Apache Spark Foundation Course - Spark Architecture Part-1

In this session, I will talk about Apache Spark Architecture. We will try to understand various moving parts of Apache Spark, and by the end of this video,

you will have a clear understanding of many Spark related jargons and the anatomy of Spark Application execution.

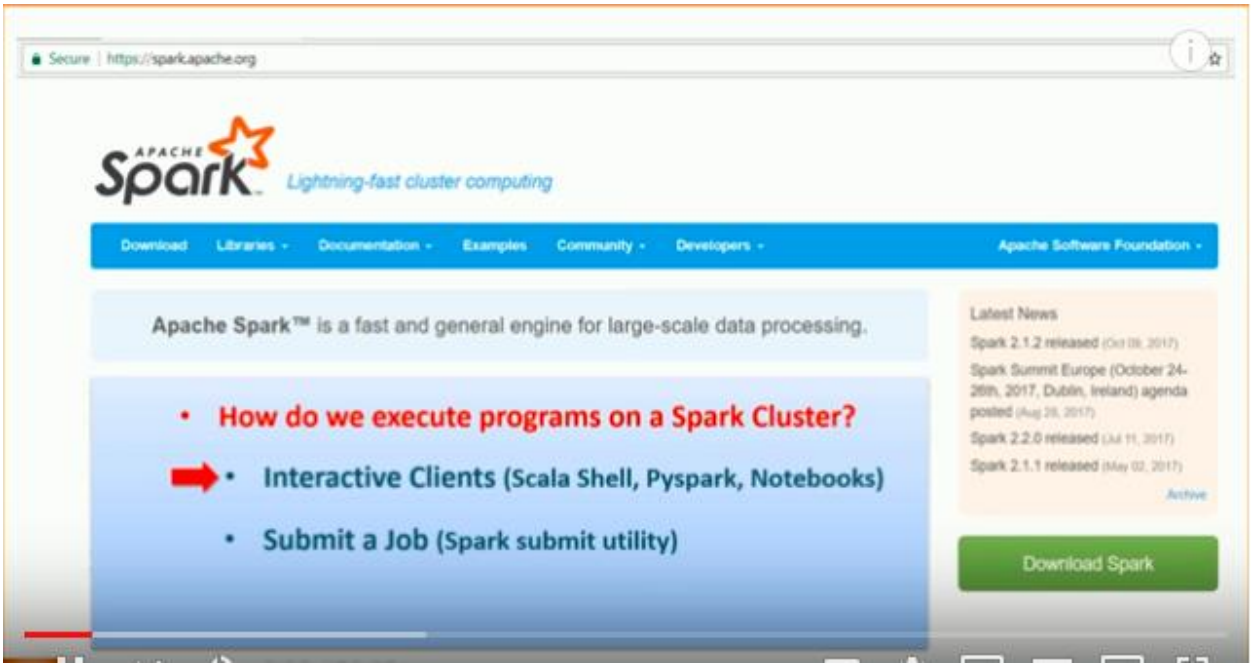
We already had an introduction to Apache Spark. I guess you learned enough to answer these questions.

1. What is Apache Spark?

- A distributed computing platform.

2. What do we do with Apache Spark?

- We create programs and execute them on a Spark Cluster.



How do we execute Spark Programs?

There are two methods.

1. Interactive clients (Scala Shell, Pyspark, Notebooks)

2. Submit a job (Spark submit utility)

I already showed you Spark Installation. We accessed Spark using Scala Shell, Pyspark Shell, and Jupyter notebooks. All of those are interactive clients.

That's the first method for executing your code on a Spark cluster. Most of the people use interactive clients during the learning or development process.

Interactive clients are best suitable for exploration purpose.

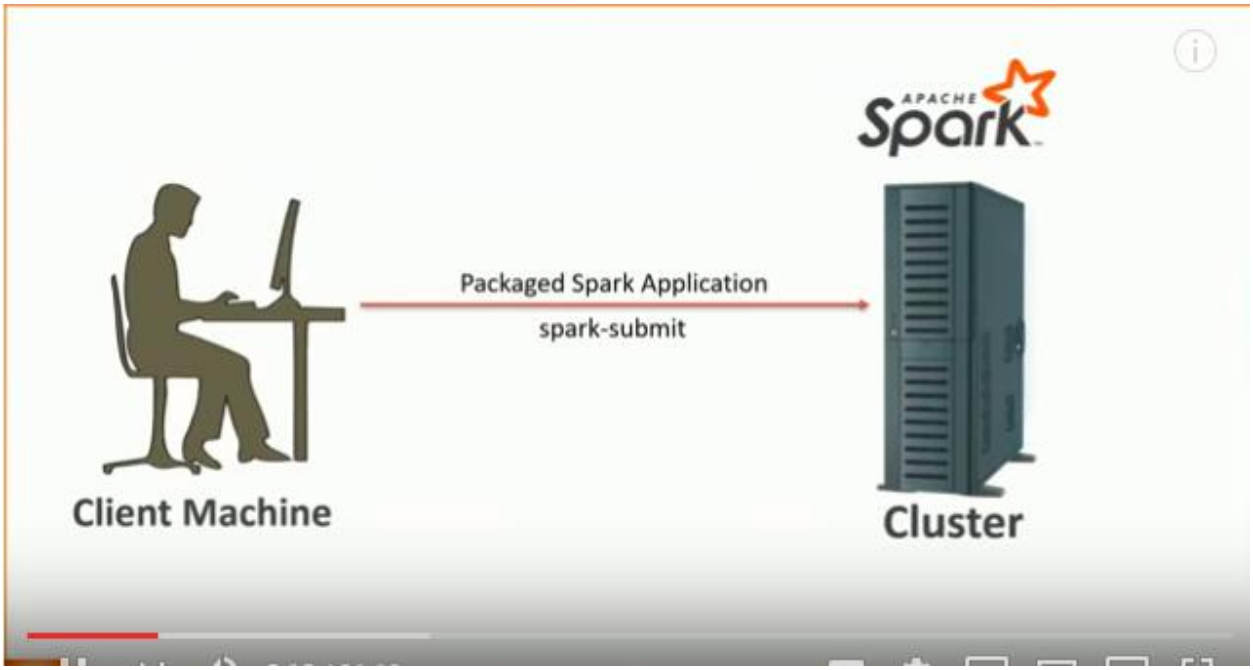
But ultimately, all your exploration will end up into a full-fledged Spark application.

It may be a streaming application. For example, reading a news feed as it arrives and applying a machine learning algorithm to figure out that what type of

users might be interested in this news.

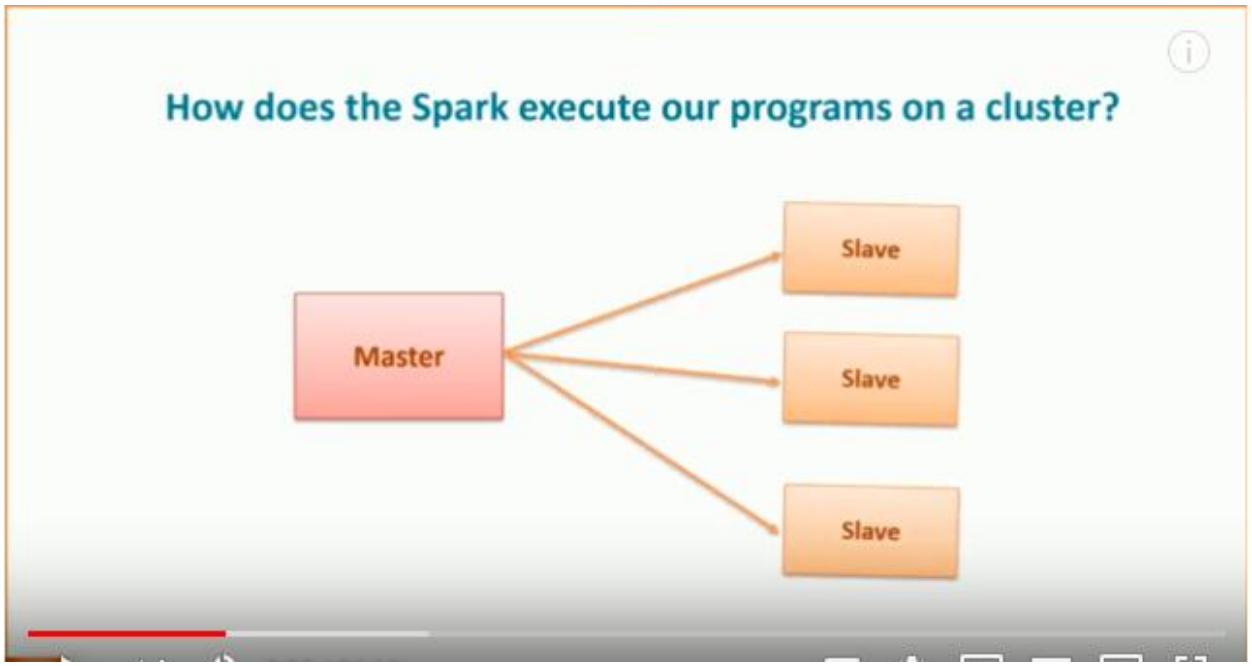
It may be a batch job. For example, my YouTube statistics. Every twenty-four hours a batch job reads all the data collected during the period and computes

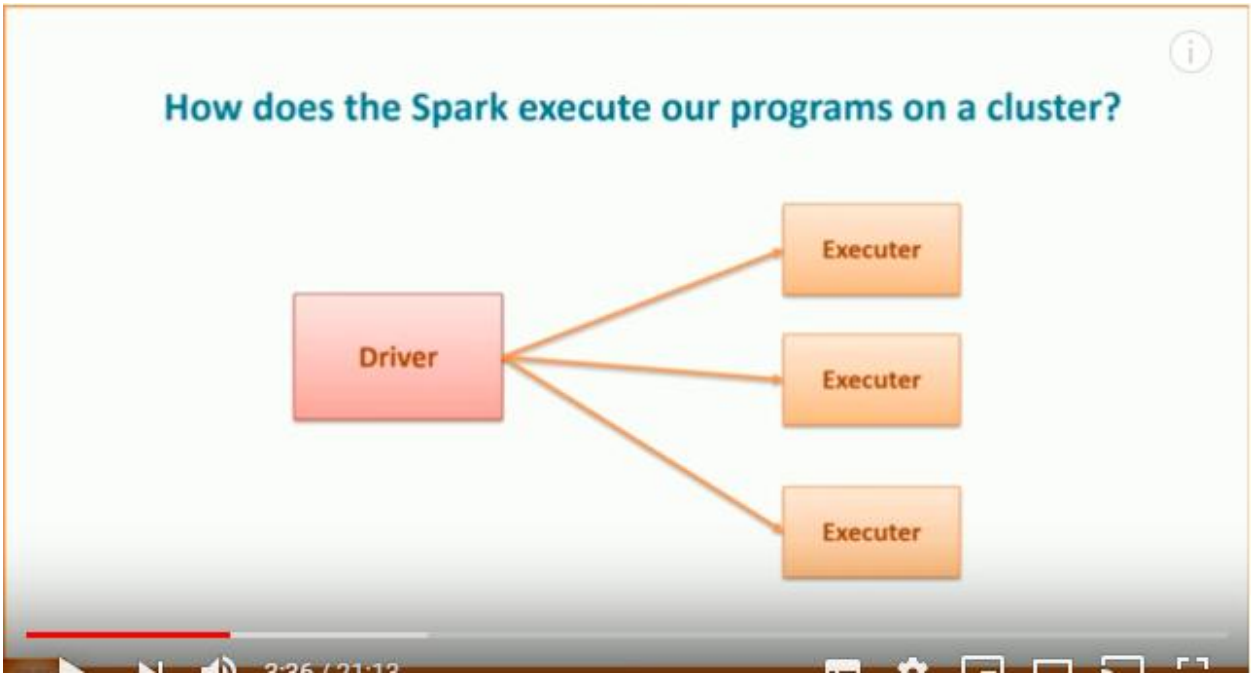
the watch time minutes for that period. Finally, it inserts one record in some database, and I see it on my dashboard.



In both the cases (A long-running streaming job or a periodic batch job), you must package your application and submit it to Spark cluster for execution.

That's the second method for executing your programs on a Spark cluster. For a production use case, you will be using this technique.





How Spark executes a program?

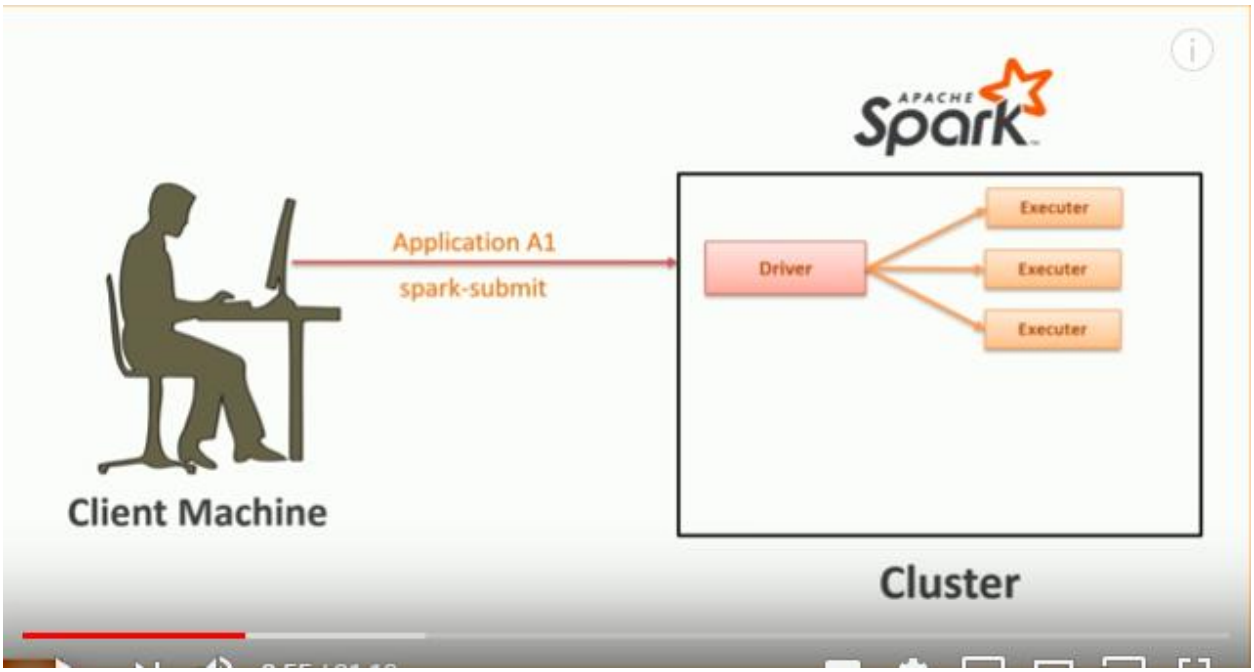
Spark is a distributed processing engine, and it follows the master-slave architecture. So, for every Spark App, it will create one master process and multiple

slave processes. In Spark terminology, the master is the driver, and the slaves are the executors.

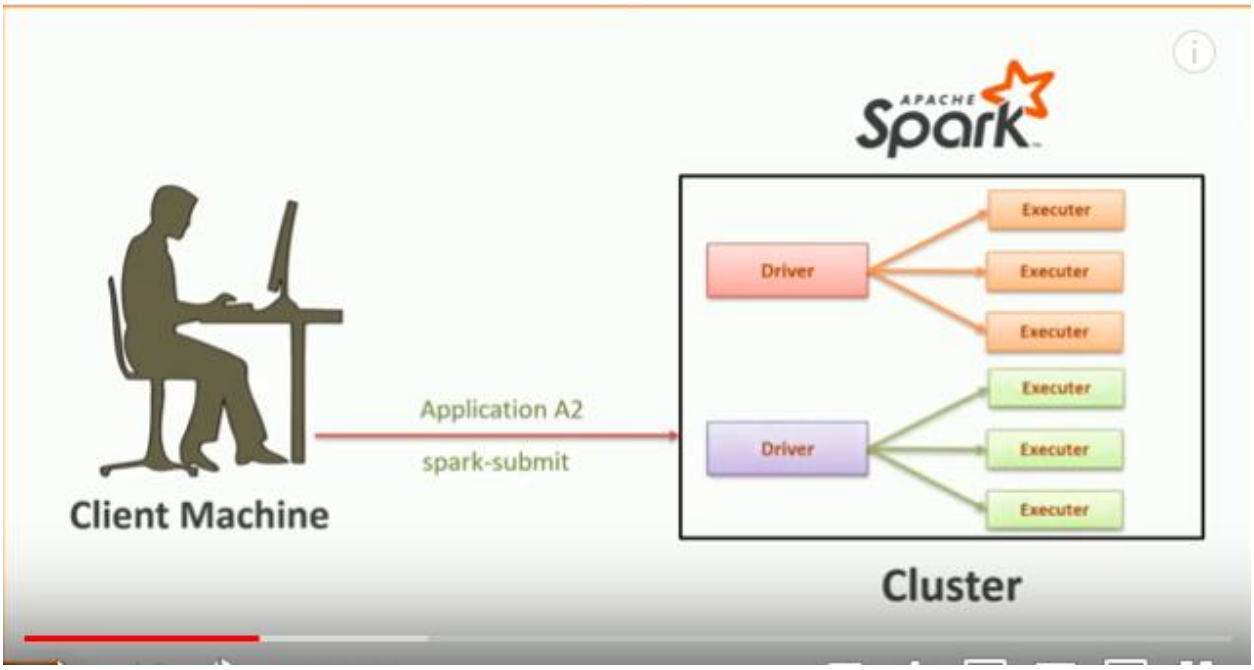
Let's try to understand it with a simple example.

Suppose you are using the Spark Submit utility. You execute an application 'A1' using Spark Submit, and Spark will create one driver process and some

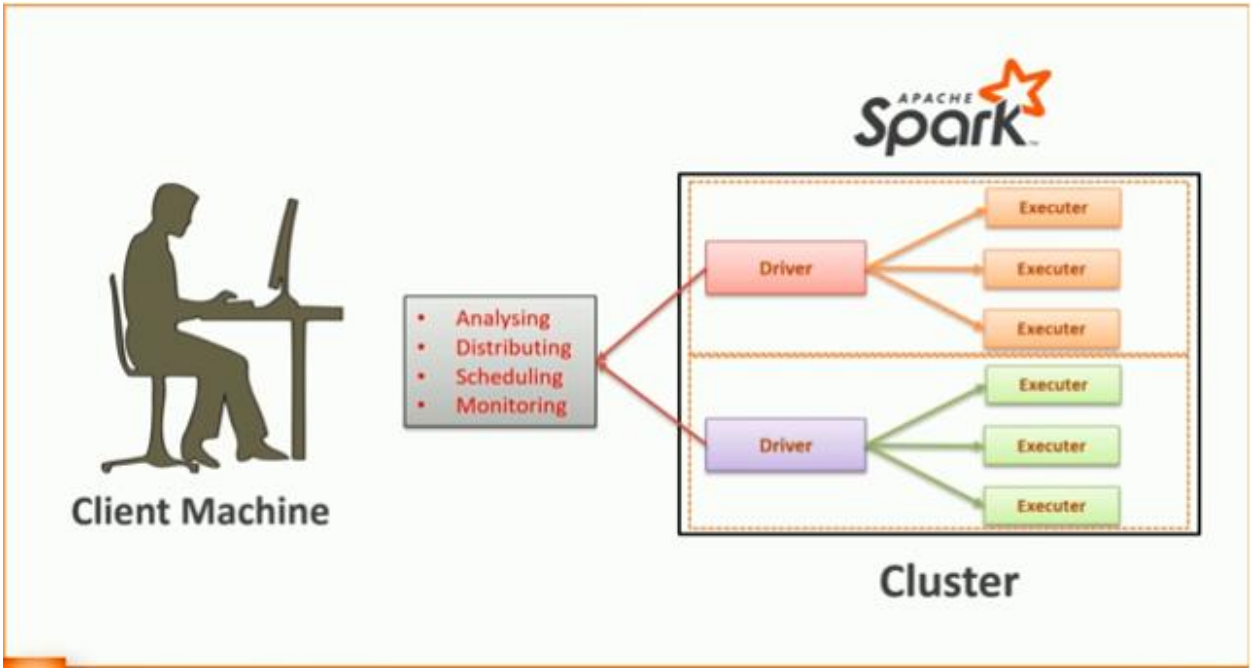
executor processes for A1. The entire set of driver and executers is exclusive for the application A1.



Now, you submit another application A2, and Spark will create one more driver process and some executor process for A2.



So, for every application, Spark creates one driver and a bunch of executors. Since the driver is the master, it is responsible for analyzing, distributing, scheduling and monitoring work across the executors. The driver is also responsible for maintaining all the necessary information during the lifetime of the application.



Now the executors, they are only responsible for executing the code assigned to them by the driver and reporting the status back to the driver.

Great. Now we know that every Spark application has a set of executors and one dedicated driver.

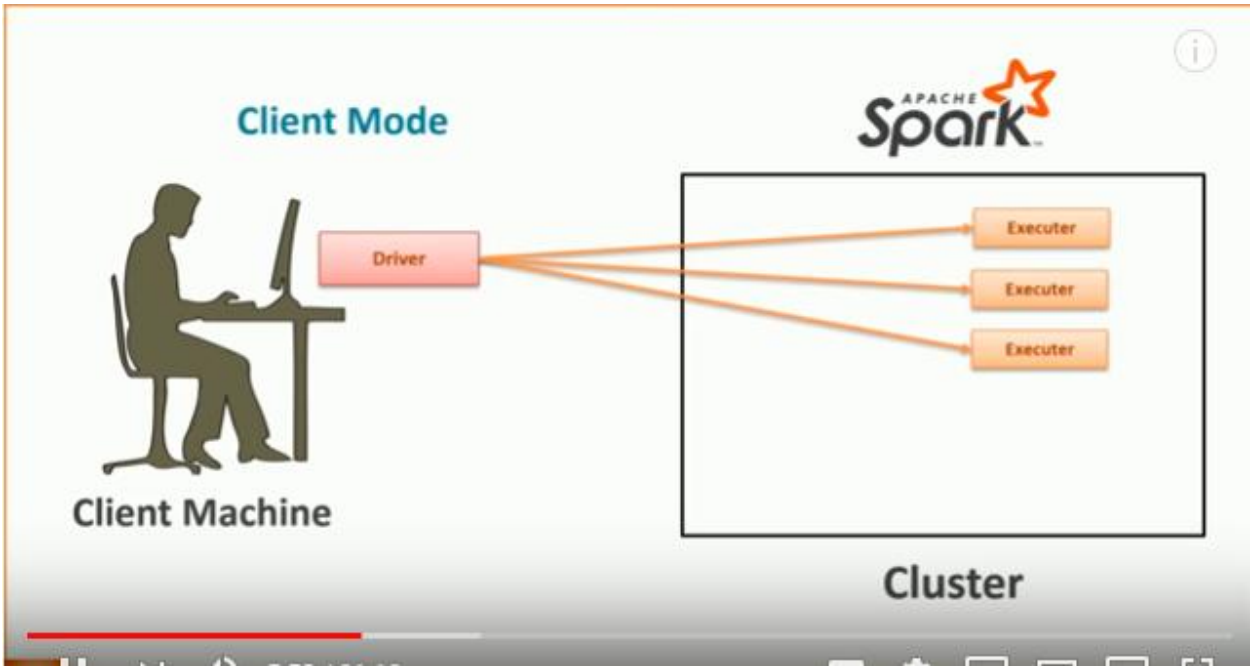
Who executes where?

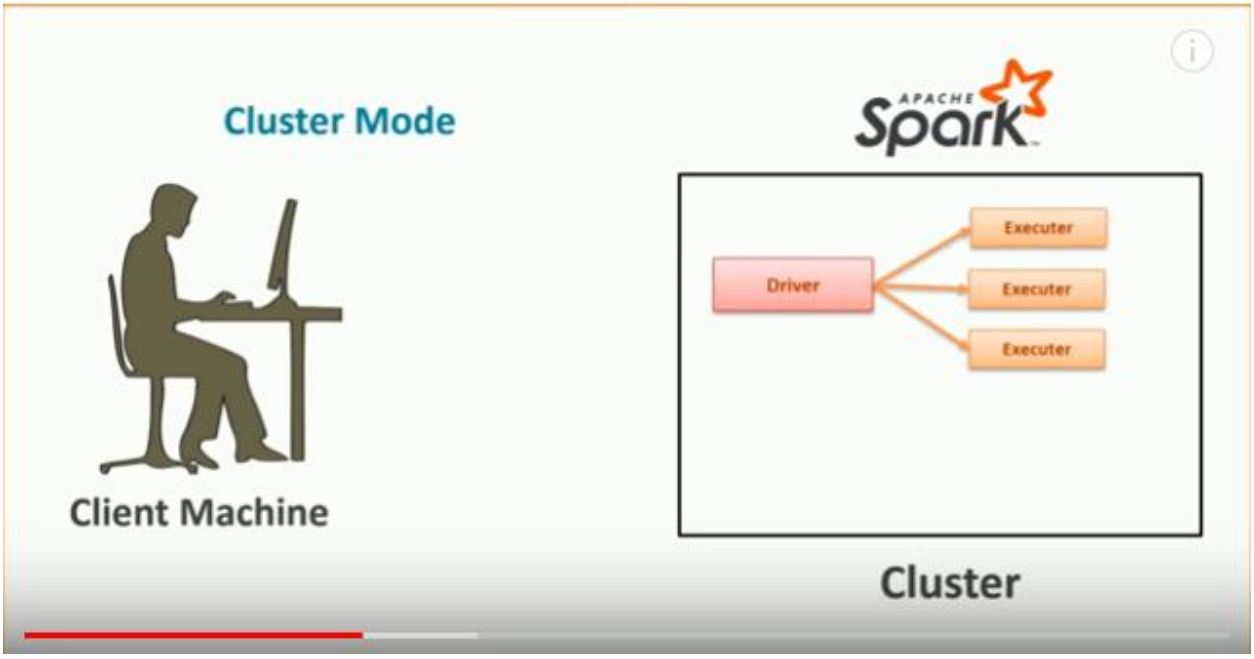
We have a cluster, and we also have a local client machine. We start the Spark application from our client machine. So, the question is what goes where?

The executors are always going to run on the cluster machines. There is no exception to this.

But you have the flexibility to start the driver on your local machine or on the cluster itself. You might be wondering that why do we have this flexibility?

I will explain that in a minute but let me formalize this idea.





When you start an application, you have a choice to specify the execution mode, and there are two options.

1. Client Mode - Start the driver on your local machine

2. Cluster Mode - Start the driver on the cluster.

The Client Mode will start the driver on your local machine, and the Cluster Mode will start the driver on the cluster.

You already know that the driver is responsible for the whole application. If anything goes wrong with the driver, your application state is gone. So, if you

start the driver on your local machine, your application is directly dependent on your local computer. You don't want that dependency in a production

application. After all, you have a dedicated cluster to run the job. Right?

Hence, the Cluster mode makes perfect sense for production deployment. Because after spark-submit, you can switch off your local computer and the

application executes independently within the cluster.

On the other side, when you are exploring things or debugging an application, you want the driver to be running locally. If the driver is running locally, you

can easily debug it, or at least it can throw back the output on your terminal. Right?

That's where the client-mode makes more sense over the cluster-mode. And hence, when you start a Spark shell or any other interactive client. You would

be using a client mode.

So, if you are running a Spark shell, your driver is running locally within the same JVM process. You won't find a separate driver process. It's only the Spark

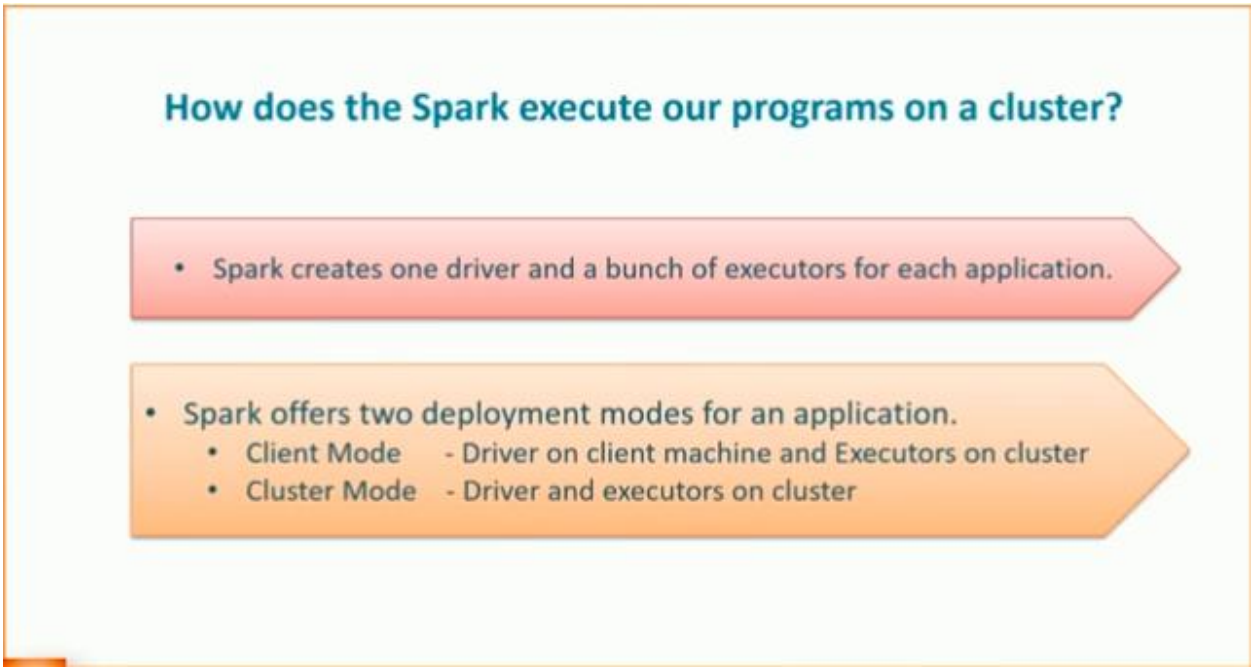
shell, and the driver is embedded within the shell.

Great, we have an answer to the following question.

How does the Spark execute our programs on a cluster?

You learned the answer. Spark will create one driver and a bunch of executors. If you are using an interactive client, your client tool itself is a driver, and you

will have some executors on the cluster. If you are using spark-submit in cluster mode, Spark will start your driver and executors on the Cluster.



Who controls the cluster?

How Spark gets the resources for the driver and the executors?

That's where we need a cluster manager.

As on the date, Apache Spark supports four different cluster managers.

1. Apache YARN

2. Apache Mesos

3. Kubernetes

4. Standalone

YARN is the cluster manager for Hadoop. As of date, YARN is the most widely used cluster manager for Apache Spark.

Apache Mesos is another general-purpose cluster manager. If you are not using Hadoop, you might be using Mesos for your Spark cluster.

I won't call the Kubernetes a cluster manager. In fact, it's a general-purpose container orchestration platform from Google. Spark on Kubernates is not yet

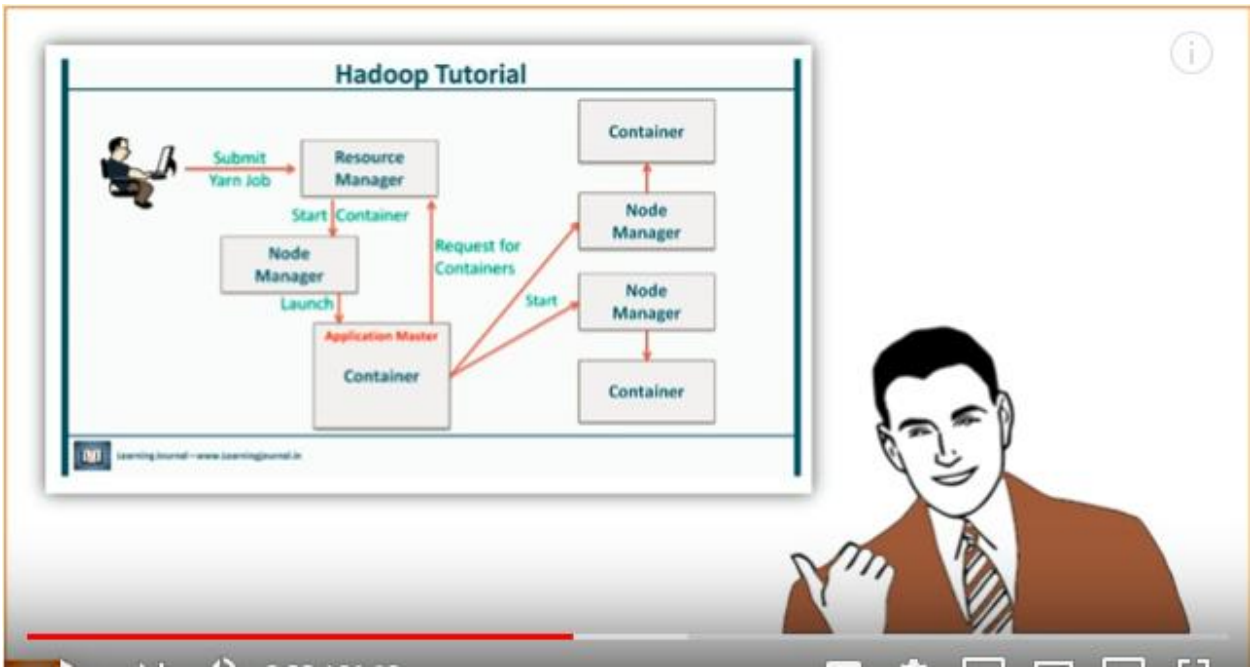
production ready but the community is working on it.

Finally, the standalone. This one is a simple and basic cluster manager that comes with Spark and makes it easy to set up a Spark cluster very quickly. I

don't think you would be using it in a production environment.

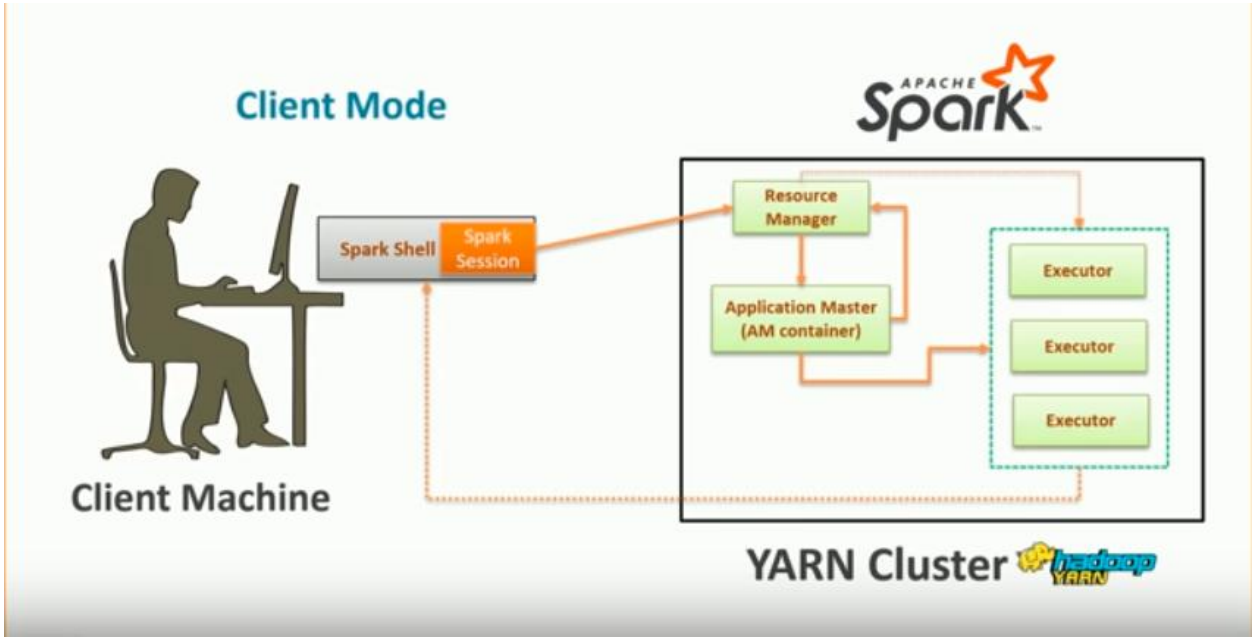
No matter which cluster manager do we use, primarily, all of them delivers the same purpose.

You might be interested in step by step process of resource allocation. You can watch the video, It explains the resource allocation process in the YARN cluster.

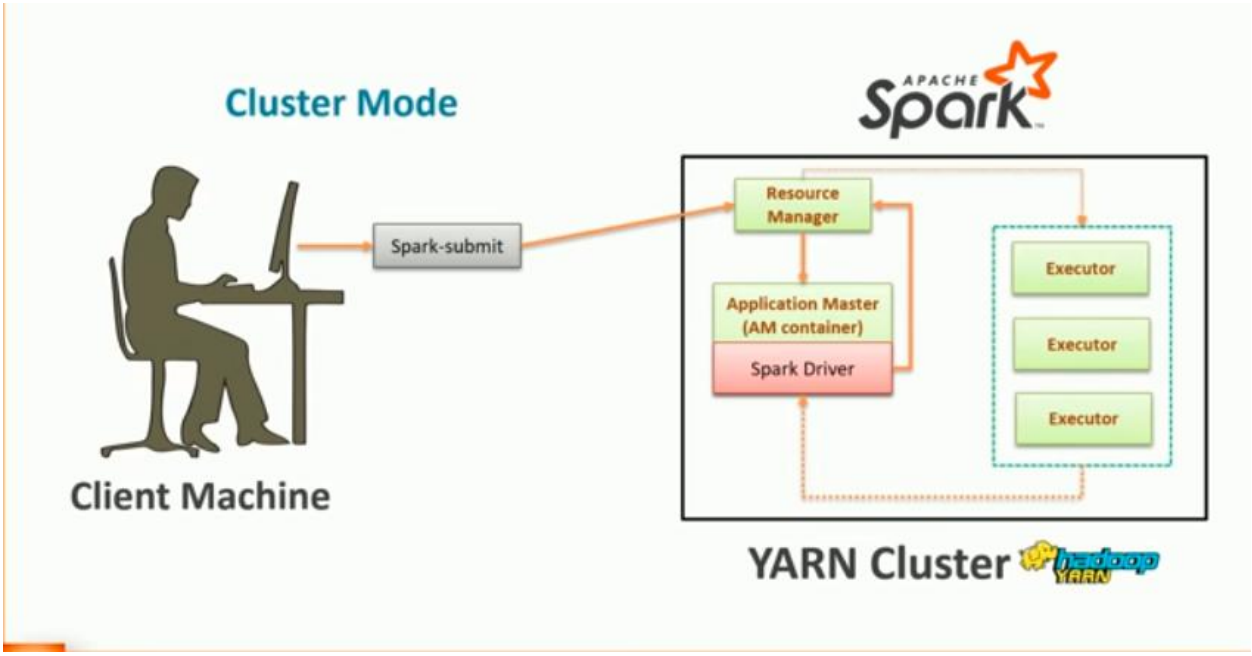


Great. We covered How Spark runs on a cluster. We learned that we have two options.

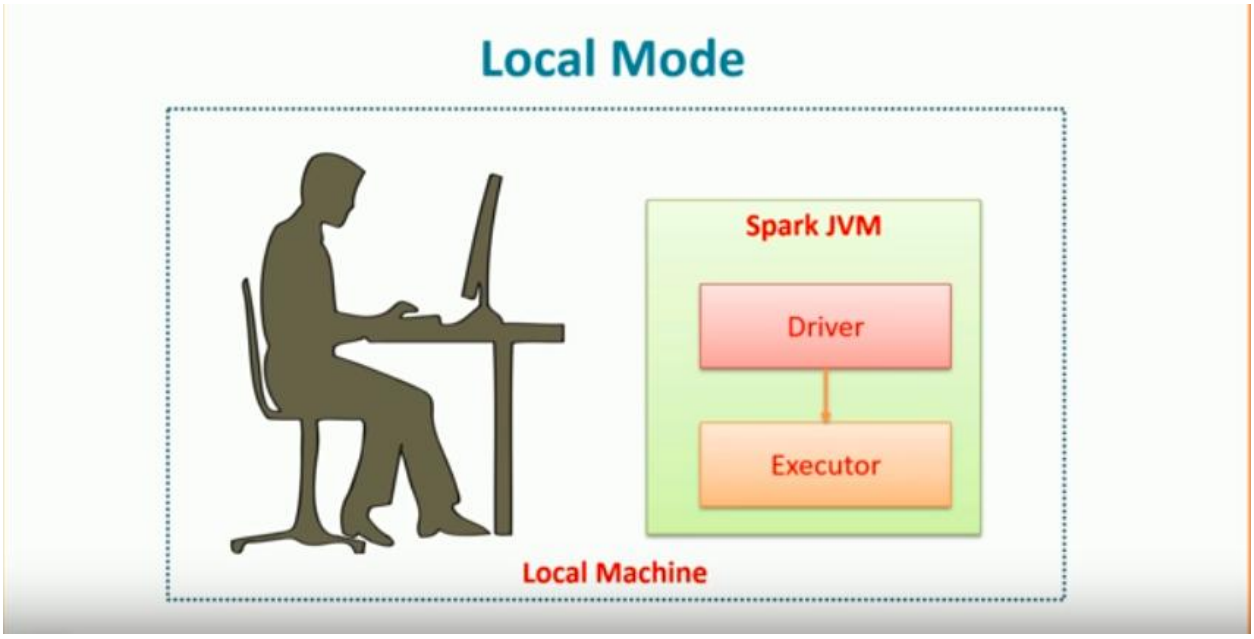
1. Client Mode



2. Cluster Mode



There is a third option as well. The Local Mode. You can also start Spark application in the Local Mode.



When you don't have enough infrastructure to create a multi-node cluster, and you still want to setup Apache Spark. You might want to di it just for learning purpose. You can use local mode.

The local mode is the most comfortable method to start a Spark Application. In this Mode, you don't need any cluster. Neither YARN nor Mesos, nothing.

You just need a local machine and the Spark binaries. You start a Spark Application in local mode. It begins in a JVM, and everything else including driver

and executor runs in the same JVM. I showed you a Spark installation in the first video . That demo uses the local mode installation. The local mode is the

most convenient method for setting up a quick learning environment.

Great, I talked about five things.

1. Driver

2. Executors

3. Client mode

4. Cluster mode

5. Local mode.

```
    spark-submit --class org.apache.spark.examples.SparkPi spark-home/spark-2.2.0-bin-hadoop2.6/examples/jars/spark-examples_2.11-
2.2.0.jar 1000
    //Start an SSh tunnel
    gcloud compute ssh --zone=us-east1-c --ssh-flag="-D" --ssh-flag="10000" --ssh-flag="-N" "spark22-notebook-m"
    //Start the chrome browser using the SSH tunnel
    cd C:\Program Files (x86)\Google\Chrome\Application
    chrome.exe  "http://spark4-m:4040" --proxy-server="socks5://localhost:10000" --host-resolver-rules="MAP * 0.0.0.0 , EXCLUDE
localhost" --user-data-dir=/tmp/spark22-notebook
    //Start a Spark sell with three executors
    spark-shell --num-executors 3
    //Submit a Spark Job in cluster mode
    spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode cluster file:///usr/lib/spark/examples/jars/spark-examples.jar
1000
```

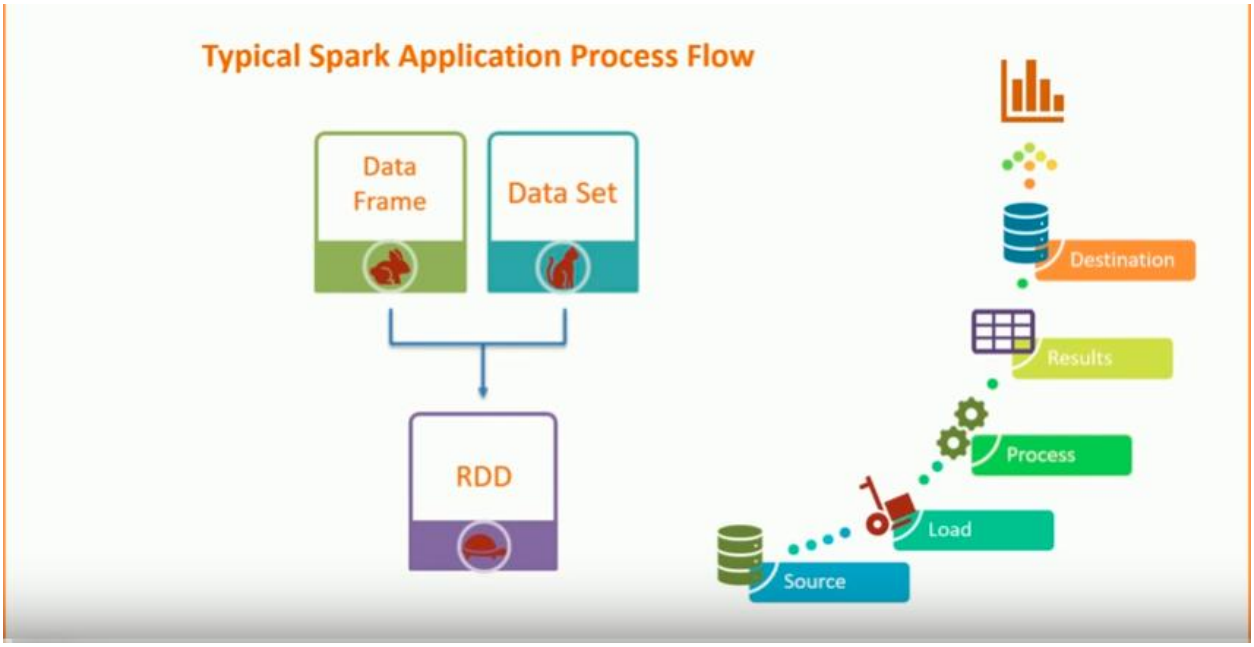
Great. We learned and explored many things. But this doesn't end the Spark Architecture discussion. I need at least one more video to deep dive further into

Apache Spark architecture and explore following.

How does the Spark break our application into smaller parts and get it done at the executors?

Apache Spark Foundation Course - Spark Architecture Part-2

In the previous session, we learned about the application driver and the executors. We know that Apache Spark breaks our application into many smaller tasks and assign them to executors. So, Spark executes the application in parallel. But do you understand the internal mechanics? That's the topic of this video. We will try to deep dive inside the Spark and investigate the internal mechanics of parallel processing. The objective of this video is to develop a solid understanding of Spark's execution model and Architecture.



All that you are going to do in Apache Spark is to read some data from a source and load it into Spark. Process the data and hold the intermediate results, and finally write the results back to a destination.
But in this process, you need a data structure to hold the data in Spark. We have three alternatives to hold data in Spark.

1. Data Frame.
2. DataSet.
3. RDD.

We will learn about all of these. In fact, Spark 2.x recommends using the first two and avoid using RDDs. So, the primary focus of this training will be on first two items. But there is a critical fact to note about RDDs. Data Frames and Datasets, both are ultimately compiled down to an RDD. So, under the hood, everything in Spark is an RDD. And for that reason, I will start with RDDs and try to explain the mechanics of parallel processing.

What is Spark RDD?

Let's define the RDD. The name stands for Resilient Distributed Dataset. However, I can describe it like this.

Spark RDD is a resilient, partitioned, distributed and immutable collection of data.

Let's quickly review this description.

1. Collection of data - This one is the most basic thing. They hold data and appears to be a Scala Collection.
2. Resilient - That means, they can recover from a failure. RDDs are fault tolerant.
3. Partitioned - Spark breaks the RDD into smaller chunks of data. These pieces are called partitions.
4. Distributed - Instead of keeping these Partitions on a single machine, Spark spreads them across the cluster. So, they are a distributed collection of data.
5. Immutable - Once defined, you can't change them. So, Spark RDD is a read-only data structure.

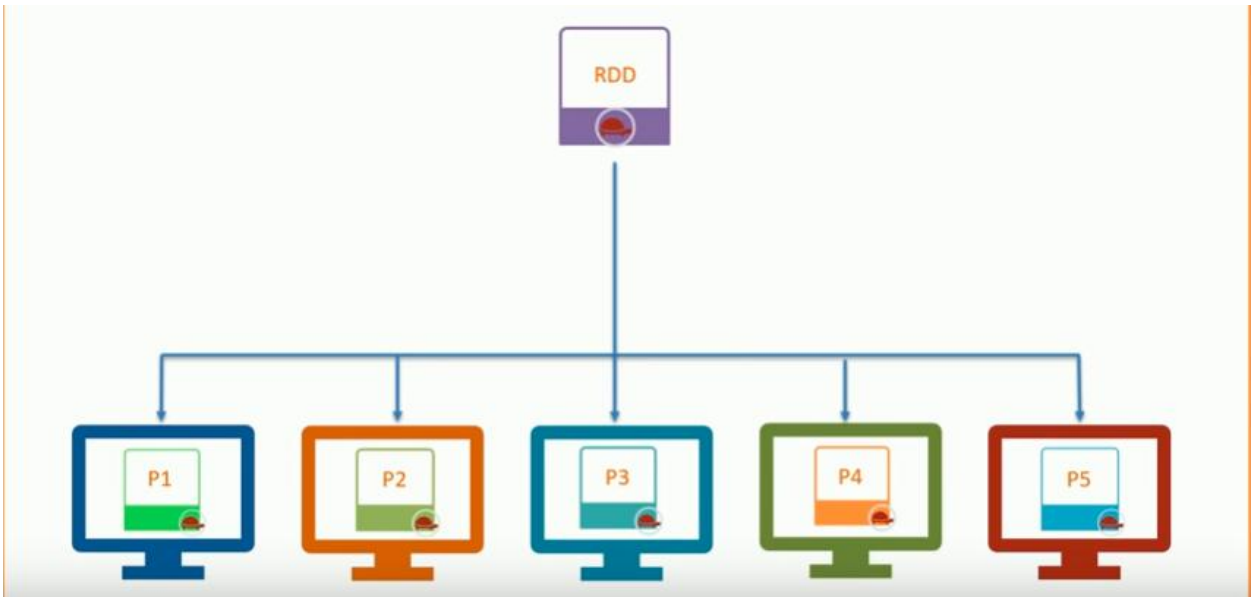
You can create an RDD using two methods.

1. Load some data from a source.
2. Create an RDD by transforming another RDD.

Let me show you an example. We will load some data from a file to create an RDD, and then I will show you the number of partitions.

```
//Shell command
find \ -name * > flist.txt
//Start Spark Shell
spark-shell
// Load the data file
val flistRDD = sc.textFile("flist.txt")
//Check the number of partitions
flistRDD.getNumPartitions
// Redefine the partitions
val flistRDD = sc.textFile("flist.txt", 5)
// The second parameter in the above API is the number of partitions.
// Verify the new partitons
flistRDD.getNumPartitions
// You can iterate to all partitions and count the number of elements in each partition.
flistRDD.foreachPartition(p =>println("No of Items in partition-" + p.count(y=>true)) )
```

Great, now we know that an RDD is a Partitioned Collection, and we can control the number of partitions. However, that wasn't the objective of this video. At the beginning of this video, I said that I am going to answer the below question.



How does the Spark break our code into a set of tasks and run it in parallel?

Let me ask you a simple question.
Given the above RDD, If I want to count the number of lines in that RDD, Can we do it in parallel?
No brainer. Right? We already have five partitions. I will give one partition to each executor and ask them to count the lines in the given partition. Then I will take the counts back from those executors and sum it. Simple. Isn't it? That's what the Spark does.
Let's execute the code to count the number of lines in this RDD.

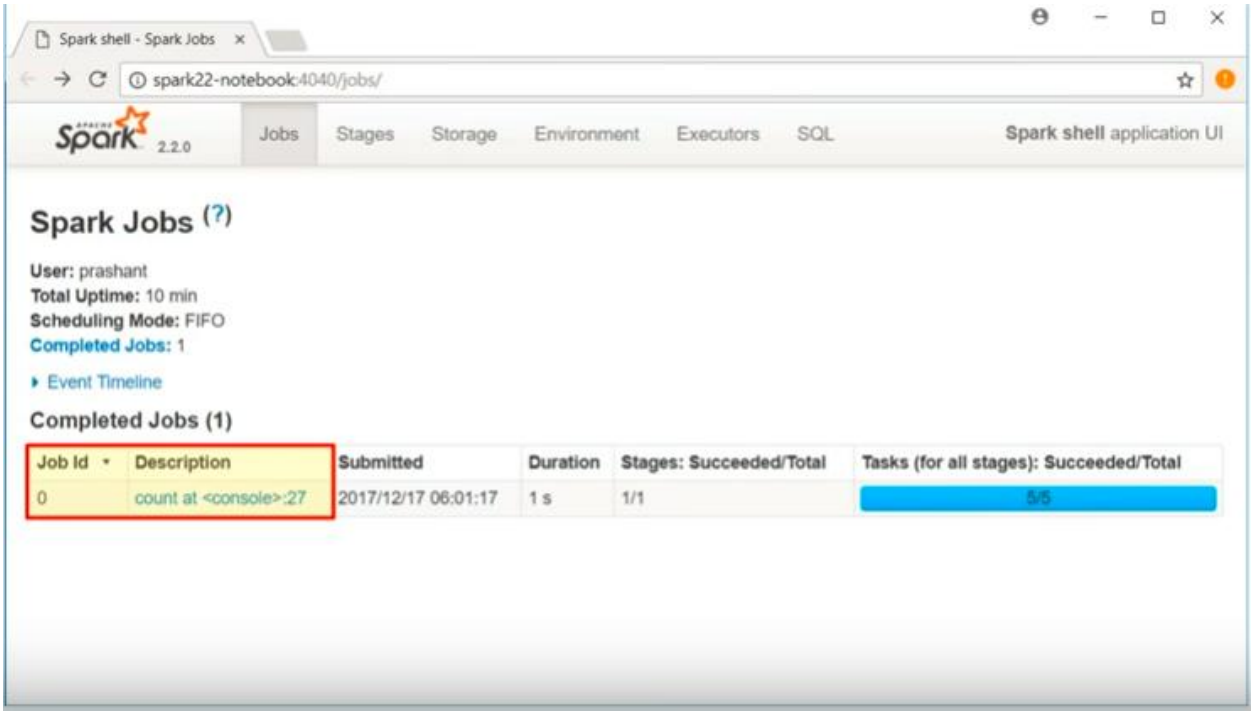
```
val flistRDD = sc.textFile("flist.txt", 5)
flistRDD.count()
```

Do you want to see what happens under the hood?
Watch the video. The video shows and explains following things.

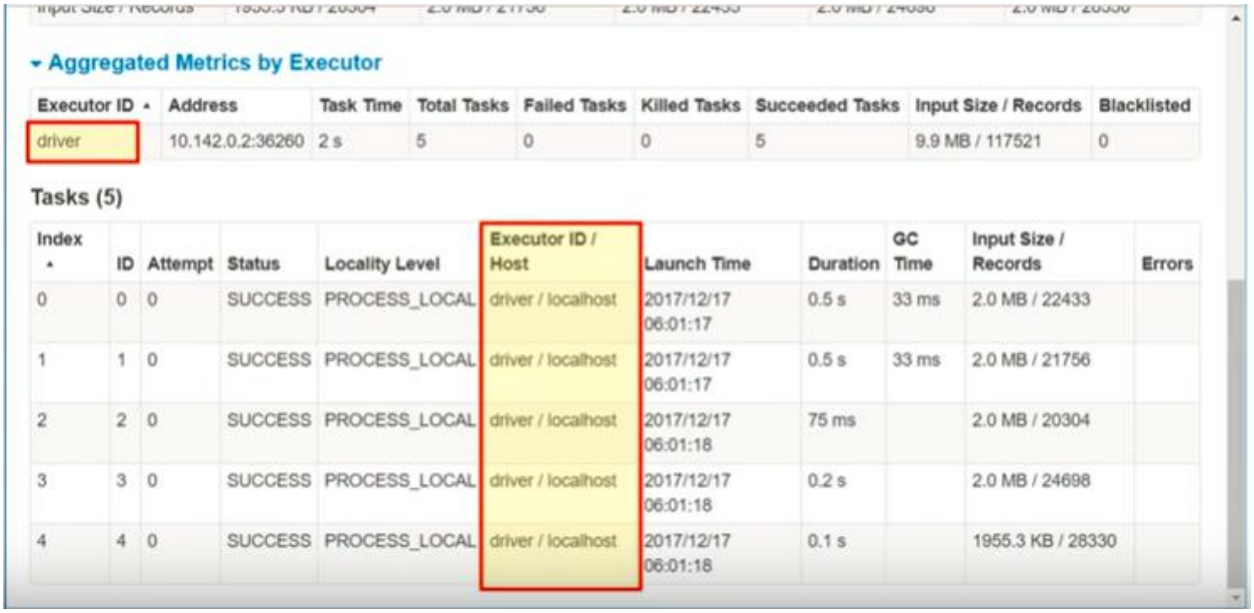
1. Spark Job
2. Job Stages
3. Spark Tasks

However, let me try to summarize it.
We loaded the file and asked for the count, and hence Spark started one Job. The job is to calculate the count.
Spark breaks that job into five tasks because we had five partitions. And it starts one counting task per partition. A task is the smallest unit of work, and it is performed by an executor.
We talk about stages in the next example.
For this example, I am executing it in local mode, and I have a single executor. Hence all these

tasks are executed by the same executor.



You can try the same example on a real multi-node cluster and see the difference.
Great. I think by the end of the video, you will have a fair idea about the parallel processing in Apache Spark.



There are two main variables to control the degree of parallelism in Apache Spark.

1. The number of partitions.
2. The number of executors.

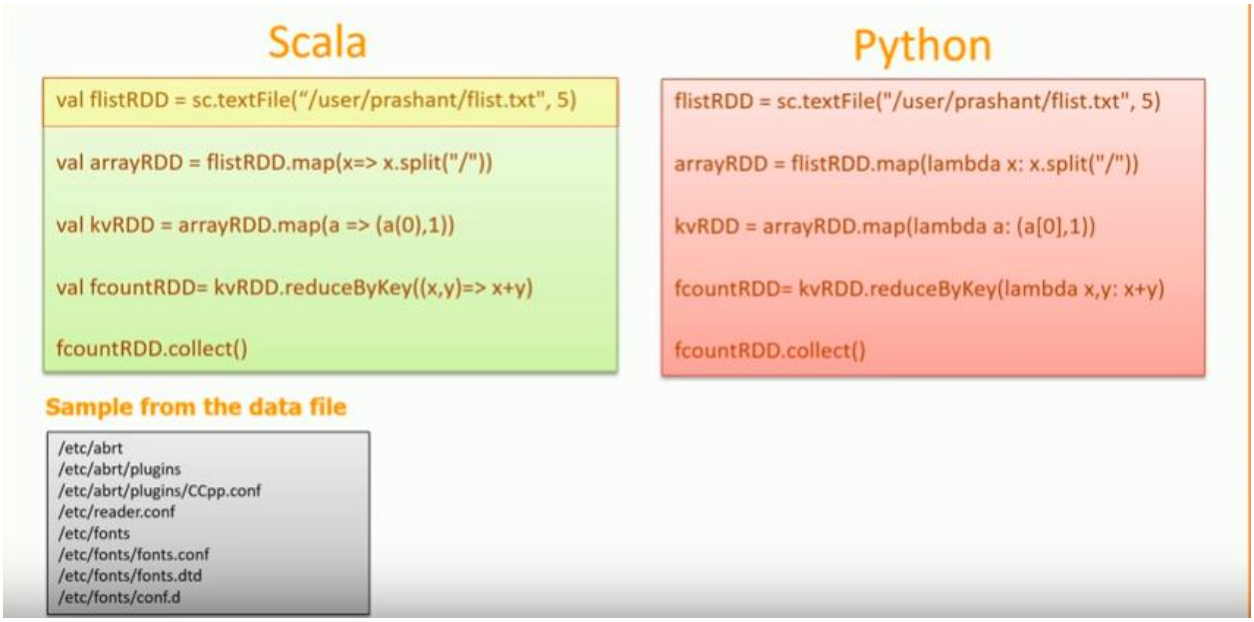
If you have ten partitions, you can achieve ten parallel processes at the most. However, if you have just two executors, all those ten partitions will be queued to those two executors.
So far so good. Let's do something little more complicated and take our understanding to the next level.

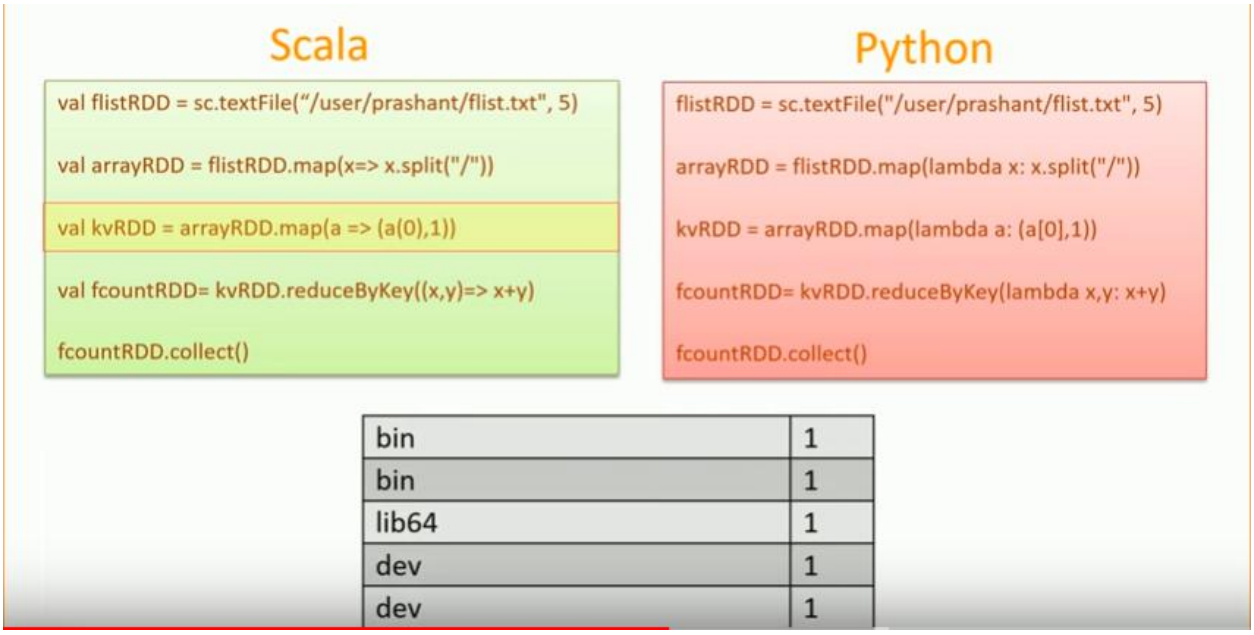
Spark RDD Example

Here is an example in Scala as well as in Python. Let me quickly explain the code.

```
//-------------------Scala Code----------------------
val flistRDD = sc.textFile("/home/prashant/flist.txt", 5)
val arrayRDD = flistRDD.map(x=>x.split("/"))
val kvRDD = arrayRDD.map(a => (a(1),1))
val fcountRDD= kvRDD.reduceByKey((x,y)=>x+y)
fcountRDD.collect()

//-------------------Python Code------------------
flistRDD = sc.textFile("/home/prashant/flist.txt", 5)
arrayRDD = flistRDD.map(lambda x: x.split("/"))
kvRDD = arrayRDD.map(lambda a: (a[1],1))
fcountRDD= kvRDD.reduceByKey(lambda x,y: x+y)
fcountRDD.collect()
```





Line one loads a text file into an RDD. The file is quite small. If you keep it in HDFS, it may have one or two blocks in HDFS, So, it is likely that you get one or two partitions by default. However, we want to make five partitions of this data file.
The line two executes a map method on the first RDD and returns a new RDD. We already know that RDDs are immutable. So, we can't modify the first RDD. Instead, we take the first RDD, perform a map operation and create a new RDD. The map operation splits each line into an array of words. Hence, the new RDD is a collection of Arrays. The third line executes another Map method over the arrayRDD. This time, we generate a tuple. A key value pair. I am taking the second element of the array as my key. And the value is a hardcoded numeric one.
What am I trying to do?
Well, I am trying to count the number of files in each different directory. That's why I am taking the directory name as a key and one as a value. Once I have the kvRDD, I can easily count the number of files. All I have to do is to group all the values by the key and sum up the 1s. That's what the fourth line is doing. The ReduceByKey means, group by key and sum those 1s.
Finally, I collect all the data back from the executors to the driver.
That's it.
I am assuming that you already know the mechanics of the map and reduce methods. I have covered all these fundamentals in my Scala tutorials. I have also included a lot of content about functional programming. If you are not familiar with those fundamentals, I strongly recommend that you first go through my Scala training to take full advantage of the Spark tutorials.
Let's execute it and see what's happening behind the scenes.
This time I want to use a mult-inode cluster.
Please watch the video. The video shows following things.

1. Start a six-node Spark cluster.
2. Create the data file on the master node.
3. Copy the data file to HDFS.
4. Start a Spark Shell.

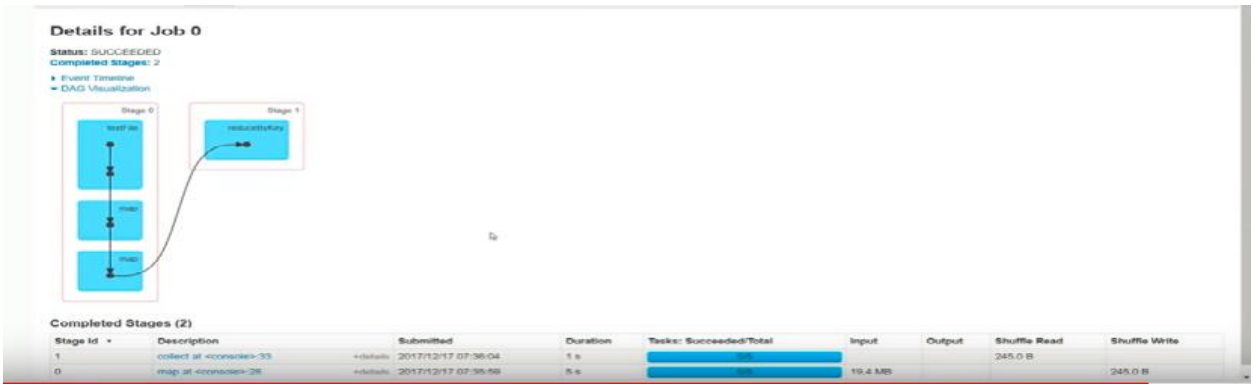
5. Paste the first four lines in the shell.
6. Check the Spark UI.

At this stage, you won't see any Spark Jobs. There is a reason for that.
All these functions that we executed on various RDDs are lazy. They don't perform anything until you run a Function that is not lazy. We call the lazy Functions as transformations. The non-lazy functions are Actions.

RDDs offer two types of operations.

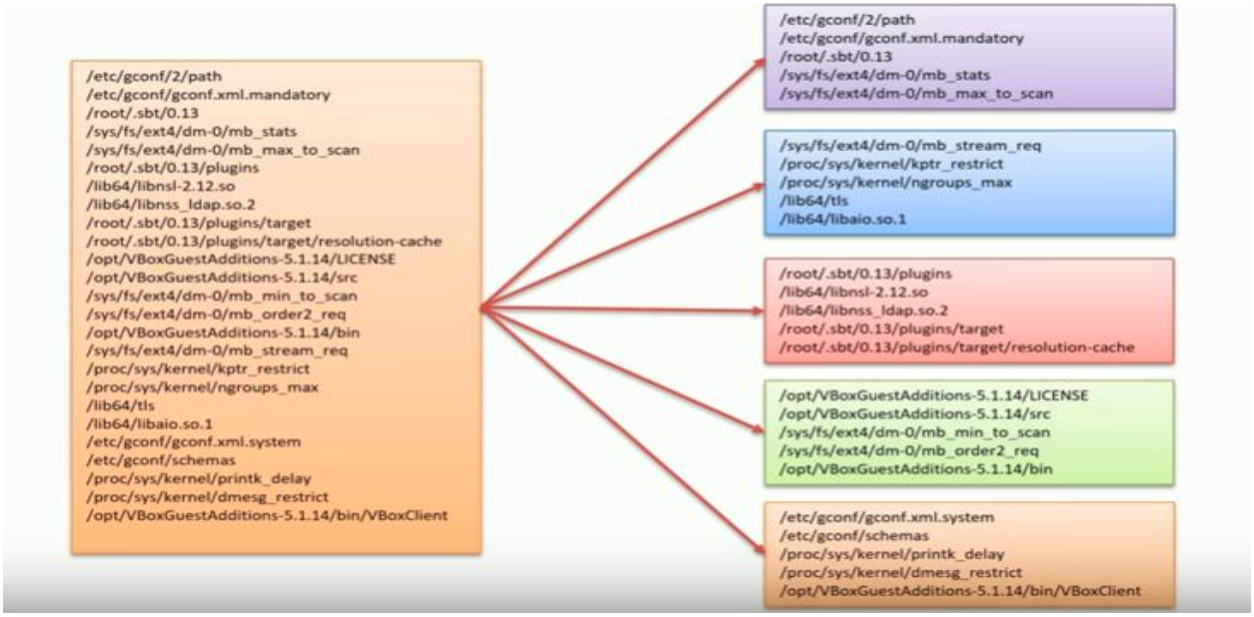
1. Transformations
2. Actions

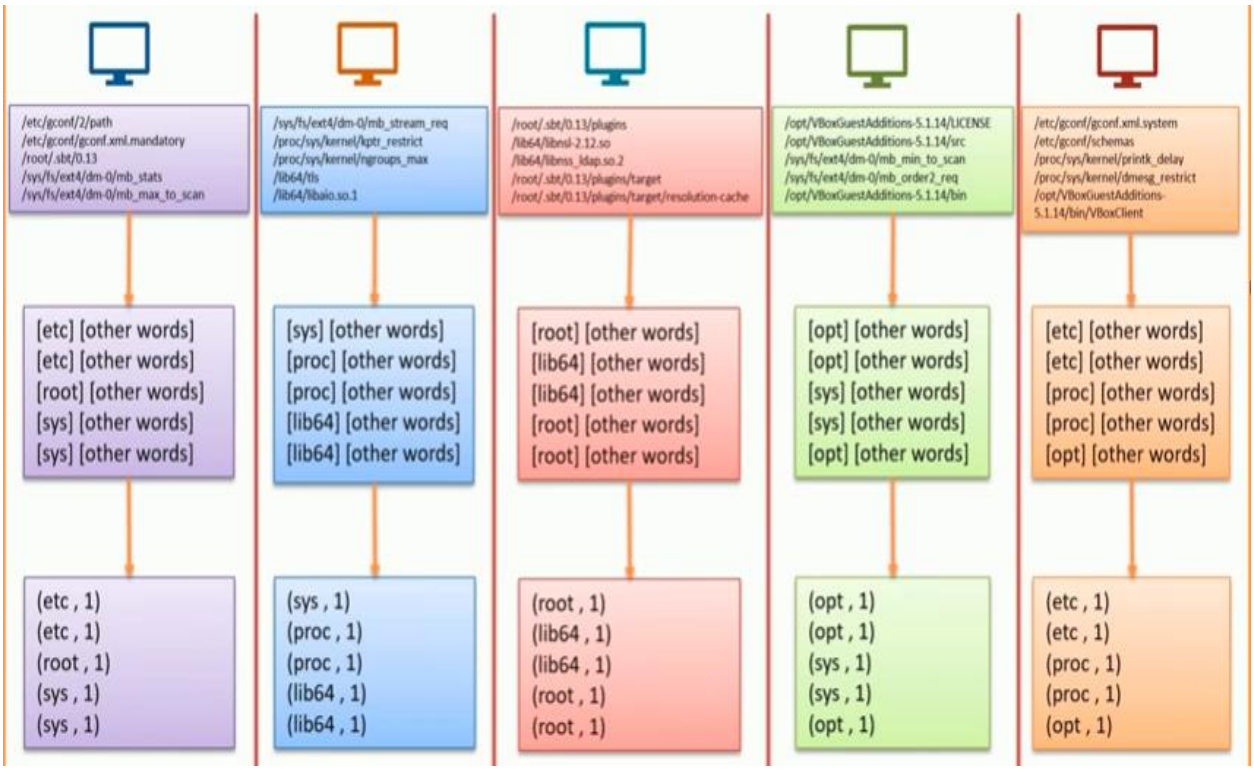
The transformation operations create a new distributed dataset from an existing distributed dataset. So, they create a new RDD from an existing RDD.
The Actions are mainly performed to send results back to the driver, and hence they produce a non-distributed dataset.
The map and the reduceByKey are transformations whereas collect is an action.
All transformations in Spark are lazy. That means, they don't compute results until an action requires them to provide results. That's why you won't see any jobs in your Spark UI.
Now, you can execute the action on the final RDD and check the Spark UI once again.
The video shows you that there is one job, two stages, and ten tasks.
Apache Spark has completed the Job in two Stages. It couldn't do it in a single Stage, and we will look at the reason in a minute. But for now, just realize that the Spark has completed that Job in two stages. We had five partitions. So, I expected five tasks. Since the job went into two Stages, you will see ten Tasks. Five Task for each stage.

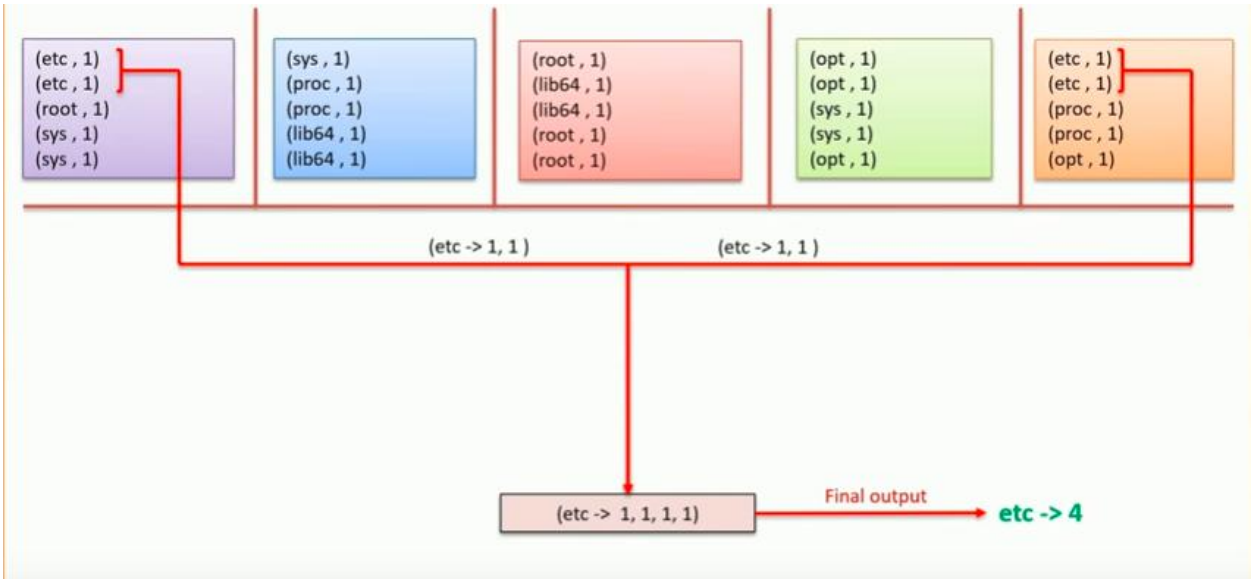


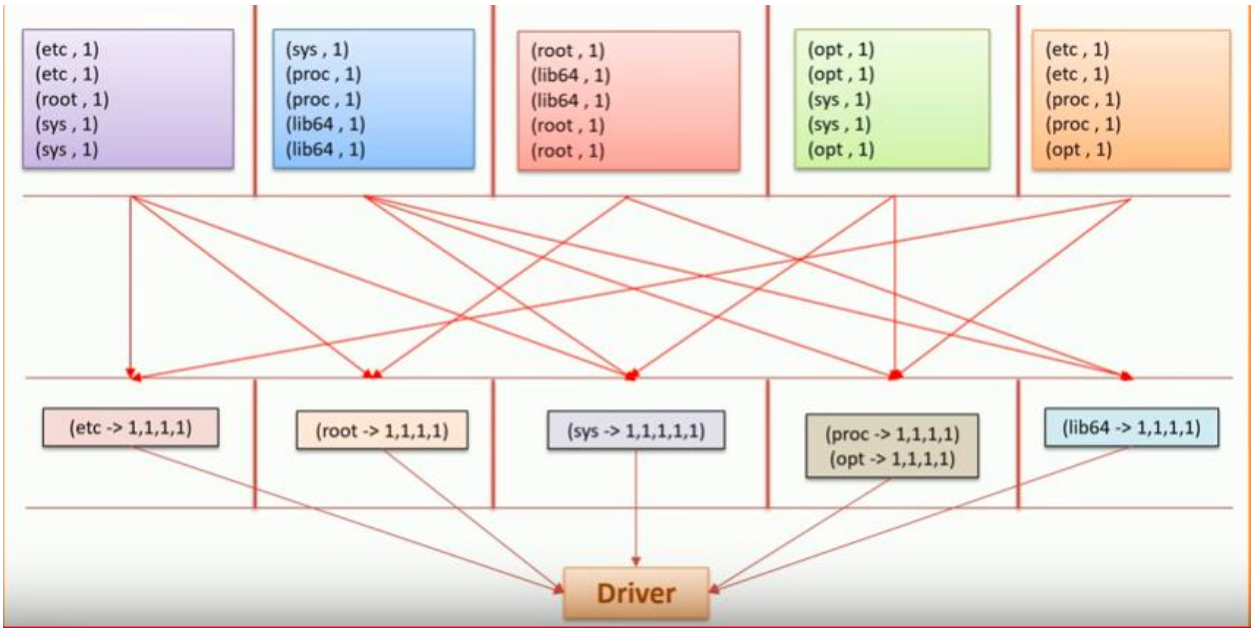
You will also see a DAG. The Spark DAG shows the whole process in a nice visualization. The DAG shows that the Spark was able to complete first three activities in a single stage. But for the ReduceByKey function, it took a new Stage. The question is Why?

The video shows you a logical diagram to explain the reason and shows the shuffle and sort activity. The Shuffle is the main reason behind a new stage. So, whenever there is a need to move data across the executors, Spark needs a new Stage. Spark engine will identify such needs and break the Job into two Stages.




/etc/gconf/2/path
/etc/gconf/gconf.xml.mandatory
/root/.sbt/0.13
/sys/fs/ext4/dm-0/mb_stats
/sys/fs/ext4/dm-0/mb_max_to_scan
/sys/fs/ext4/dm-0/mb_stream_req
/proc/sys/kernel/kptr_restrict
/proc/sys/kernel/ngroups_max
/lib64/tls
/lib64/libaio.so.1
/root/.sbt/0.13/plugins
/lib64/libnsl-2.12.so
/lib64/libnss_ldap.so.2
/root/.sbt/0.13/plugins/target
/root/.sbt/0.13/plugins/target/resolution-cache
/opt/VBoxGuestAdditions-5.1.14/LICENSE
/opt/VBoxGuestAdditions-5.1.14/src
/sys/fs/ext4/dm-0/mb_min_to_scan
/sys/fs/ext4/dm-0/mb_order2_req
/opt/VBoxGuestAdditions-5.1.14/bin
/etc/gconf/gconf.xml.system
/etc/gconf/schemas
/proc/sys/kernel/printk_delay
/proc/sys/kernel/dmesg_restrict
/opt/VBoxGuestAdditions-5.1.14/bin/VBoxClient
[etc] [other words]
[etc] [other words]
[root] [other words]
[sys] [other words]
[sys] [other words]
[sys] [other words]
[proc] [other words]
[proc] [other words]
[lib64] [other words]
[lib64] [other words]
[root] [other words]
[lib64] [other words]
[lib64] [other words]
[root] [other words]
[root] [other words]
[opt] [other words]
[opt] [other words]
[sys] [other words]
[sys] [other words]
[opt] [other words]
[etc] [other words]
[etc] [other words]
[proc] [other words]
[proc] [other words]
[opt] [other words]
(etc , 1)
(etc , 1)
(root , 1)
(sys , 1)
(sys , 1)
(sys , 1)
(proc , 1)
(proc , 1)
(lib64 , 1)
(lib64 , 1)
(root , 1)
(lib64 , 1)
(lib64 , 1)
(root , 1)
(root , 1)
(opt , 1)
(opt , 1)
(sys , 1)
(sys , 1)
(opt , 1)
(etc , 1)
(etc , 1)
(proc , 1)
(proc , 1)
(opt , 1)


(etc , 1)
(etc , 1)
(root , 1)
(sys , 1)
(sys , 1)
(sys , 1)
(proc , 1)
(proc , 1)
(lib64 , 1)
(lib64 , 1)
(root , 1)
(lib64 , 1)
(lib64 , 1)
(root , 1)
(root , 1)
(opt , 1)
(opt , 1)
(sys , 1)
(sys , 1)
(opt , 1)
(etc , 1)
(etc , 1)
(proc , 1)
(proc , 1)
(opt , 1)
(etc -> 1, 1)
(etc -> 1, 1)
(etc -> 1, 1, 1, 1)
Final output
etc -> 4



While learning Apache Spark, whenever you do something that needs moving data, for example, a group by operation or a join operation, you will notice a new Stage.
Once we have these key based partitions, it is simple for each executor to calculate the count for the keys that they own. Finally, they send these results to the driver because we executed the collect method.
This video gives you a fair Idea about the following things.

1. RDDs – RDDs are the core of Spark. They represent a partitioned and distributed data set.
2. Transformations and Actions – We can perform transformations and actions over the RDDs.
3. Spark Job – An action on an RDD triggers a job.
4. Stages – Spark breaks the job into stages.Shuffle and Sort – A shuffle activity is a reason to break the job into two stages.
5. Shuffle and Sort – A shuffle activity is a reason to break the job into two stages.
6. Tasks – Each stage is executed in parallel tasks. The number of parallel tasks is directly dependent on the number of partitions.
7. Executors – Apart from the tasks, the number of available executors is also a constraint on the degree of parallelism.