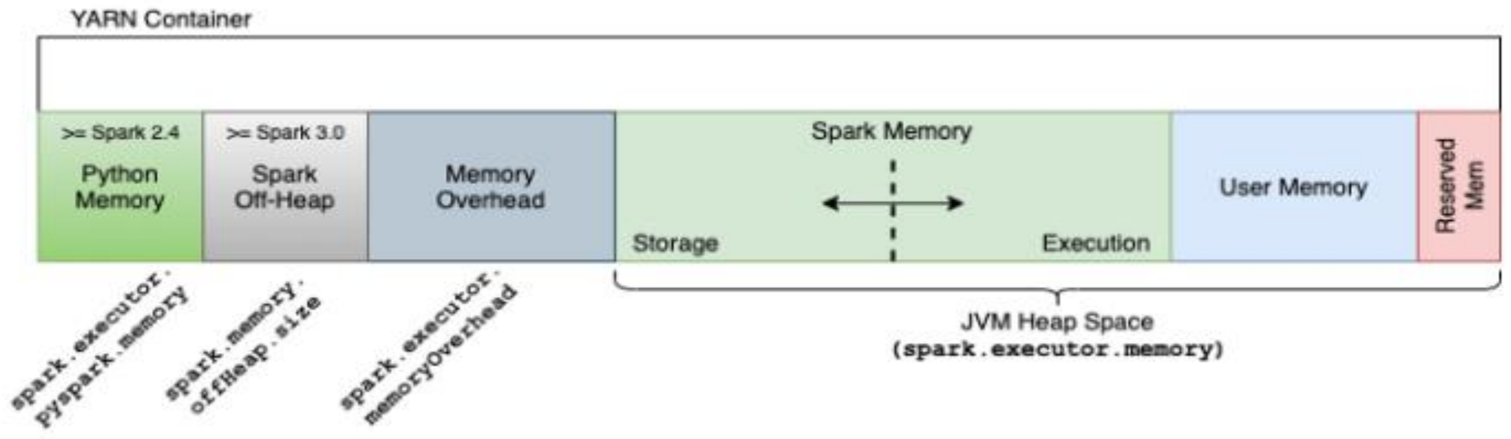
Spark Memory Management: Python Memory

spark.python.worker.memory:
controls the amount of memory reserved for each pyspark worker beyond which it spills over to the disk. In other words, it is the amount of memory that can be occupied by the objects created via the Py4J bridge during a Spark operation. In case this parameter is not set, the default value is 512MB.

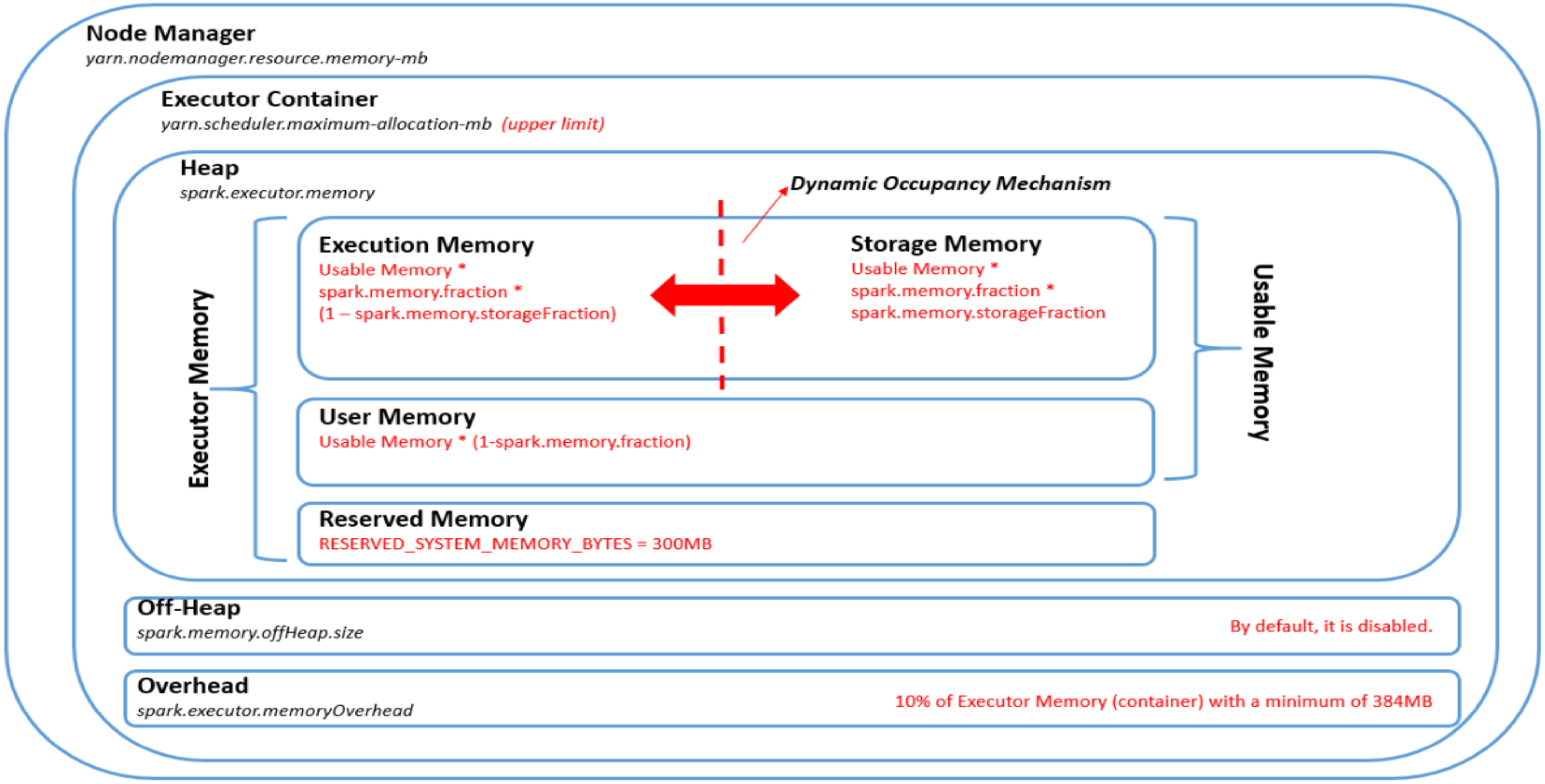
spark.executor.pyspark.memory:
the python worker process can potentially occupy the entire node's memory. Since this portion of memory is not tracked by YARN, this might lead to over-scheduling in the node (because YARN assumes the memory occupied by the python worker to be free). This can lead to page-swaps in the memory and slow down all the YARN containers on that node.



Spark Memory Management: Total Container Memory

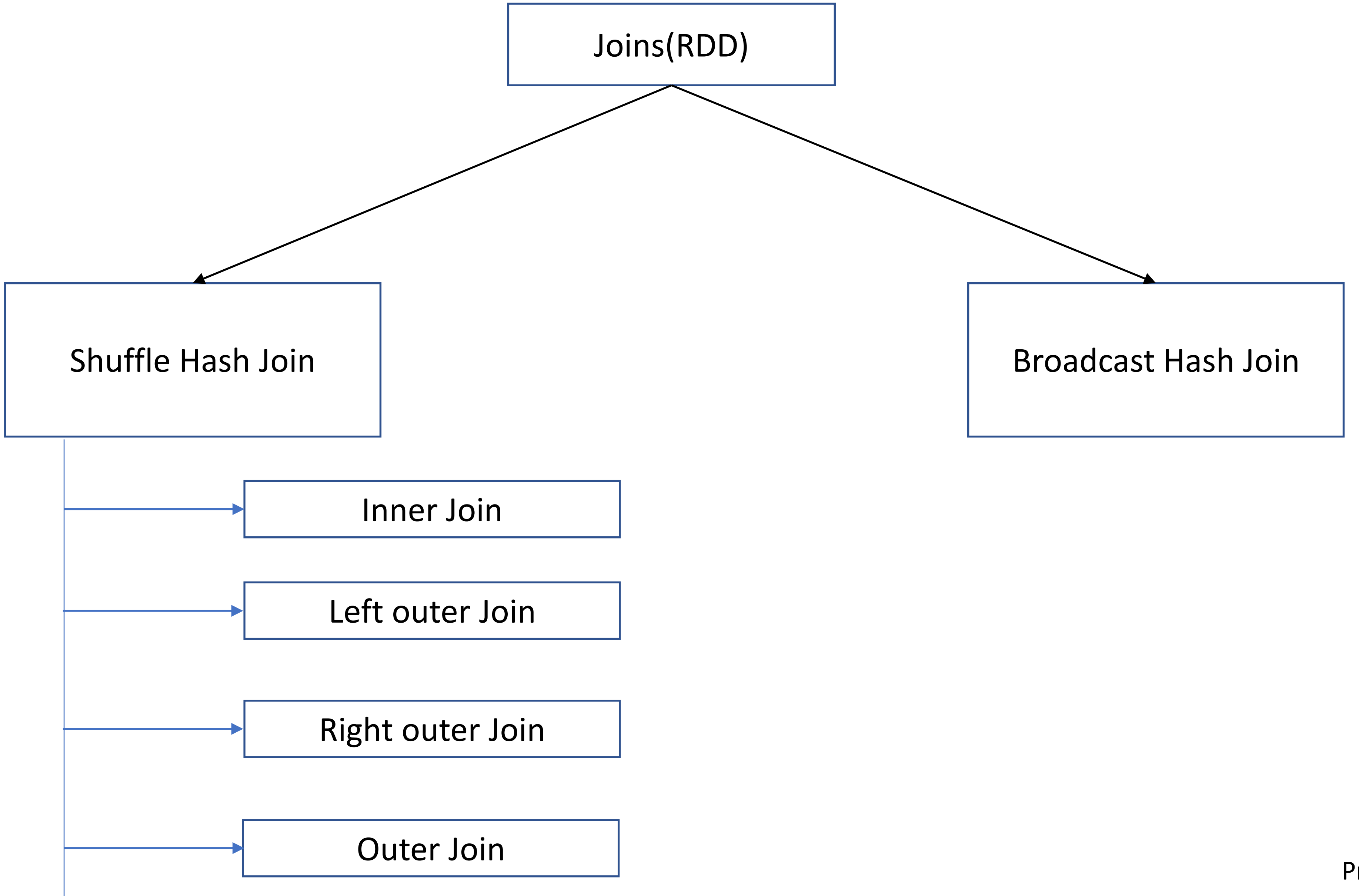


Spark Memory Management: Total Container Memory


Node Manager
yarn.nodemanager.resource.memory-mb
Executor Container
yarn.scheduler.maximum-allocation-mb (upper limit)
Heap
spark.executor.memory
Dynamic Occupancy Mechanism
Execution Memory
Usable Memory * spark.memory.fraction * (1 – spark.memory.storageFraction)
Storage Memory
Usable Memory * spark.memory.fraction * spark.memory.storageFraction
User Memory
Usable Memory * (1-spark.memory.fraction)
Reserved Memory
RESERVED_SYSTEM_MEMORY_BYTES = 300MB
Executor Memory
Usable Memory
Off-Heap
spark.memory.offHeap.size
By default, it is disabled.
Overhead
spark.executor.memoryOverhead
10% of Executor Memory (container) with a minimum of 384MB

Prudhvi Akella@2021

Joins


Joins(RDD)
Shuffle Hash Join
Broadcast Hash Join
Inner Join
Left outer Join
Right outer Join
Outer Join

Joins: Shuffle Hash Join

Joins in general are expensive since they require that corresponding keys from each RDD are located at the same partition so that they can be combined locally.

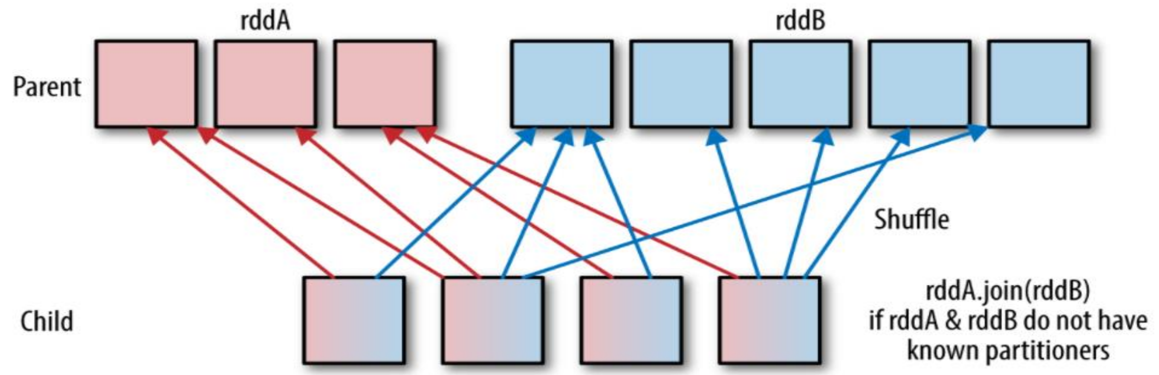


Figure 4-1. Shuffle join

The default implementation of a join in Spark is a shuffled hash join.. The shuffled hash join ensures that data on each partition will contain the same keys by partitioning the second dataset with the same default partitioner as the first, so that the keys with the same hash value from both datasets are in the same partition. While this approach always works, it can be more expensive than necessary because it requires a shuffle.

The shuffle can be avoided if:

- Both RDDs have a known partitioner.
- One of the datasets is small enough to fit in memory, in which case we can do a broadcast hash join (we will explain what this is later).

Shared Variables

Spark has two types of shared variables

- broadcast variables(Read only variables)
- Accumulator (Write only variables)

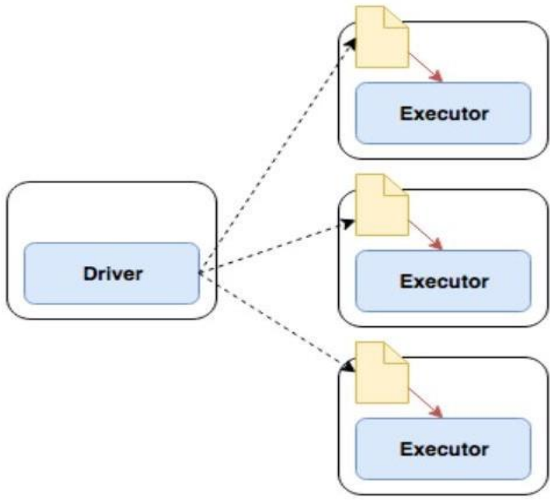
- Each of which can only be written in one context (driver or worker, respectively) and read in the other.
- Broadcast variables can be written in the driver program and read on the executors
- accumulators are written onto the executors and read on the driver.

Broadcast variables:

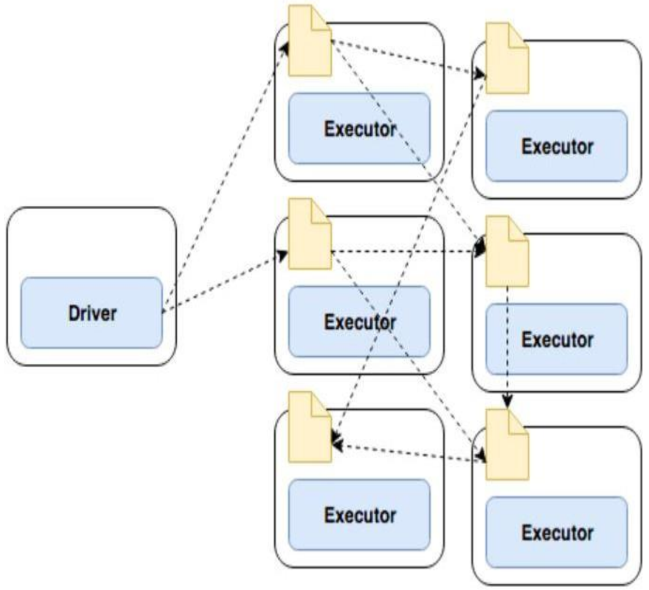
When Spark sees the use of a broadcast variable in your code, Spark will serialize the data and send it to all executors involved in your application. The broadcast variables are cached on the executor side and all tasks in the application will have access to the data in the broadcast variable.

Broadcast variables are read-only variable cached on each machine.

Spark uses torrent protocol while broadcasting a variable to executors. The Torrent protocol is a Peer-to-Peer protocol which is know to perform very well for distributing data sets across multiple peers. The advantage of the Torrent protocol is that peers share blocks of a file among each other(Executors) not relying on a central entity (Driver) holding all the blocks.



First driver will copy it to set of executors



Executors will copy within them selves

Prudhvi Akella@2021

Joins: Broadcast Hash Join

A broadcast hash join pushes one of the RDDs (the smaller one) to each of the worker nodes. Then it does a map-side combine with each partition of the larger RDD. If one of your RDDs can fit in memory or can be made to fit in memory it is always beneficial to do a broadcast hash join, since it doesn't require a shuffle.

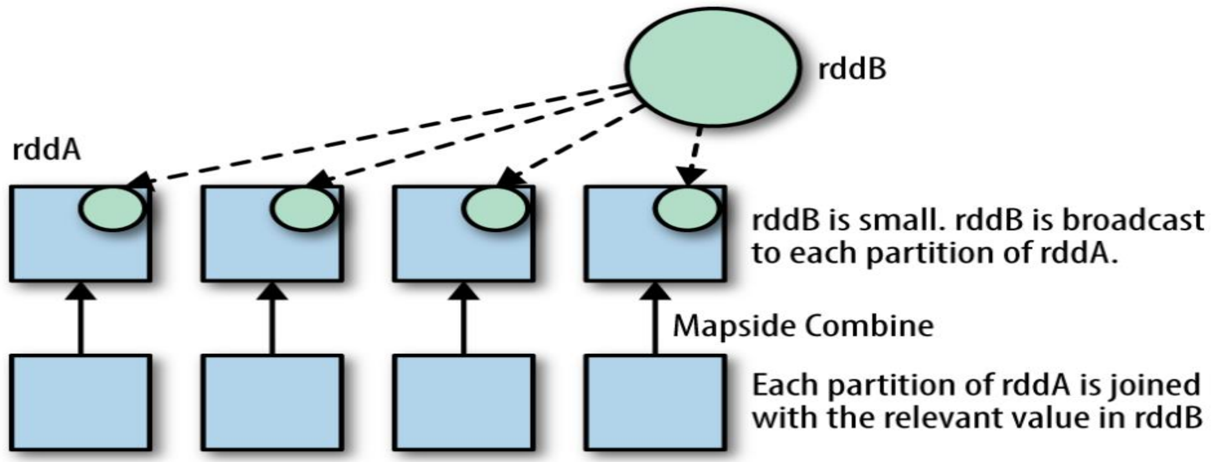

rddB
rddA
rddB is small. rddB is broadcast to each partition of rddA.
Mapside Combine
Each partition of rddA is joined with the relevant value in rddB

Figure 4-5. Broadcast hash join

Accumulators

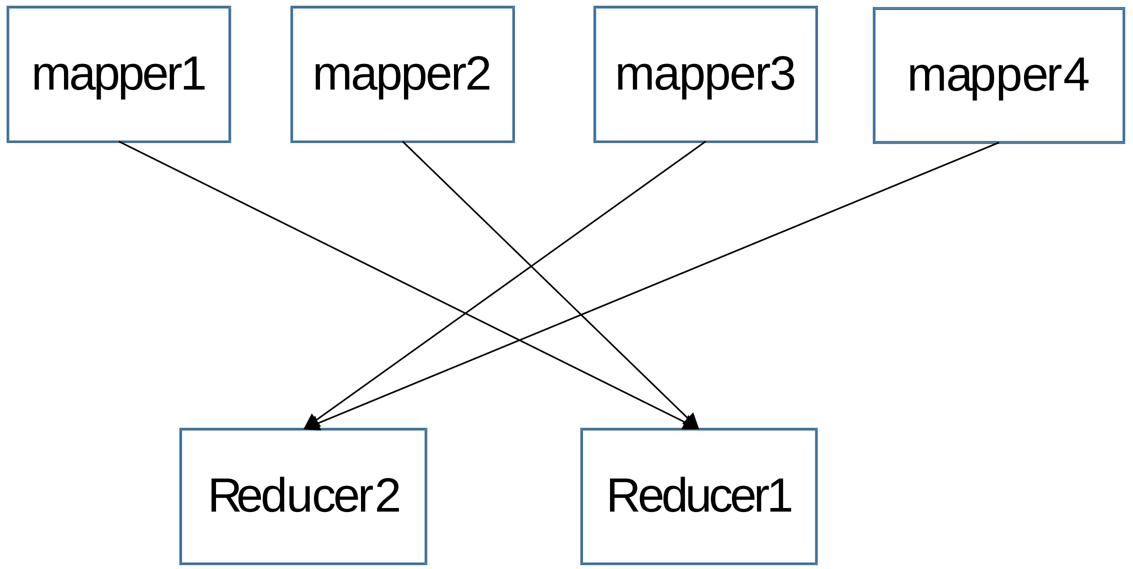
Accumulators are is one of the shared variable and write-only variables shared among executors and created with **SparkContext.accumulator** with default value ,modified with **+=** and accessed with **value** method.

Using accumulators is complicated by Spark's **run-at-least-once** guarantee for transformations. If a transformation needs to be recomputed for any reason, the accumulator updates during that transformation will be repeated. This means that accumulator values may be very different than they would be if tasks had run only once.

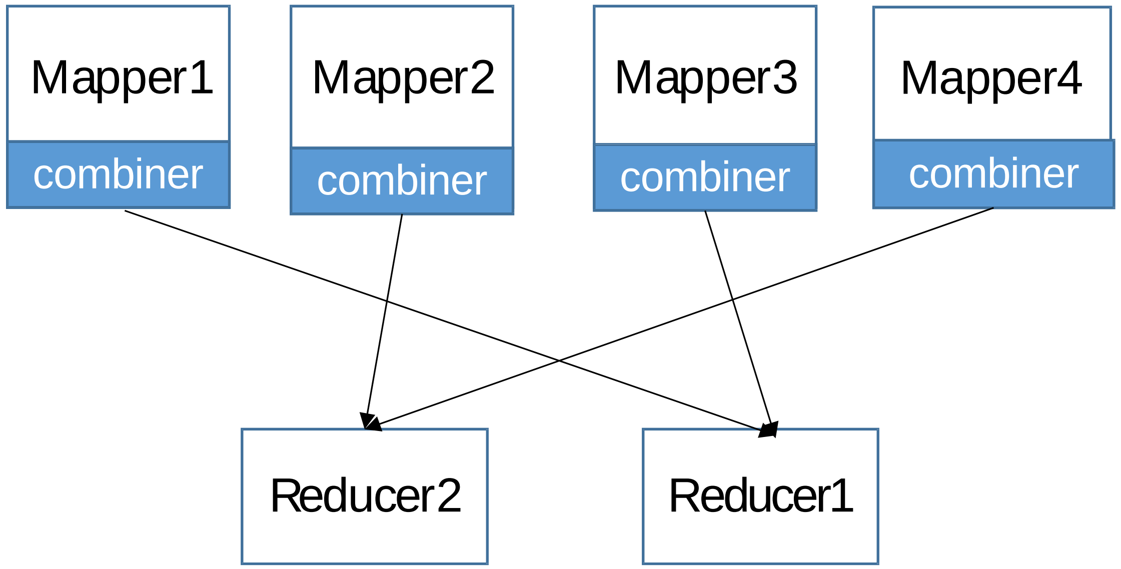
In other words, Accumulators are the write only variables which are initialized once and sent to the workers. These workers will update based on the logic written and sent back to the driver which will aggregate or process based on the logic. Only driver can access the accumulator's value.

MapPartition as Combiner

Map partition can also acts like combiner or like mini reducer that means when ever you want to perform some sort of aggregation then spark application will enter reduce phase because without having all the values that belongs to same key in same partition we cannot perform aggregation in this process lots of data will be shuffle across the network which causes network congestion to decrease that what ever the logic your reducer is doing the same thing we put in mapper phase as combiner to achieve this we use mapPartitions



Data Transfer b/w mapper and reducer are high



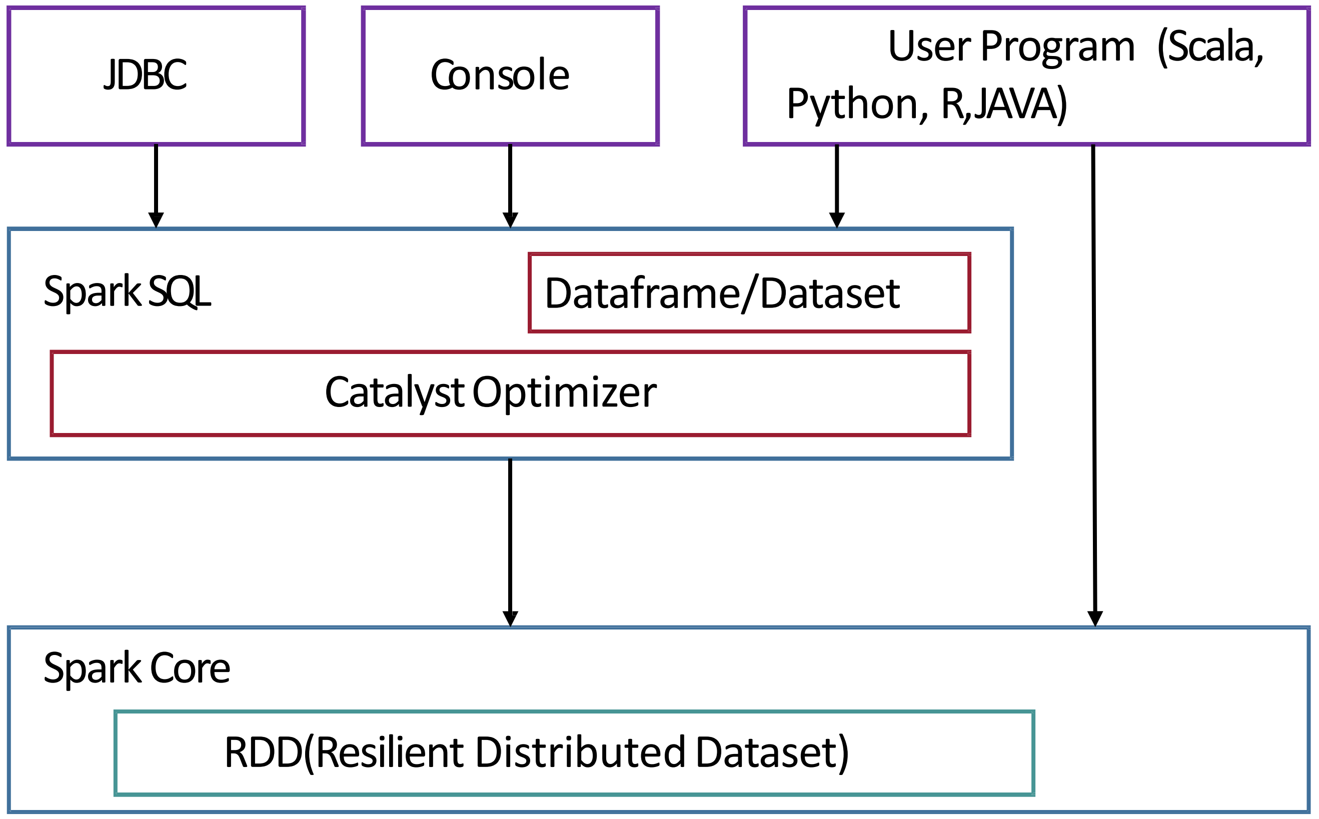
Data Transfer b/w mapper and reducer are low because combiner is acting like mini-reducer and reducing data at mapper side it self

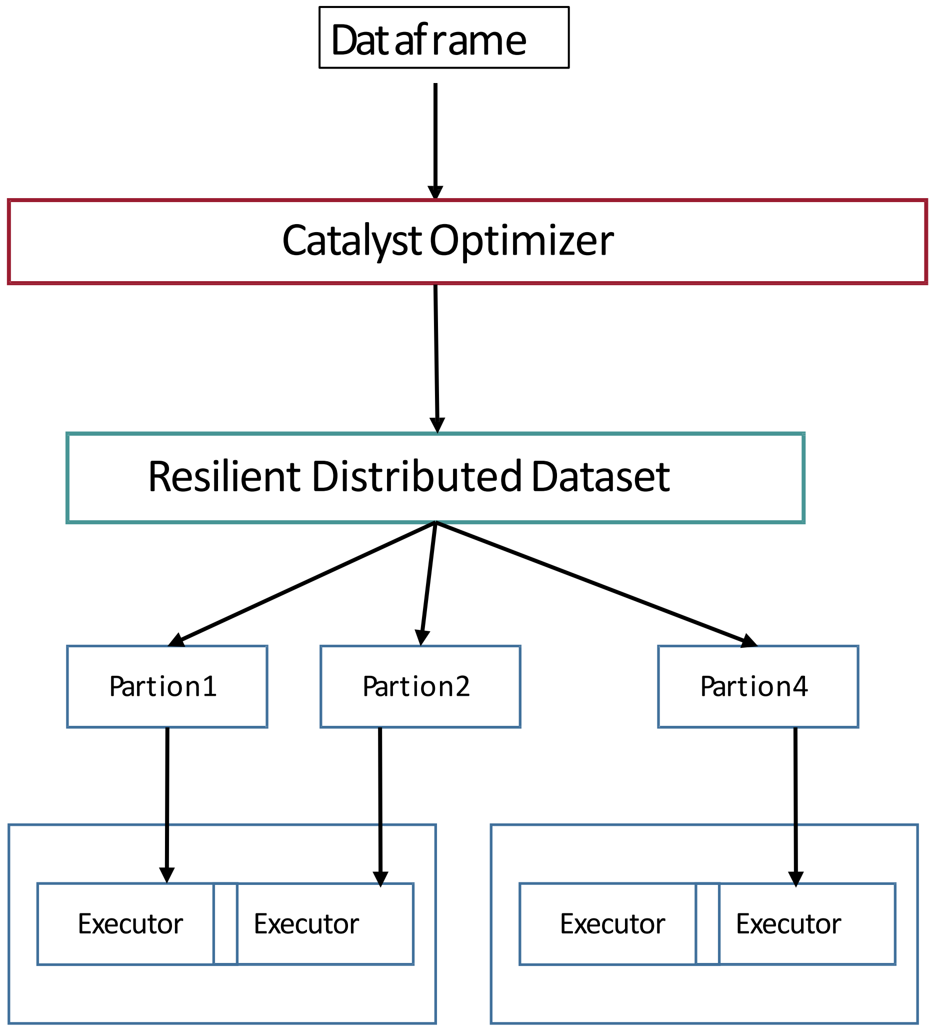
© 2020 Prudhvi Akella

Prudhvi Akella@2021

Spark SQL

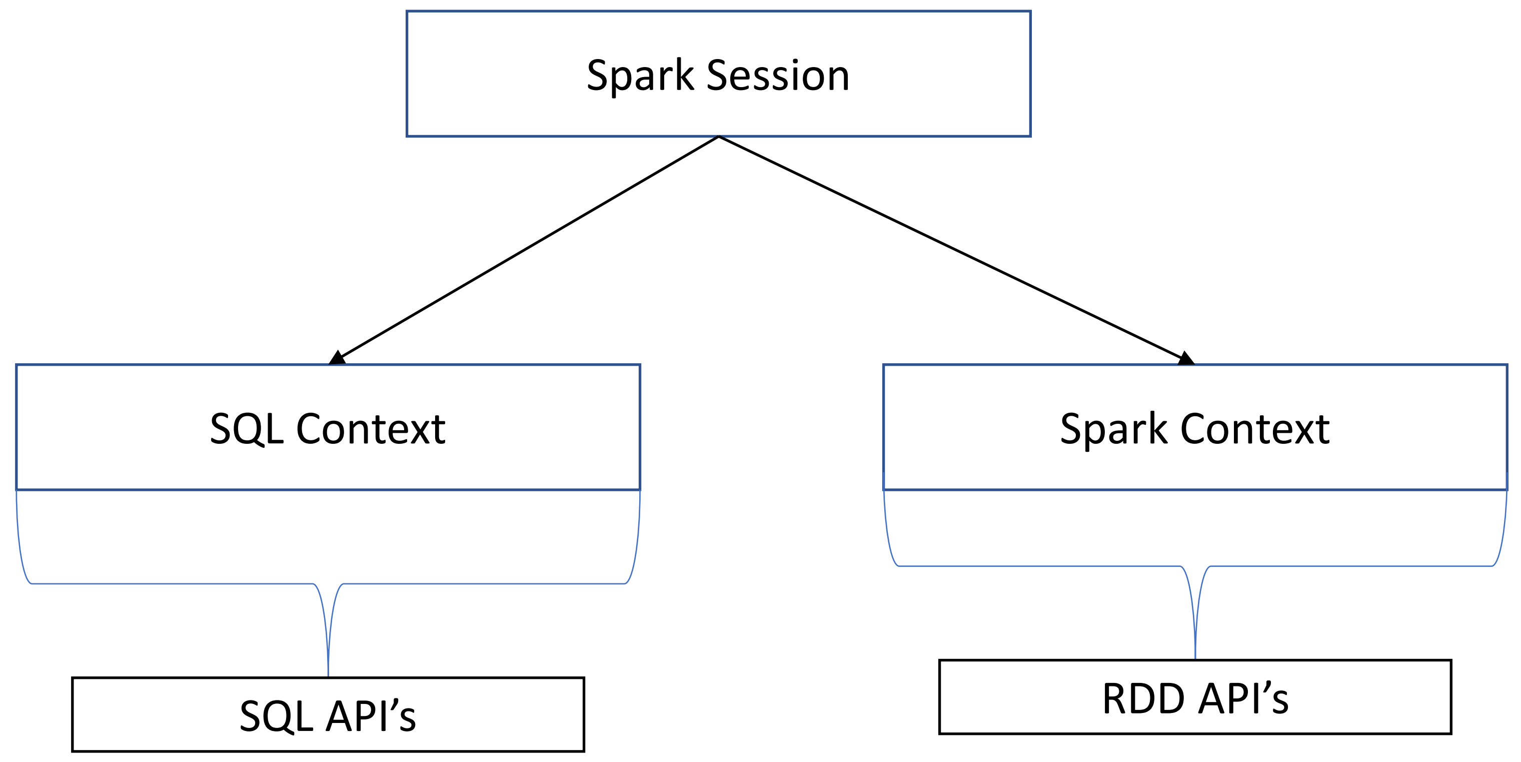
Spark SQL High level Architecture


JDBC
Console
User Program (Scala, Python, R,JAVA)
Spark SQL
Dataframe/Dataset
Catalyst Optimizer
Spark Core
RDD(Resilient Distributed Dataset)



Prudhvi Akella@2021

Spark Session



Factors effects Partitions Behaviour on Read API’s in SparkSQL

Is file splitable or not?

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)
bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

maxSplitBytes

- Each of the data file to read is a split.
- Therefore if the file is splitable with a size more than maxSplitBytes then the file is split into multiple chunk of maxSplitBytes the last chunk is being less than or equal to maxSplitBytes.
- If the files in not splitable then there will be only one file chunk/partition of size equivalent to file size self.
- Once the chunks are identified one or more chunks will be packed into one partition

Important point to Remember:

spark partition = file chunk size + openCostInBytes

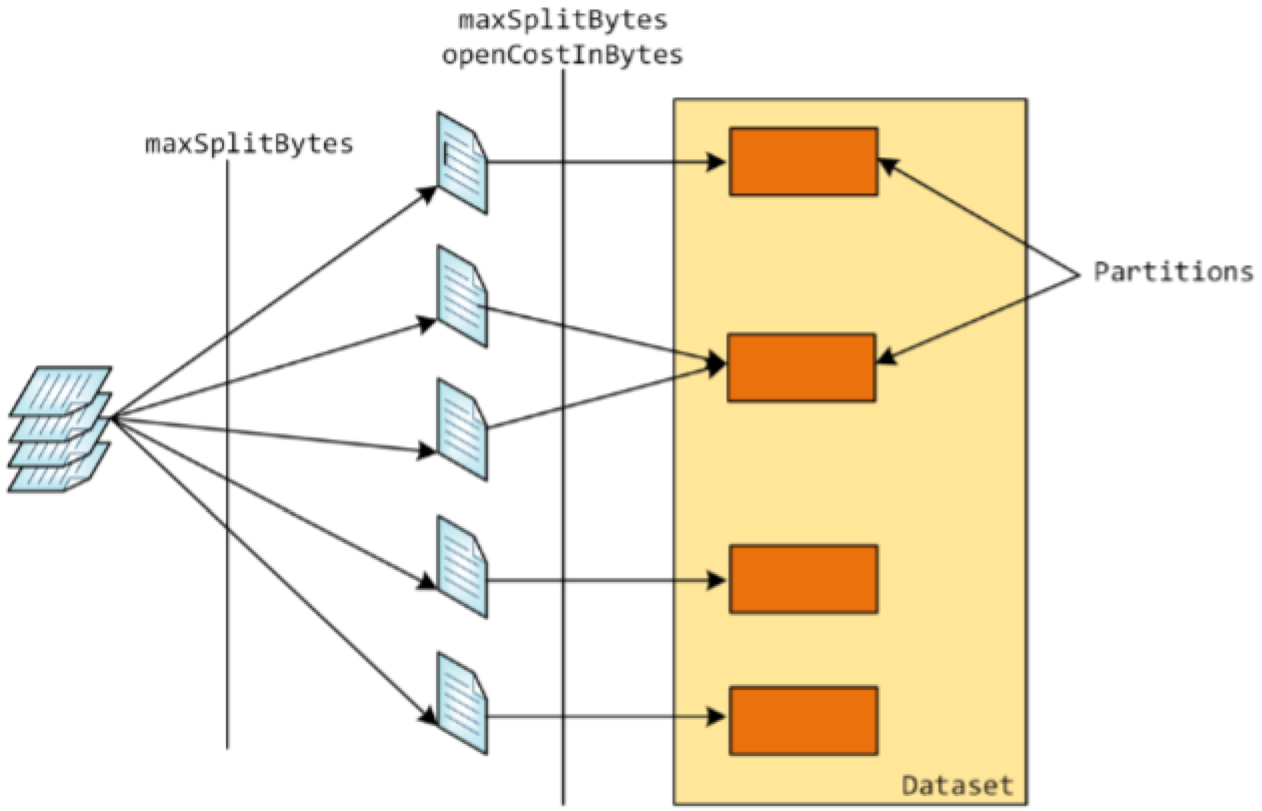


Fig. 2.1: Illustration of Procedure to split a set of input data files into the partition of a Dataset

Example1	
No of file to read	**54**
File format	**parquet**
File size	**65MB**
No of Cores	**10**
spark.sql.files.maxparititonBytes	**128MB (Default)**
spark.sql.files.openCostInBytes	**4MB(default)**
Spark.default.parallelism	**10(defaulted to number of cores)**

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)

bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

bytesperCore = ((54*65)+(54* 4))/10 = 372MB

maxSplitBytes = Min(128,372) = 128MB

The number of output partitions are 54. As two files cannot be packed into one partitions(65 + 65 = 130) so for each file one partition will be created.

Example2	
No of file to read	54
File format	parquet
File size	63MB
No of Cores	10
spark.sql.files.maxparititonBytes	128MB (Default)
spark.sql.files.openCostInBytes	4MB(default)
Spark.default.parallelism	10(defaulted to number of cores)

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)

bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

bytesperCore = ((54*63)+(54* 4))/10 = 372MB

maxSplitBytes = Min(128,361.8) = 128MB

The number of output partitions are 54. Even though two files(63 + 63 = 126) can be fit into one partition without considering openCostInBytes. After considering openCostInBytes 63 + 4 = 67MB we cannot fit both files into single partition.

Example3	
No of file to read	54
File format	parquet
File size	40MB
No of Cores	10
spark.sql.files.maxparititonBytes	128MB (Default)
spark.sql.files.openCostInBytes	4MB(default)
Spark.default.parallelism	10(defaulted to number of cores)

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)

bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

bytesperCore = ((54*40)+(54* 4))/10 = 237.6MB

maxSplitBytes = Min(128, 237.6) = 128MB

The number of output partitions are 18. In this case three file (40MB + 40MB + 40MB +4MB = 124MB) can be packed into one partition. So 54 / 3 = 18

Example4	
No of file to read	54
File format	parquet
File size	40MB
No of Cores	10
spark.sql.files.maxparititonBytes	88MB (Default)
spark.sql.files.openCostInBytes	4MB(default)
Spark.default.parallelism	10(defaulted to number of cores)

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)

bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

bytesperCore = ((54*40)+(54* 4))/10 = 237.6MB

maxSplitBytes = Min(88, 237.6) = 88MB

The number of output partitions are 18. In this case two file (40MB + 40MB +4MB = 84MB) can be packed into one partition. So 54 / 2= 27

Example5	
No of file to read	54
File format	parquet
File size	40MB
No of Cores	10
spark.sql.files.maxparititonBytes	128MB (Default)
spark.sql.files.openCostInBytes	4MB(default)
Spark.default.parallelism	400

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)

bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

bytesperCore = ((54*40)+(54* 4))/400 = 237.6MB

maxSplitBytes = Min(128, 5.94) = 5.94

The number of output partitions are 378.

Example6	
No of file to read	10
File format	Csv(unsplitable compression gzip)
File size	355MB
No of Cores	10
spark.sql.files.maxparititonBytes	128MB (Default)
spark.sql.files.openCostInBytes	4MB(default)
Spark.default.parallelism	10(defaulted to number of cores)

maxSplitBytes = Minimum(maxParititonBytes, bytesperCore)

bytesperCore = (Sum of all data files + No of files * openCostInBytes) / default.parallelism.

bytesperCore = ((355*10)+(10* 4))/10 = 359MB

maxSplitBytes = Min(128, 359) = 128

The number of output partitions are 10. As file is compressed with gzip which is not splitable so for each file one partition will be created.

Partitions at the time of Shuffling

Default = 200 Shuffle partitions(don’t why its 200 and its hardcoded)
In almost every situation 200 will not work. If we are working with shuffle stage of 20GB 200 is fine. I think no one works with spark of 20GB.

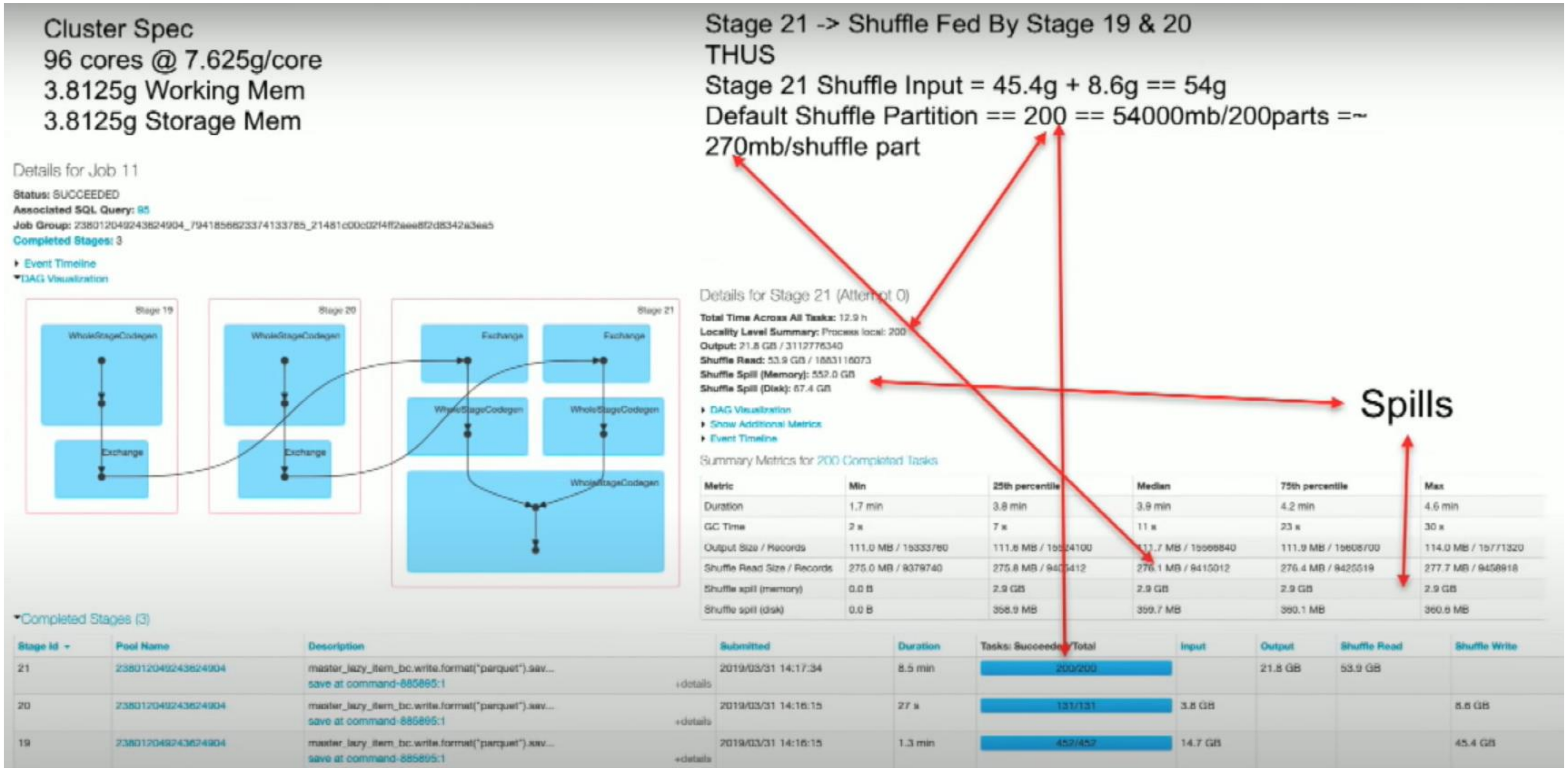
Example:
Shuffle Stage Input: 210G
X = 210000MB/200MB = 1050

Shuffle partitions = cluster core > X then cluster core else X

Shuffle partitions = cluster core = 2000 > 1050 then 1050

For Example in the coming Image you see the shuffle write

Partitions at the time of Shuffling



Cluster Spec
96 cores @ 7.625g/core
3.8125g Working Mem
3.8125g Storage Mem

Stage 21 -> Shuffle Fed By Stage 19 & 20
THUS
Stage 21 Shuffle Input = 45.4g + 8.6g == 54g
Default Shuffle Partition == 200 == 54000mb/200parts =~
270mb/shuffle part

Details for Job 11

Status: SUCCEEDED
Associated SQL Query: 85
Job Group: 2380120492436824904_7941856823374133785_21481c00c02f4ff2aee8f2d8342a3ea5
Completed Stages: 3

▸ Event Timeline
▾ DAG Visualization

Details for Stage 21 (Attempt 0)

Total Time Across All Tasks: 12.9 h
Locality Level Summary: Process local: 200
Output: 21.8 GB / 3112776340
Shuffle Read: 53.9 GB / 1883116073
Shuffle Spill (Memory): 552.0 GB
Shuffle Spill (Disk): 67.4 GB

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 200 Completed Tasks

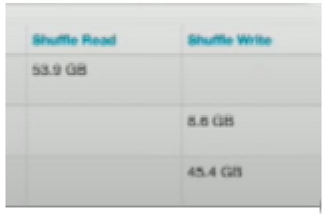
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.7 min	3.8 min	3.9 min	4.2 min	4.6 min
GC Time	2 s	7 s	11 s	23 s	30 s
Output Size / Records	111.0 MB / 15333760	111.8 MB / 15524100	111.7 MB / 15566840	111.9 MB / 15608700	114.0 MB / 15771320
Shuffle Read Size / Records	275.0 MB / 9379740	275.8 MB / 9405412	276.1 MB / 9415012	276.4 MB / 9425519	277.7 MB / 9458918
Shuffle spill (memory)	0.0 B	2.9 GB	2.9 GB	2.9 GB	2.9 GB
Shuffle spill (disk)	0.0 B	358.9 MB	359.7 MB	360.1 MB	360.6 MB

▾ Completed Stages (3)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
21	2380120492436824904	master_lazy_item_bc.write.format("parquet").sav... save at command-885895:1 +details	2019/03/31 14:17:34	8.5 min	200/200		21.8 GB	53.9 GB	
20	2380120492436824904	master_lazy_item_bc.write.format("parquet").sav... save at command-885895:1 +details	2019/03/31 14:16:15	27 s	131/131	3.8 GB			8.6 GB
19	2380120492436824904	master_lazy_item_bc.write.format("parquet").sav... save at command-885895:1 +details	2019/03/31 14:16:15	1.3 min	452/452	14.7 GB			45.4 GB

Partitions at the time of Shuffling

Always the shuffle read should be equal to the shuffle writes. If they are not equally that means the data is not shuffled correctly.

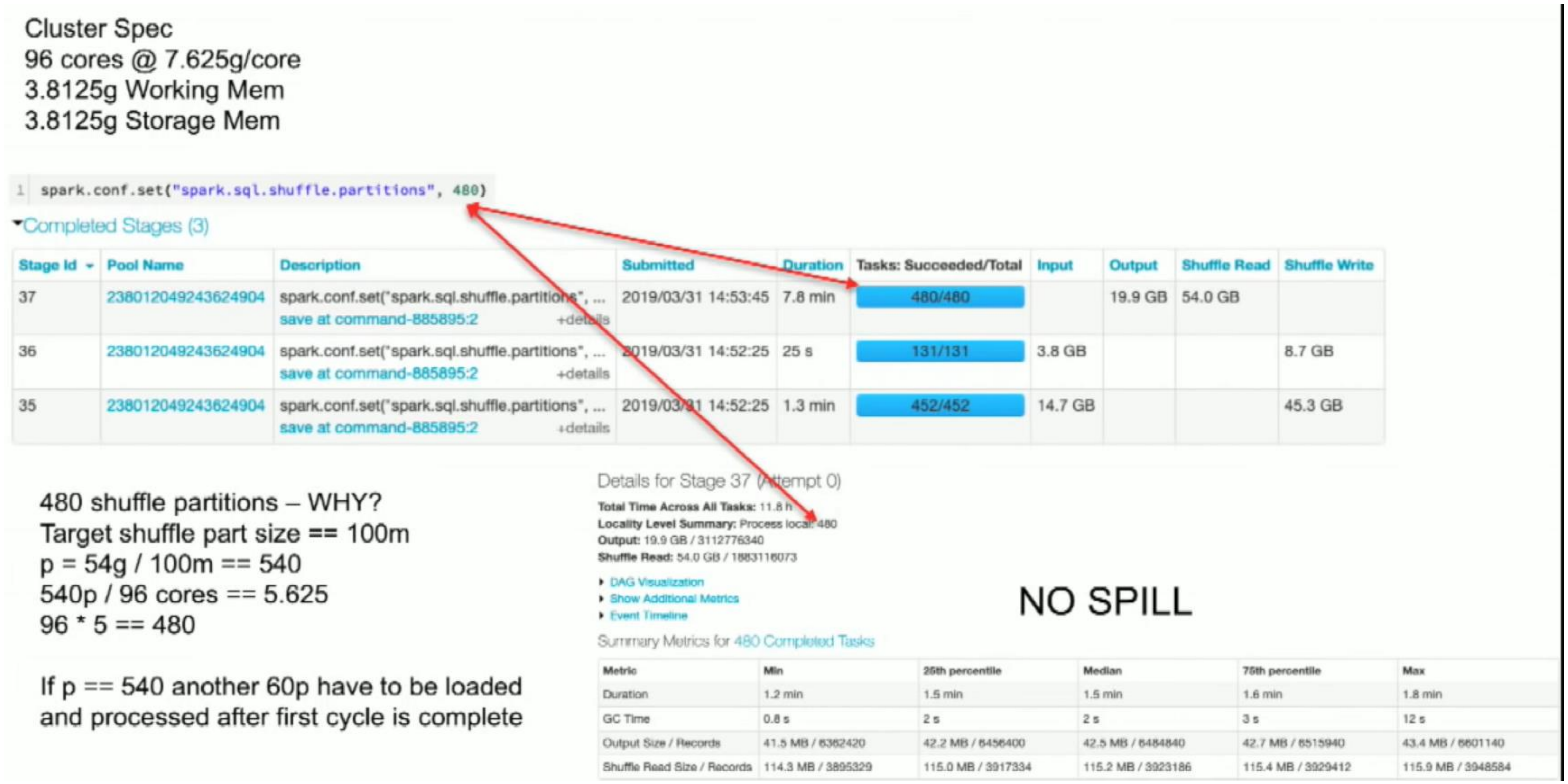


Shuffle Read	Shuffle Write
53.9 GB	
	8.8 GB
	45.4 GB

Spill are slowest and they occur if you don't provide them enough shuffle partitions at the time of shuffling.

If you clearly observe there is lots shuffle spill(memory):552.9Gb and shuffle spill(disk):57.4Gb is observed for 53.9Gb shuffle data. Even though that shuffle data is 53.9Gb the spill is 552.9Gb which is 10 times more than actual data this is because of the data is decompressed in deserialized to put on to disk and into memory that it gets to this large size. On a job that has 35Tb of shuffle Its not uncommon to see a 90 Tb of spill it might complete if you have enough disk.

Partitions at the time of Shuffling



Cluster Spec
96 cores @ 7.625g/core
3.8125g Working Mem
3.8125g Storage Mem

```
spark.conf.set("spark.sql.shuffle.partitions", 480)
```

Completed Stages (3)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
37	238012049243624904	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885895:2 +details	2019/03/31 14:53:45	7.8 min	480/480		19.9 GB	54.0 GB	
36	238012049243624904	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885895:2 +details	2019/03/31 14:52:25	25 s	131/131	3.8 GB			8.7 GB
35	238012049243624904	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885895:2 +details	2019/03/31 14:52:25	1.3 min	452/452	14.7 GB			45.3 GB

480 shuffle partitions – WHY?
Target shuffle part size == 100m
p = 54g / 100m == 540
540p / 96 cores == 5.625
96 * 5 == 480

If p == 540 another 60p have to be loaded and processed after first cycle is complete

Details for Stage 37 (Attempt 0)

Total Time Across All Tasks: 11.8 h
Locality Level Summary: Process local: 480
Output: 19.9 GB / 3112776340
Shuffle Read: 54.0 GB / 1883116073

- DAG Visualization
- Show Additional Metrics
- Event Timeline

NO SPILL

Summary Metrics for 480 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.2 min	1.5 min	1.5 min	1.6 min	1.8 min
GC Time	0.8 s	2 s	2 s	3 s	12 s
Output Size / Records	41.5 MB / 6382420	42.2 MB / 6456400	42.5 MB / 6484840	42.7 MB / 6515940	43.4 MB / 6601140
Shuffle Read Size / Records	114.3 MB / 3895329	115.0 MB / 3917334	115.2 MB / 3923186	115.4 MB / 3929412	115.9 MB / 3948584

Partitions at the time of Shuffling

My Targeted shuffle part size = 100MB (Means I want all my shuffle partitions not more than 100MB)
Total cores available are 96

Partitions = 54g/100Mb= 540

540 / 96 = 5.625 partitions per core we can round off to 5

96 * 5 = 480 shuffle partitions.

With 480 shuffle partitions we can see there is no spill to the disk.

Prudhvi Akella@2021