Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Cluster and Runtime Architecture

Lecture:
Spark Cluster

Spark Cluster

In the previous lectures, we have learned to load and process Spark data in two ways:

1. Using Spark SQL
2. Using Spark Dataframe API

Apache Spark is a distributed computing platform. So, we write code and then run it on a distributed Spark Cluster.

Spark runs on the following type of clusters.

1. Hadoop Cluster
2. Non-Hadoop Cluster

The Hadoop cluster runs two primary services:

1. Hadoop YARN
2. Hadoop HDFS

The YARN comprises a resource manager on the master and a node manager on each worker machine. The Hadoop HDFS comprises a name node service on the master and a data node service on each worker machine. The YARN resource manager and the node manager give you the computation facility. And the HDFS gives you the storage facility.

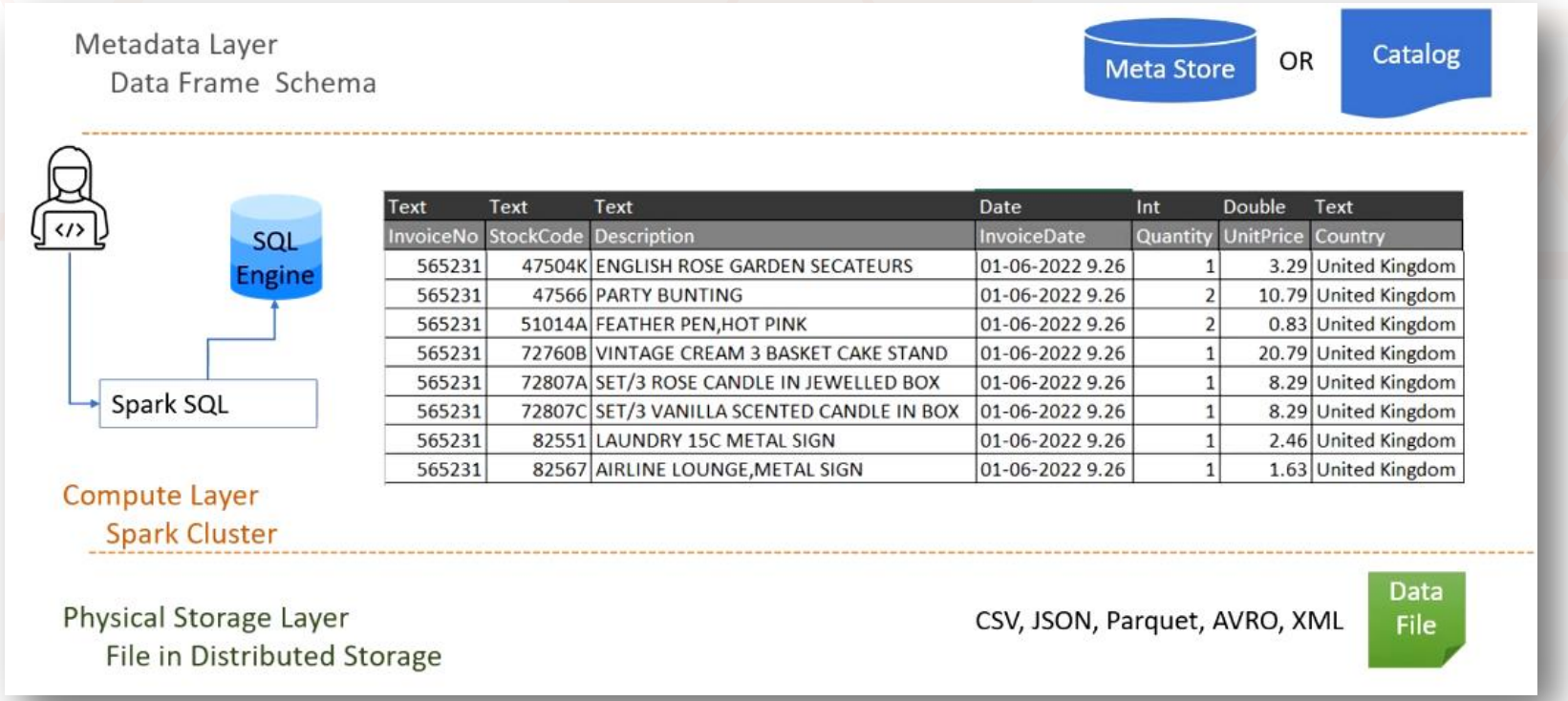
Spark Dataframe and data tables are organized into three layers.

1. Metadata layer
2. Compute Layer
3. Storage Layer

The first layer is the metadata layer, and it stores table and dataframe metadata.

The compute layer executes the Spark SQL engine and your application code.

And the storage layer keeps your data file.



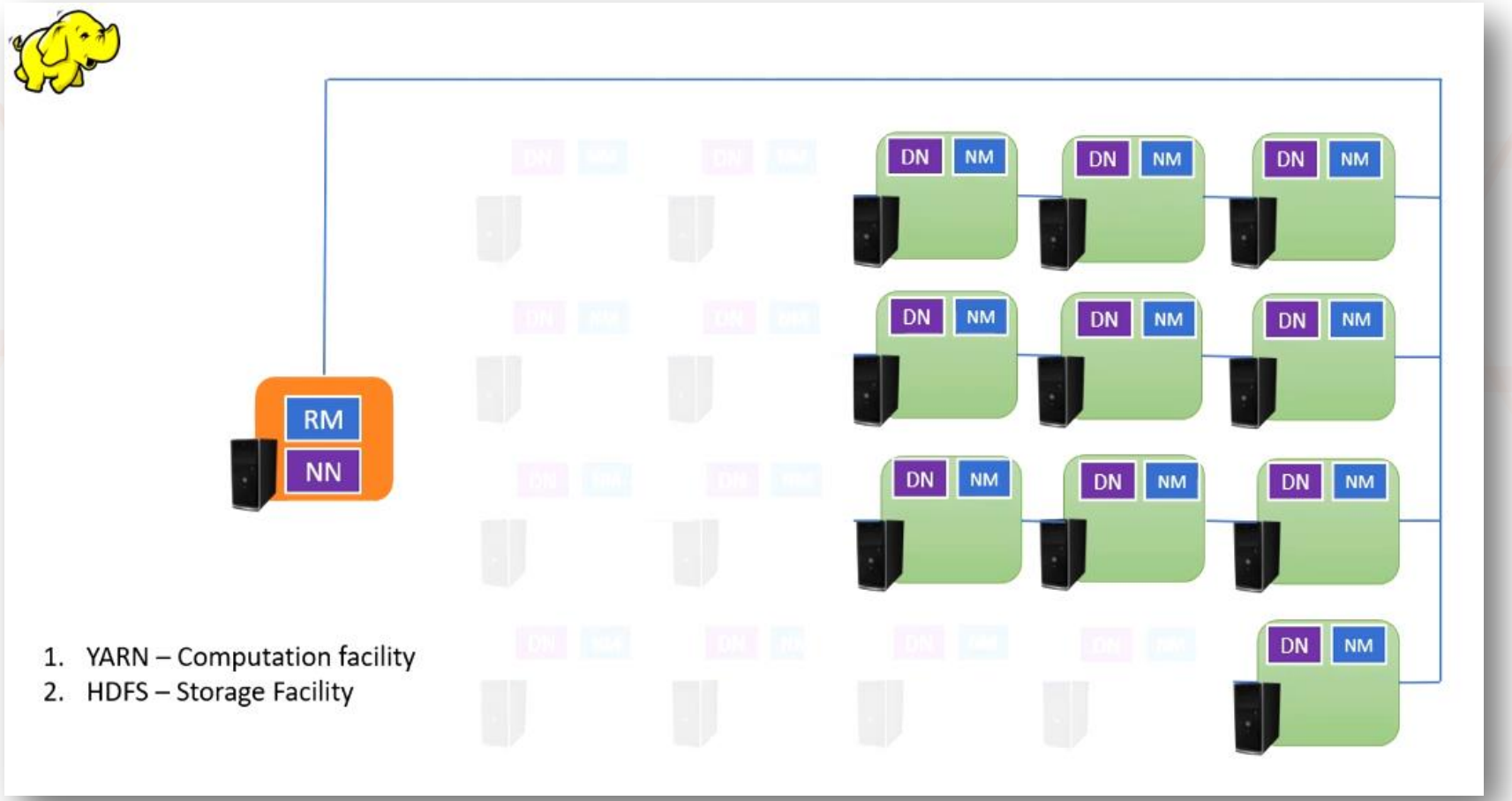
Text	Text	Text	Date	Int	Double	Text
InvoiceNo	StockCode	Description	InvoiceDate	Quantity	UnitPrice	Country
565231	47504K	ENGLISH ROSE GARDEN SECATEURS	01-06-2022 9.26	1	3.29	United Kingdom
565231	47566	PARTY BUNTING	01-06-2022 9.26	2	10.79	United Kingdom
565231	51014A	FEATHER PEN,HOT PINK	01-06-2022 9.26	2	0.83	United Kingdom
565231	72760B	VINTAGE CREAM 3 BASKET CAKE STAND	01-06-2022 9.26	1	20.79	United Kingdom
565231	72807A	SET/3 ROSE CANDLE IN JEWELLED BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	72807C	SET/3 VANILLA SCENTED CANDLE IN BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	82551	LAUNDRY 15C METAL SIGN	01-06-2022 9.26	1	2.46	United Kingdom
565231	82567	AIRLINE LOUNGE,METAL SIGN	01-06-2022 9.26	1	1.63	United Kingdom

The HDFS is the storage service of the Hadoop cluster. So, the HDFS becomes the storage layer of Spark tables and dataframe. Similarly, the YARN becomes the compute layer.

Text	Text	Text	Date	Int	Double	Text
InvoiceNo	StockCode	Description	InvoiceDate	Quantity	UnitPrice	Country
565231	47504K	ENGLISH ROSE GARDEN SECATEURS	01-06-2022 9.26	1	3.29	United Kingdom
565231	47566	PARTY BUNTING	01-06-2022 9.26	2	10.79	United Kingdom
565231	51014A	FEATHER PEN,HOT PINK	01-06-2022 9.26	2	0.83	United Kingdom
565231	72760B	VINTAGE CREAM 3 BASKET CAKE STAND	01-06-2022 9.26	1	20.79	United Kingdom
565231	72807A	SET/3 ROSE CANDLE IN JEWELLED BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	72807C	SET/3 VANILLA SCENTED CANDLE IN BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	82551	LAUNDRY 15C METAL SIGN	01-06-2022 9.26	1	2.46	United Kingdom
565231	82567	AIRLINE LOUNGE,METAL SIGN	01-06-2022 9.26	1	1.63	United Kingdom

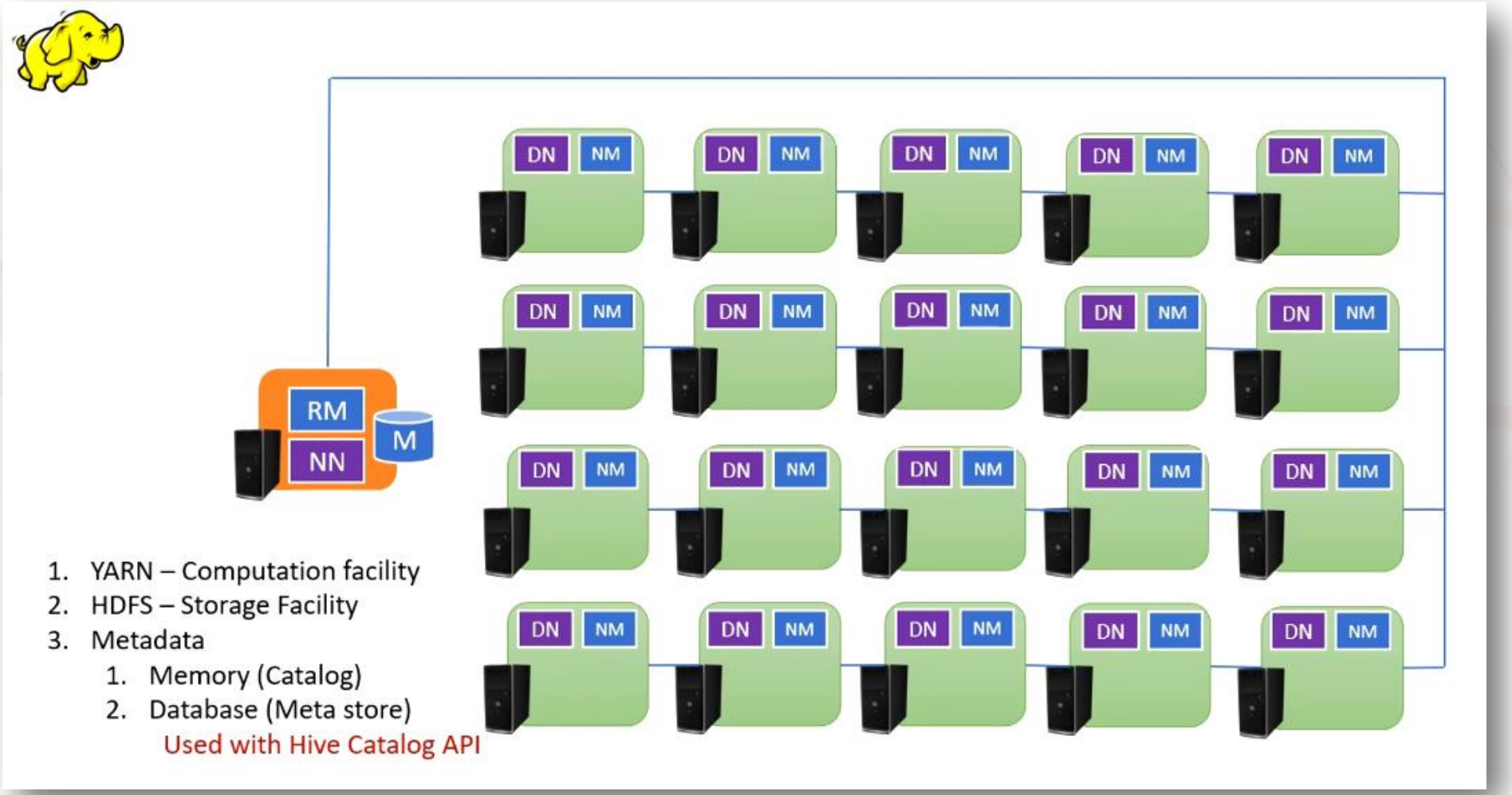
In the diagram, the blue boxes represent the YARN service, so they make the compute layer of Spark. And all the purple boxes make the HDFS, and they are Spark's storage layer.

We have twenty worker nodes in this cluster. If we remove ten nodes from this cluster, we will lose some storage and the compute capacity of the cluster. Removing one node will reduce the YARN service count and the HDFS service count. So, we lose both the services: Node Manager and Data Node.



Every time we add one more node, it will increase the storage and compute both. And when we remove one worker node, it will reduce storage and compute both.
We might need 100 PB of storage but only 50 CPUs to run our application. So, we will buy a lot of storage and a little CPU power.

Metadata are stored both in the memory and in the database.
Also, we installed MySQL database on the master node. This MySQL database is used as a Hive metadata store.



Important Interview Question:

Where does Spark store its metadata?

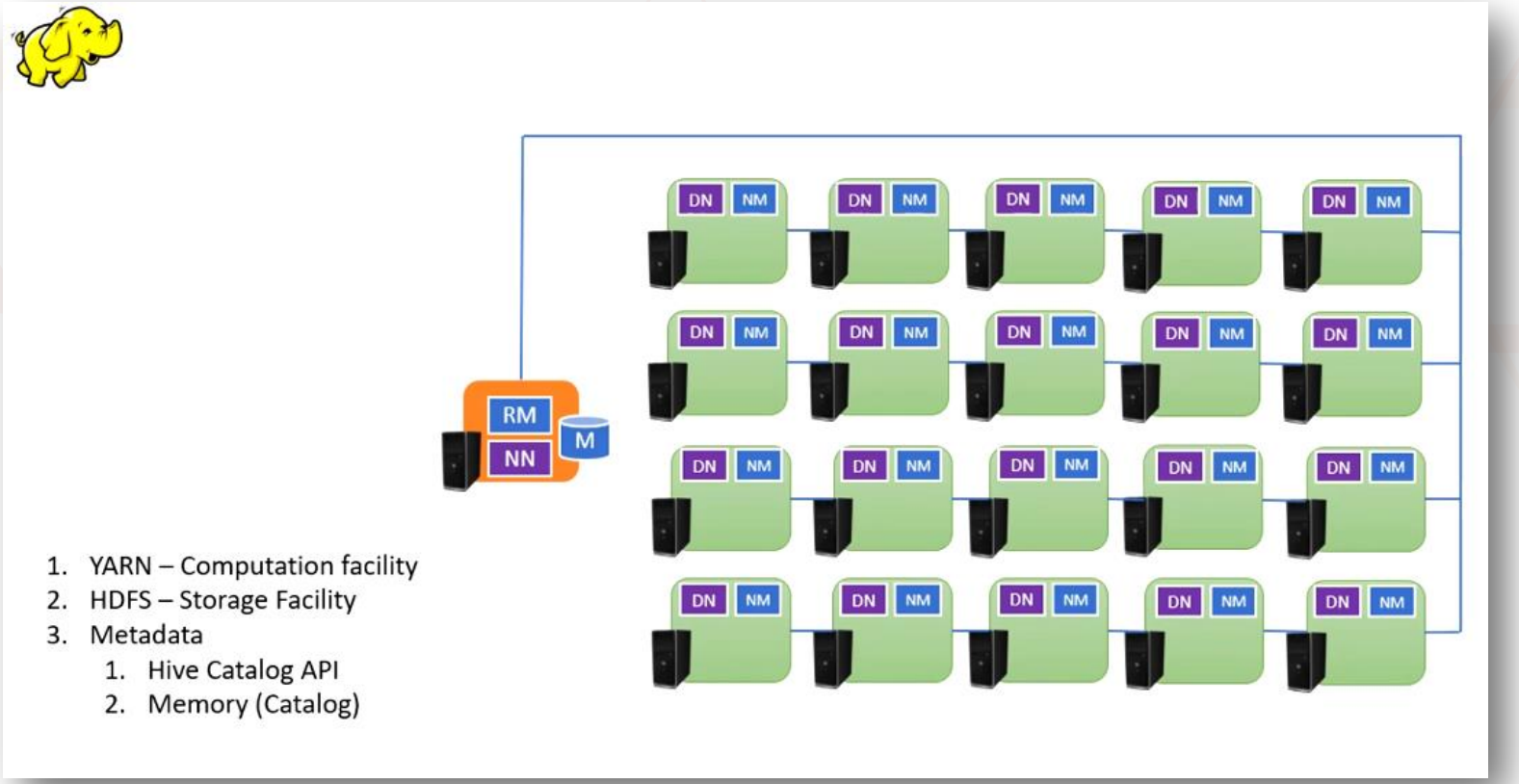
Short answer: Spark uses Hive's metadata catalog to store its metadata.

A **long answer** should convey the following points:

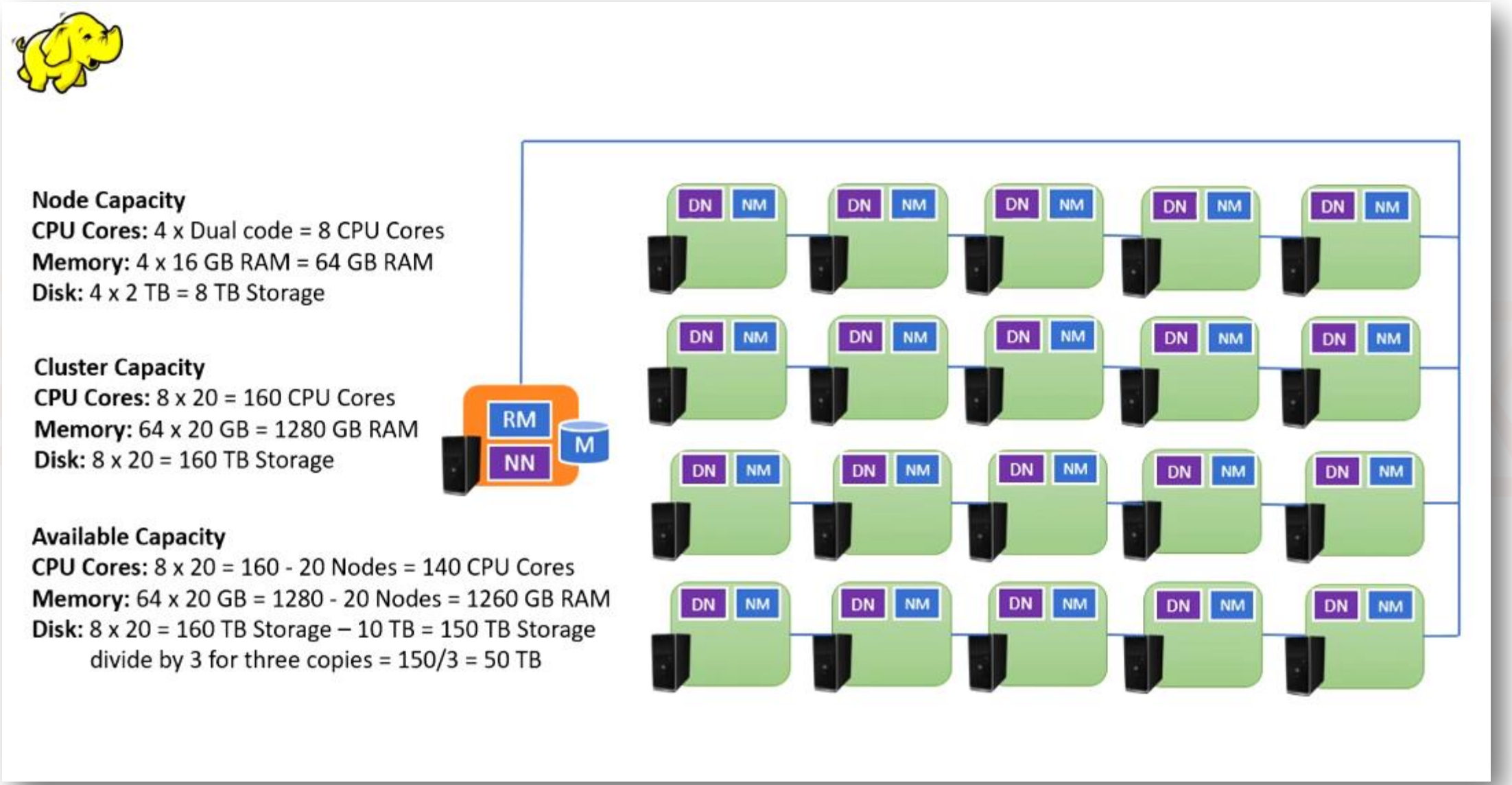
1. Spark on Hadoop stores metadata in the Hive metadata store.
2. Hive metadata store is backed by the MySQL database and a Hive metadata catalog API.
3. MySQL database is not mandatory. We can configure Hadoop to use three other RDBMS such as PostgreSQL, Oracle, and MS SQL Server.

We have three layers of Apache spark on the Hadoop cluster.

1. Spark Compute Layer is built on top of Hadoop YARN
2. Spark storage layer is built on top of Hadoop HDFS
3. Spark metadata layer :
    a) Built on top of Hive metadata catalog
    b) The Spark dataframe catalog lives in the compute layer memory.



The following diagram shown below depicts the cluster capacity.



Node Capacity
CPU Cores: 4 x Dual code = 8 CPU Cores
Memory: 4 x 16 GB RAM = 64 GB RAM
Disk: 4 x 2 TB = 8 TB Storage

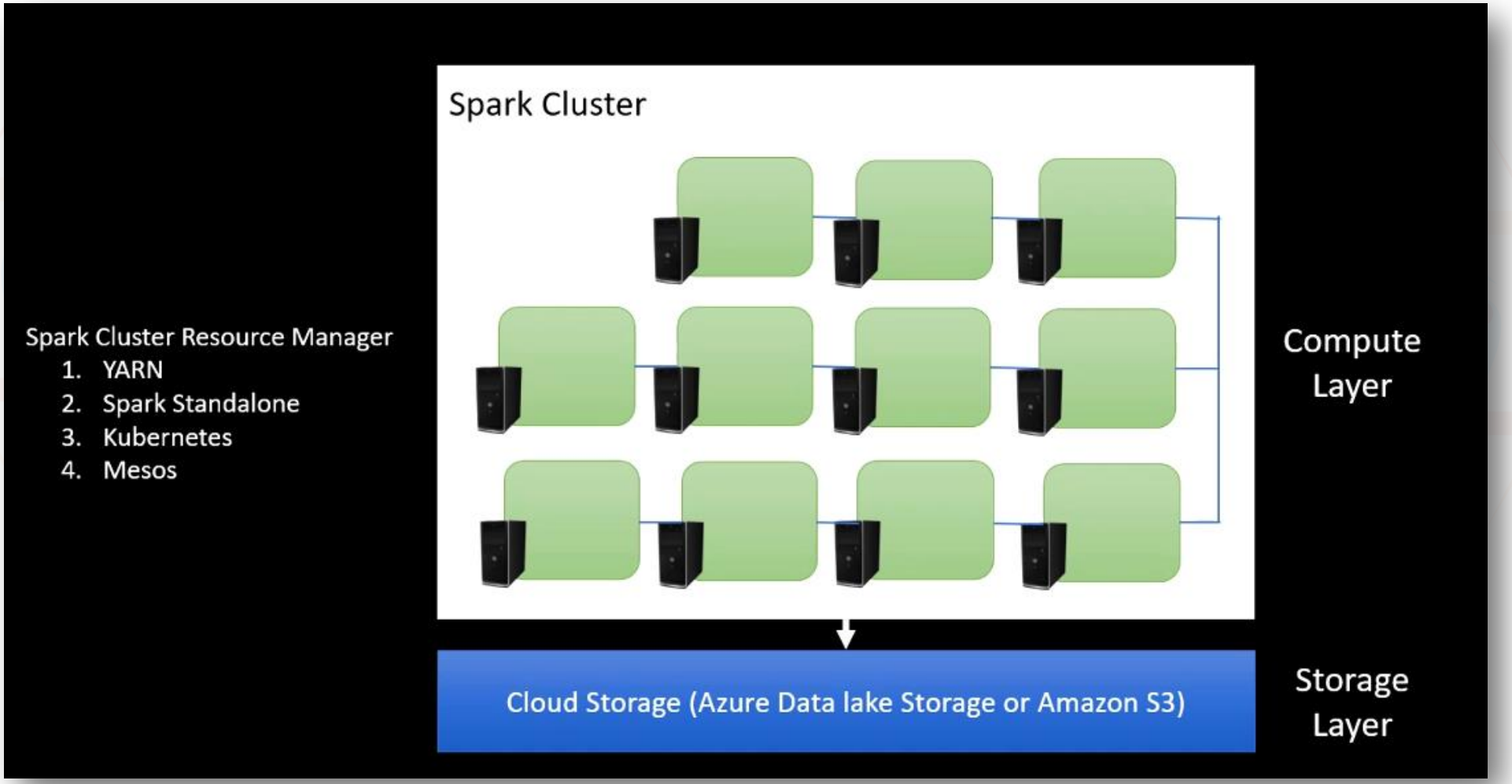
Cluster Capacity
CPU Cores: 8 x 20 = 160 CPU Cores
Memory: 64 x 20 GB = 1280 GB RAM
Disk: 8 x 20 = 160 TB Storage

Available Capacity
CPU Cores: 8 x 20 = 160 - 20 Nodes = 140 CPU Cores
Memory: 64 x 20 GB = 1280 - 20 Nodes = 1260 GB RAM
Disk: 8 x 20 = 160 TB Storage – 10 TB = 150 TB Storage
divide by 3 for three copies = 150/3 = 50 TB

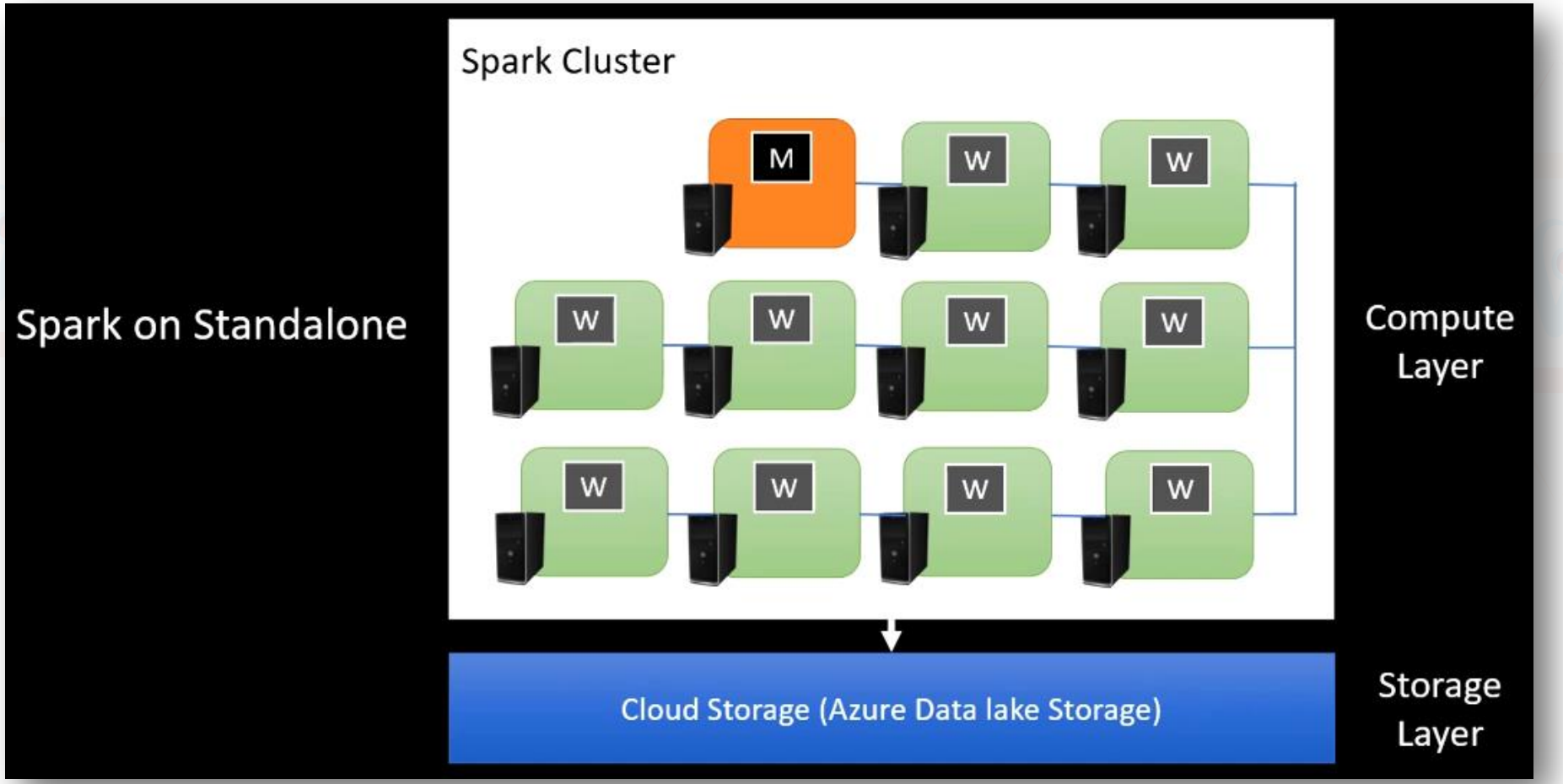
The following options shows how we can run Spark without Hadoop:

1. Spark on a single-node machine
    a) Run Spark on a single node Databricks Community Cloud.
    b) Run Spark on your local machine
2. Spark on Non-Hadoop Cloud cluster
    a) Databricks Spark cluster.

Databricks Spark cluster becomes the compute layer of our Spark. The cloud platform allows us to create a separate storage layer. Spark implemented a choice of the cluster resource manager and offers four options.



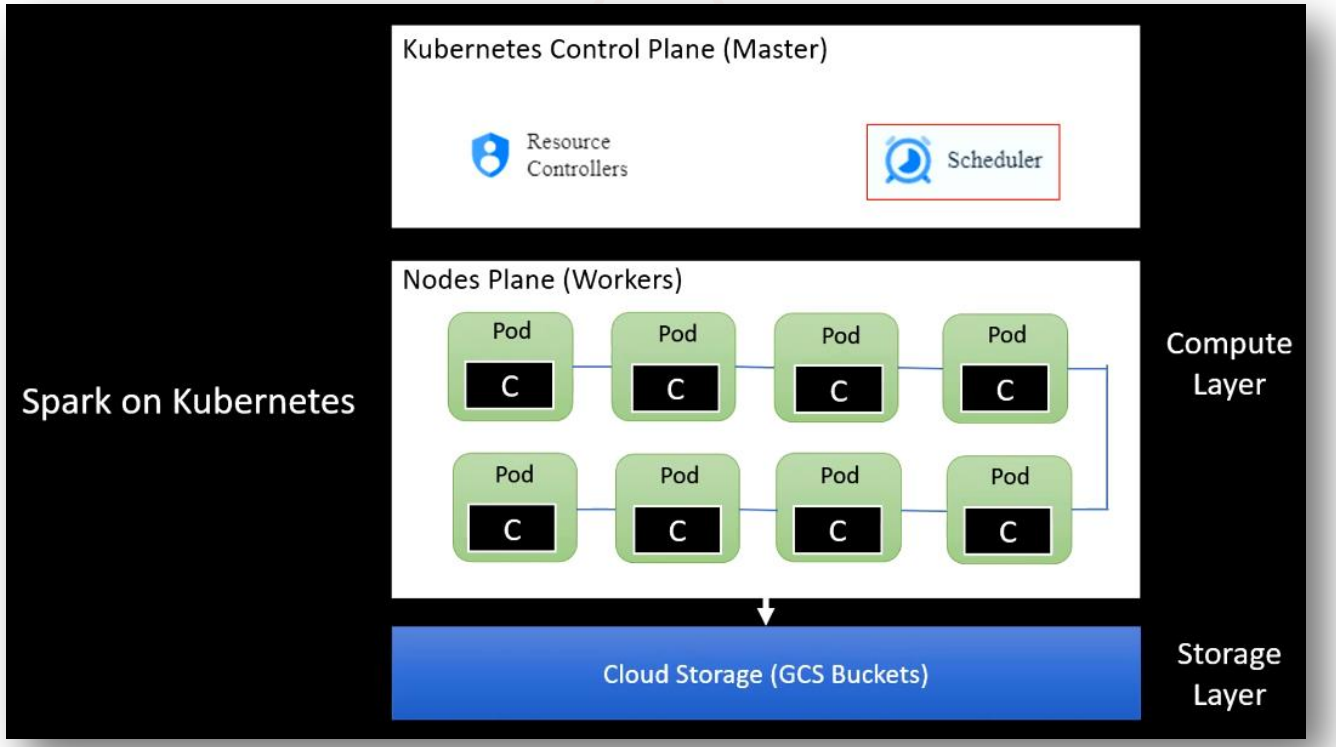
Spark creators also created a new resource manager and named it Spark Standalone resource manager. Databricks cloud will start ten virtual machines in the cloud . It will start one VM to become a Spark master, and others will work as the worker node.



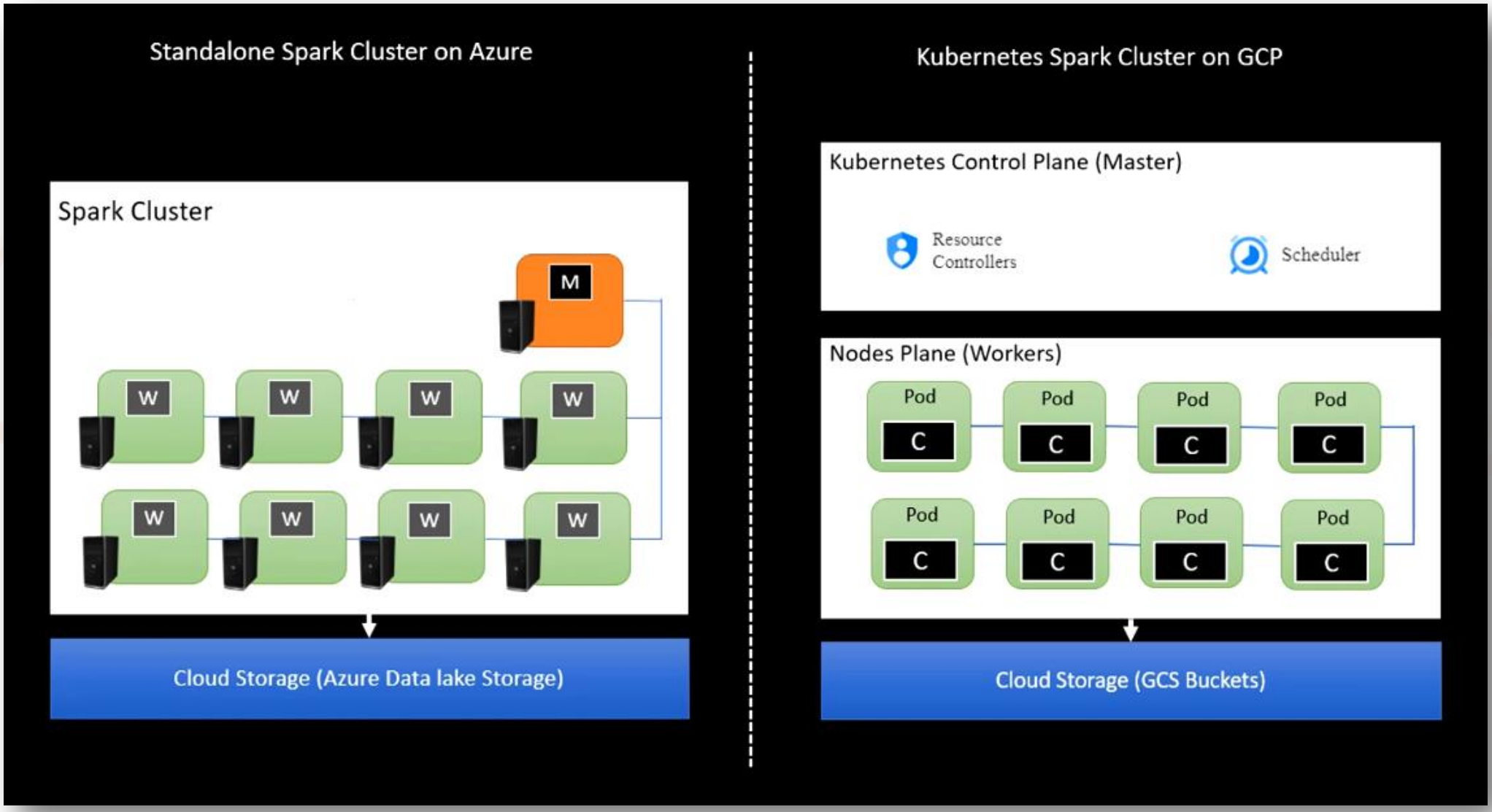
Spark also offered Kubernetes as a cluster resource manager. Kubernetes cluster consists of two main components.

1. Control plane
2. Nodes plane

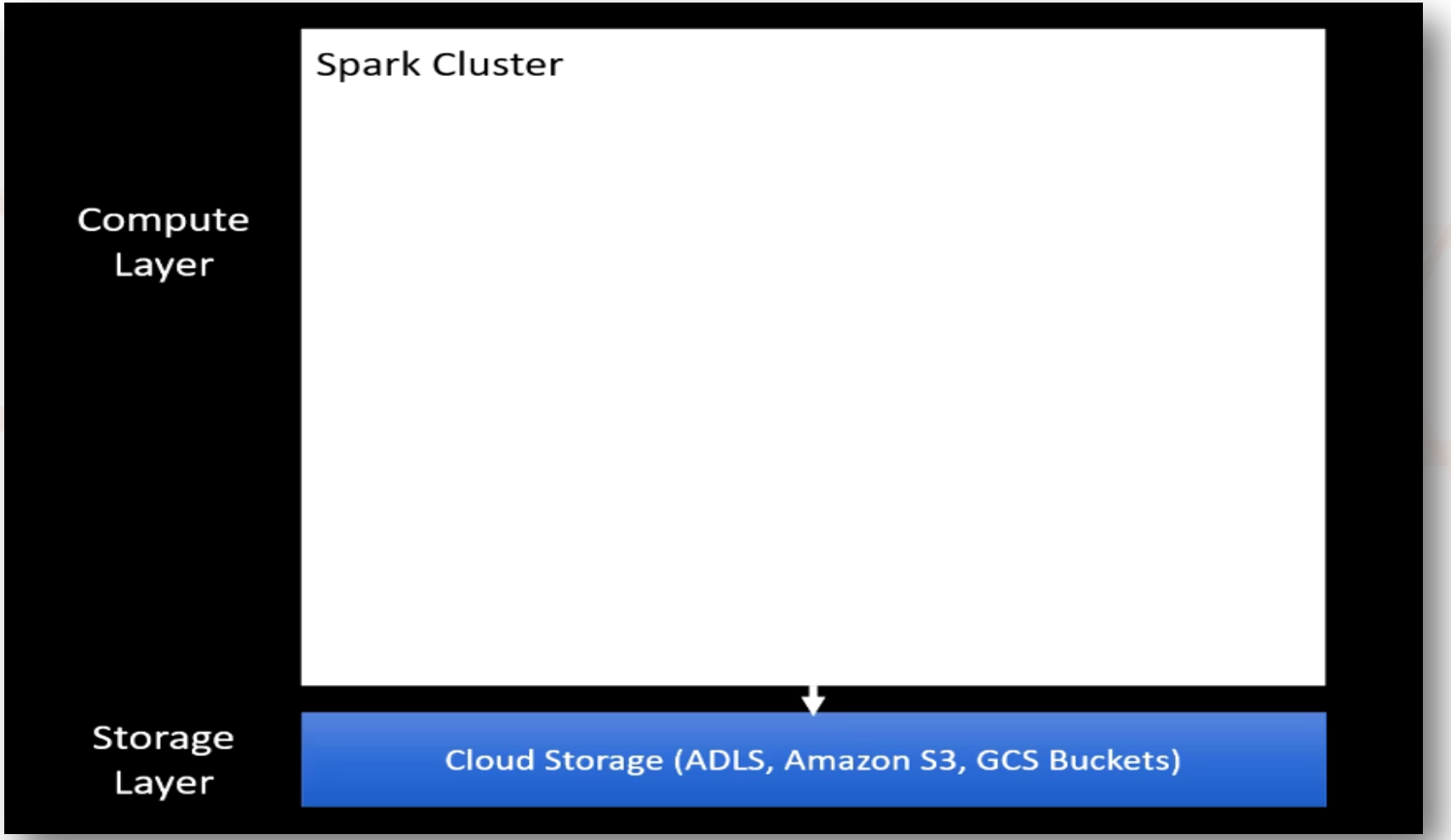
Control plane runs a resource controller and a scheduler service to act as a Kubernetes master. Data plane runs the workers as Kubernetes Pods. These pods run applications inside a Docker container. The control plane works as your Spark master.



Difference between a standalone Spark cluster in Azure and the Kubernetes cluster in Google cloud.



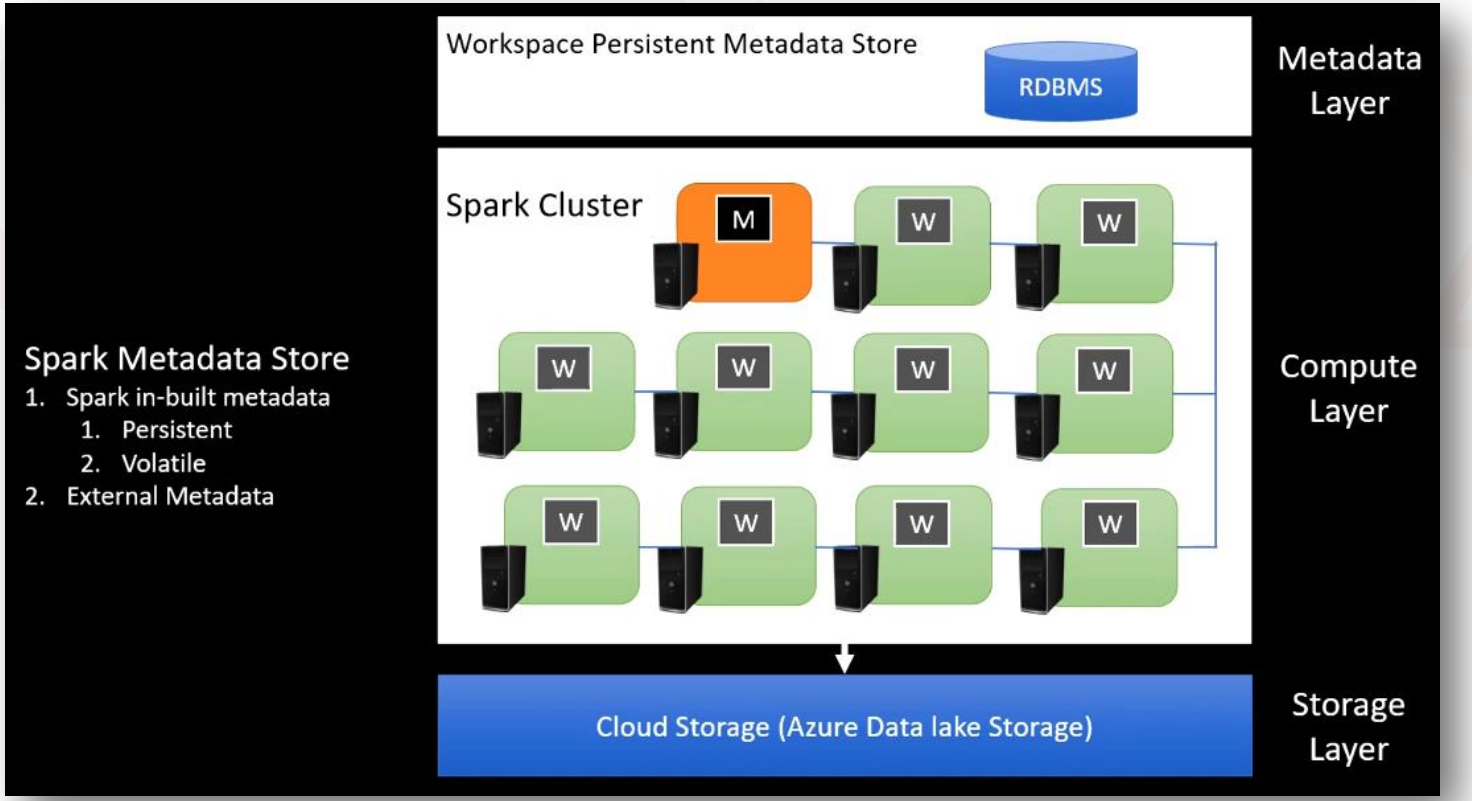
Cloud platforms offer affordable cloud storage. For example, Azure offers Azure Blob storage and Azure Data Lake storage, AWS offers Amazon S3 storage & Google Cloud offers Google Cloud Storage.
If needed, we can purchase more storage from the cloud provider.
I can shut down the Spark cluster when I do not need it and save some money.



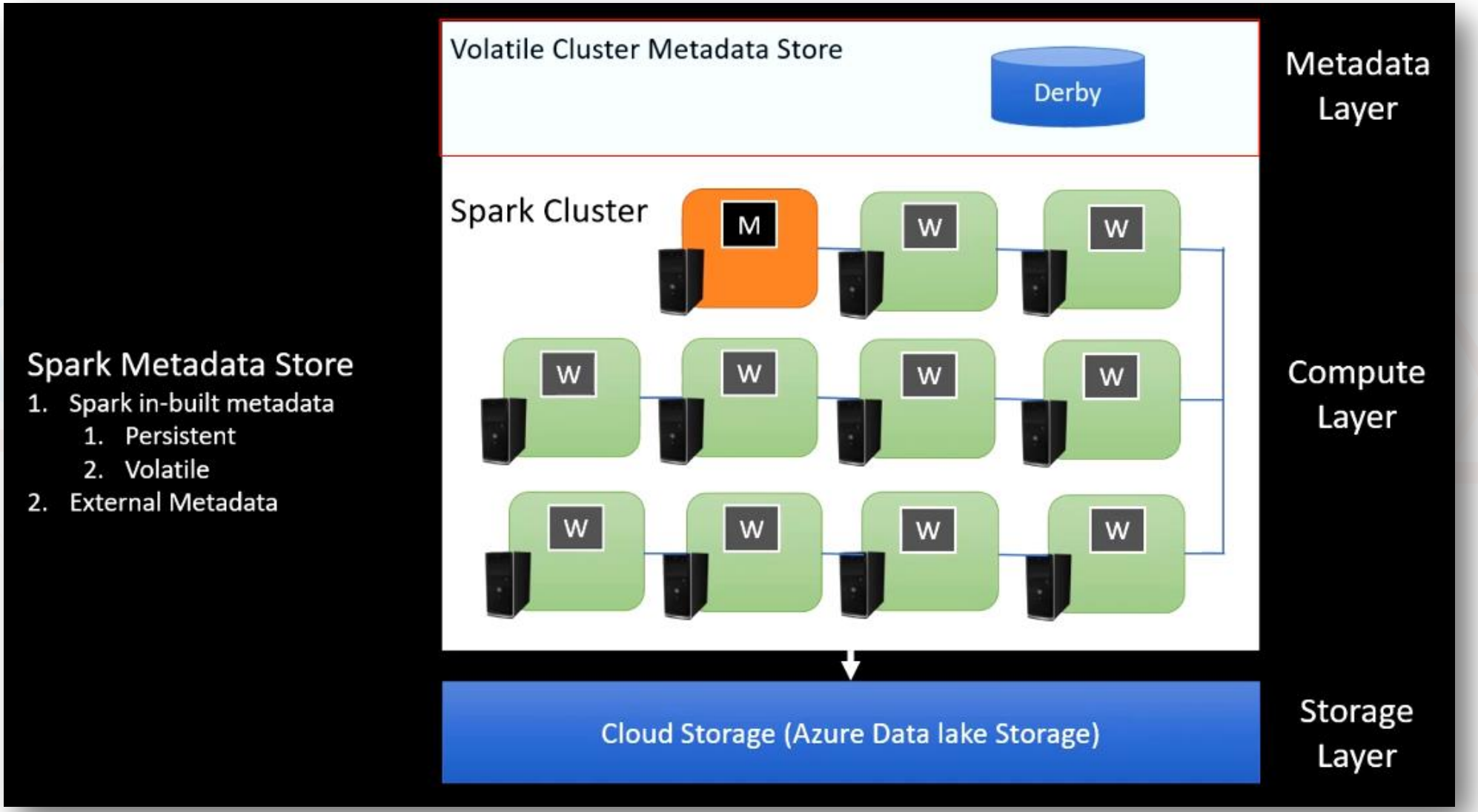
Spark on the cloud platform gives you the following options.

1. Spark in-built metadata
    a) Persistent
    b) Volatile
2. External Metadata

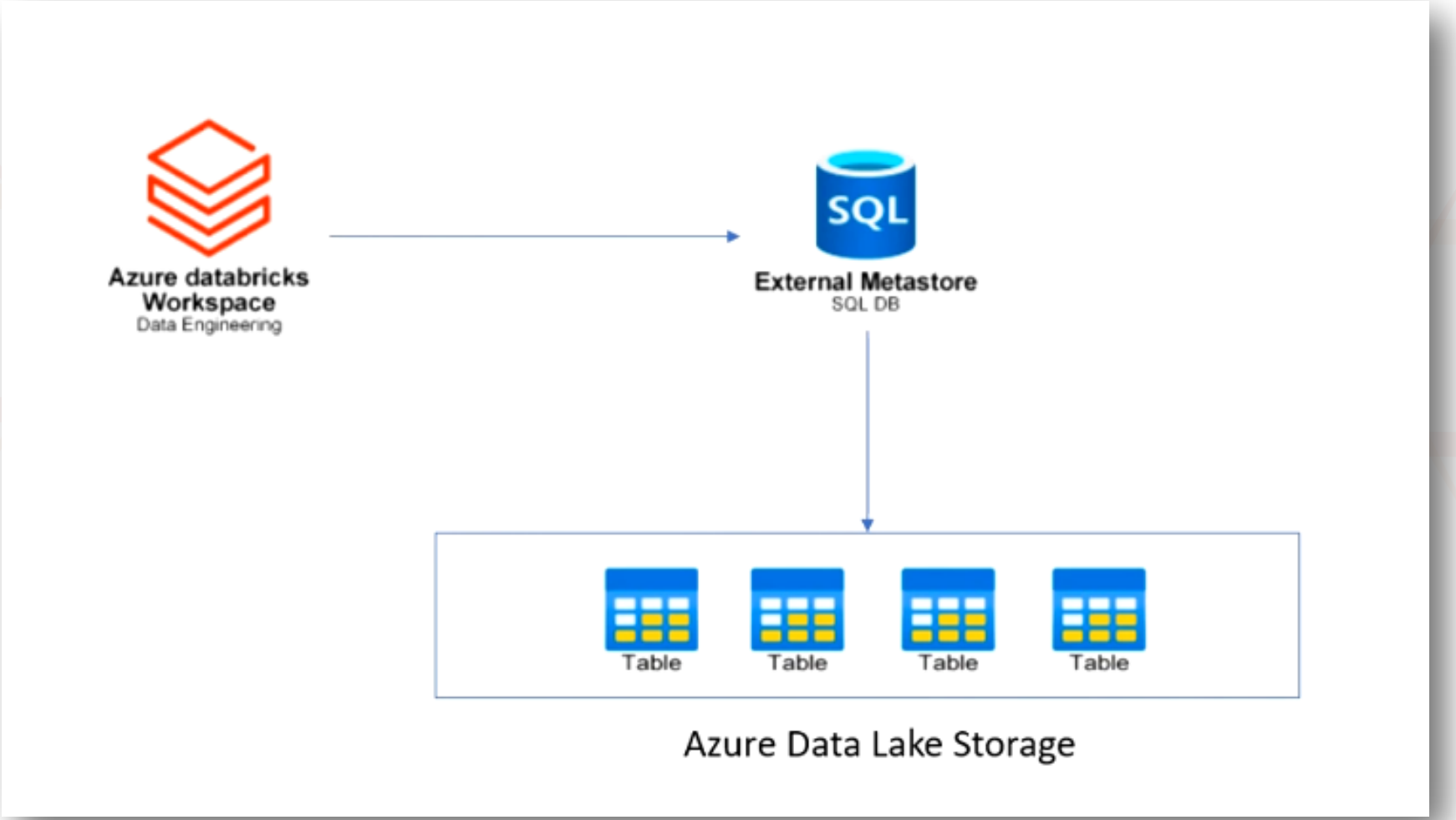
Databricks will create a workspace metadata layer by installing and configuring an RDBMS database. We can create a cluster, delete it, create a new cluster and still use the same metadata.



Databricks creates a volatile metadata store using the Derby database. The Derby database is a fast in-memory database.



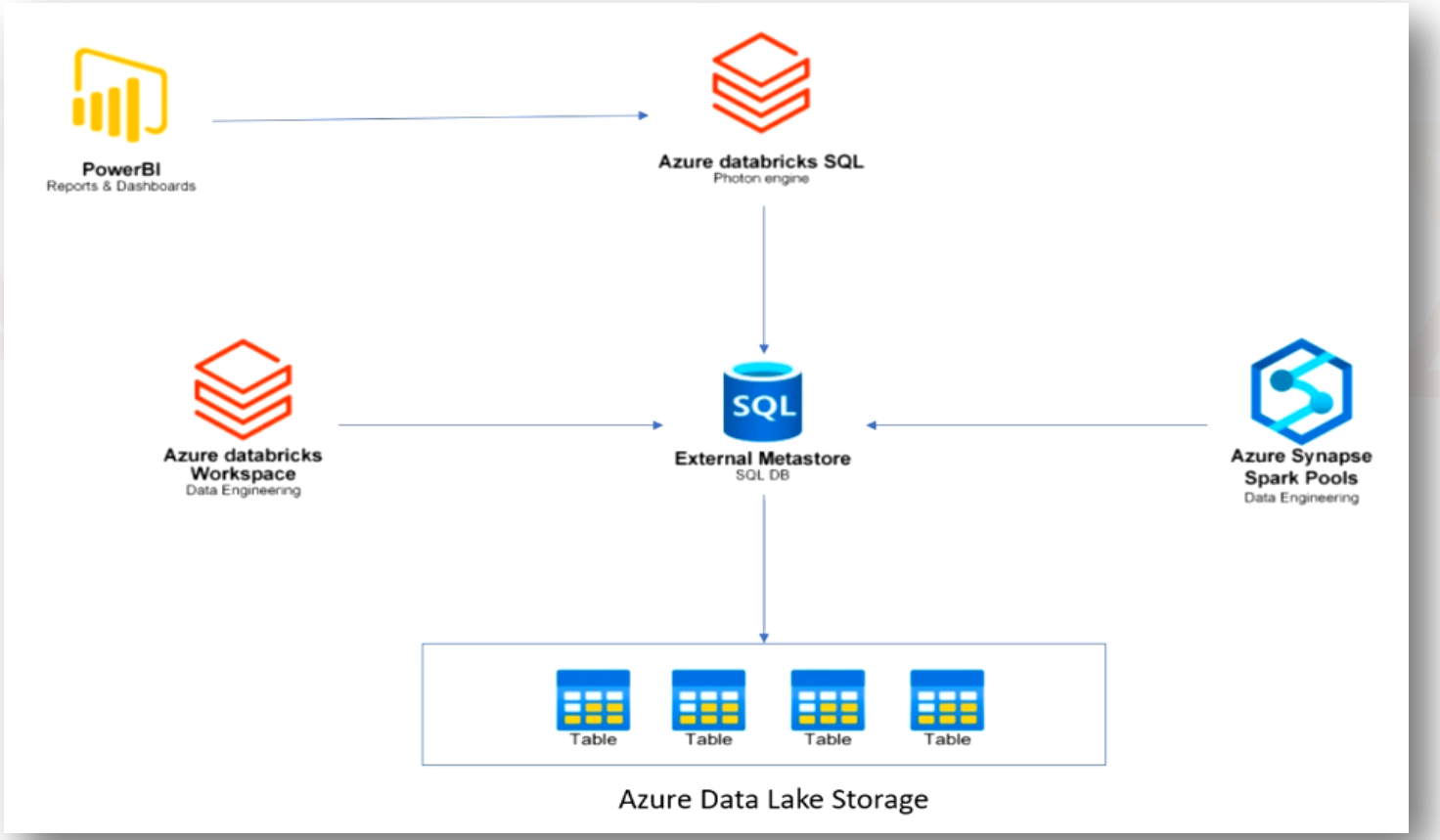
Here is a three-layer architecture of a data engineering project, Azure Databricks workspace is your compute layer, the SQL database is your metadata layer, and the Azure data lake storage is your storage layer.

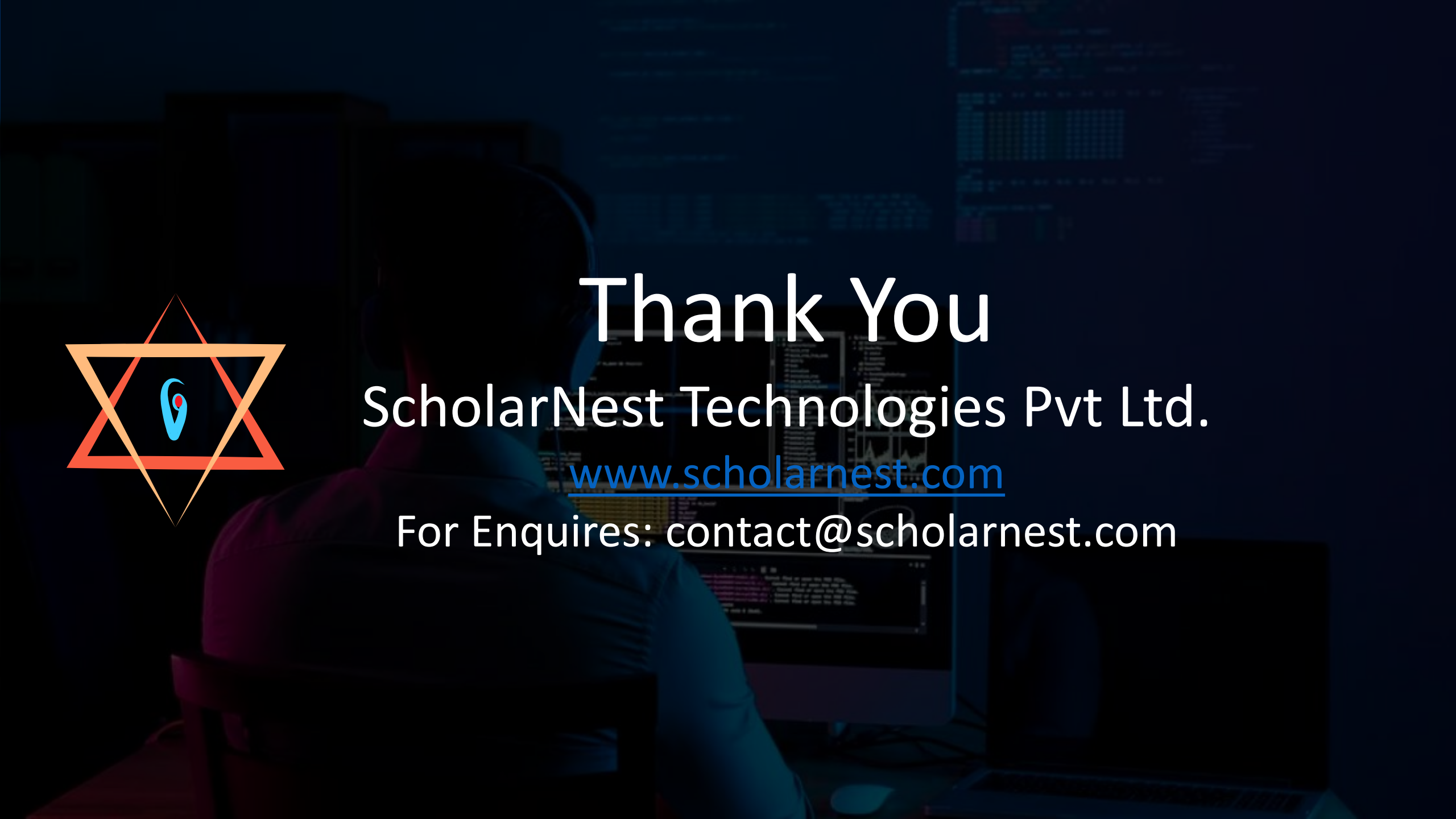


We have two separate compute layers :

1. Data Engineering workspace
2. SQL workspace

The data engineering workspace is optimized for batch processing workload. And the SQL workspace is optimized for interactive SQL workload. We can integrate Azure Synapse with the same metadata store and access your Spark tables from the Azure Synapse on the Azure cloud.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Cluster and Runtime Architecture

Lecture:
Running Spark Application

Running Applications on Spark Cluster

We have two approaches to working with Apache Spark as shown below.
In the interactive approach, we write spark Dataframe code or spark SQL code, run it and see the results immediately. The, we run our completed Spark applications on a Spark cluster as a Spark Job or Spark application. Each of these two approaches can be performed using various applications listed below.

Interactive Approach:

We have already seen the interactive approach earlier while creating Spark Notebooks, writing some code in the notebook cells, and running it to see the results immediately.

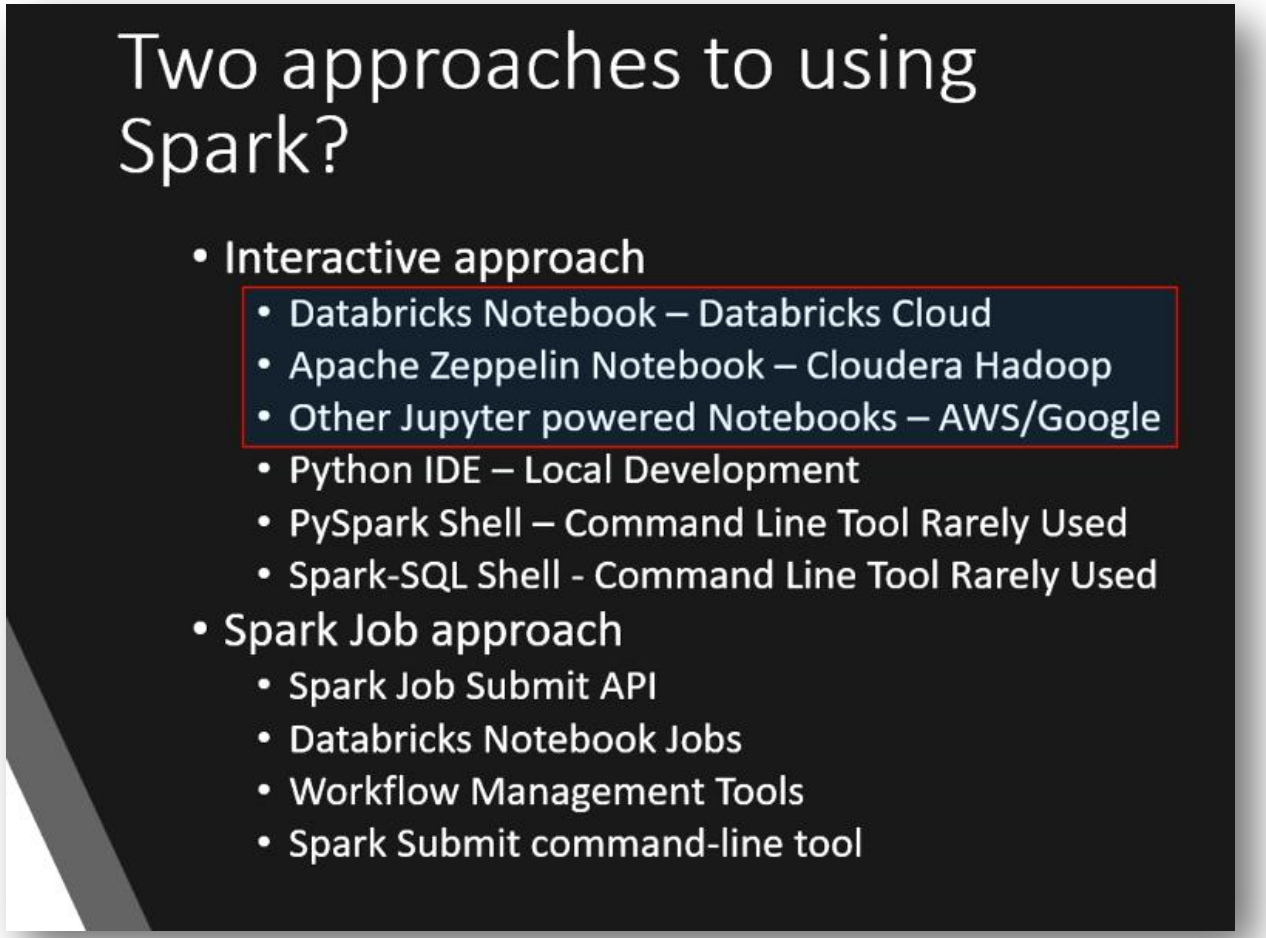
Data Analysts and Data Scientists mainly use this approach.

A Data Analyst uses this approach to explore, understand and analyze the data.

Similarly, Data scientists also use this approach to develop ML models and try and explore things.

However, Spark developers also use the interactive approach during the application development, they write code, run it, and test it to see if things are working.

We have six different tools for using Spark in an interactive mode as shown below. The first three options are Notebook environments. We have already seen Databricks Notebook, however, the other Notebook environments are also the same. The notebook idea was invented by the Jupyter Notebook for Python developers. And all the three notebook environments listed below are internally based on the Jupyter notebooks, but different vendors customize them.



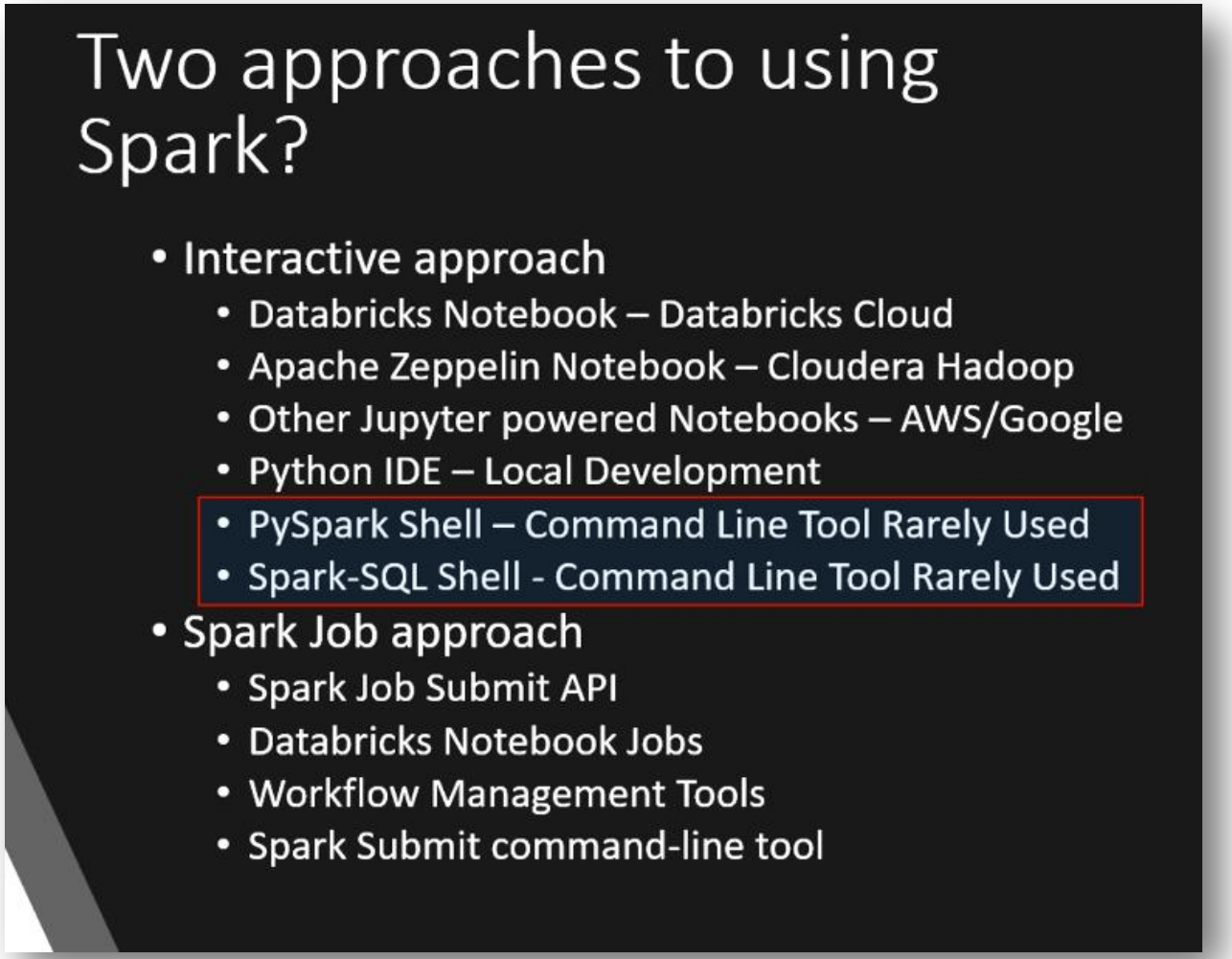
Databricks notebooks are available only in the Databricks environment. So you cannot use Databricks Notebook on other types of Spark clusters.
Cloudera Hadoop gives you an Apache Zeppelin notebook.
Apache Zeppelin is almost the same as the Databricks notebook. And you can use the Zeppelin notebook on Cloudera Hadoop for interactive Spark.
Amazon EMR and Google Dataproc are the other two spark clusters. They also have their customized versions of Jupyter notebooks.
Nowadays, Spark is integrated with many tools and almost every data platform. Databricks and Cloudera are the most popular, but there are many more. But everyone gives you some notebook to work with Spark. And all those notebooks are almost the same user interface.

The next option is to use the Python IDE such as PyCharm. The Python IDE is not suitable for Data Analysts and Data Scientists. But half of the Spark developers community will use the Python IDE.

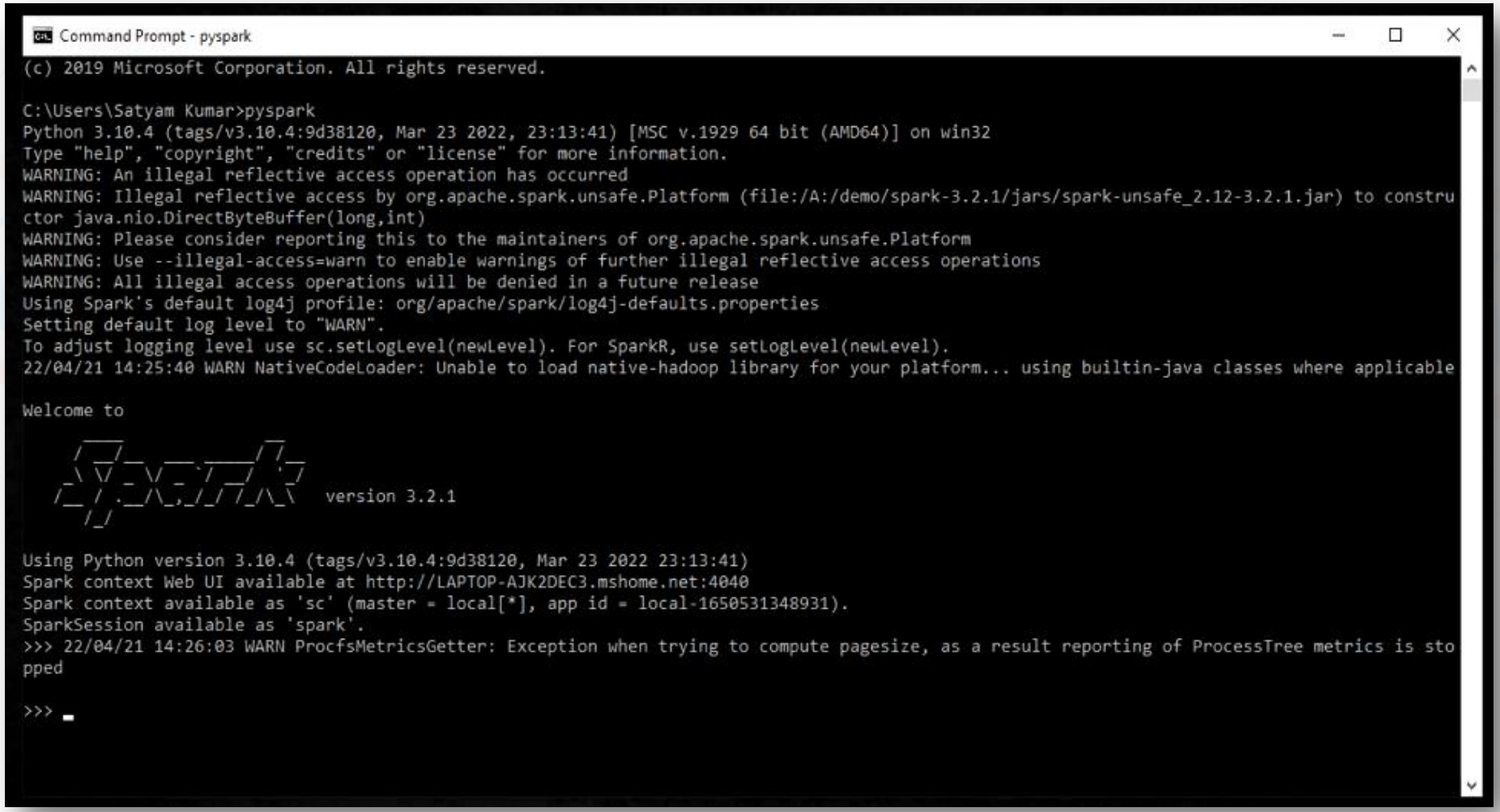
The next set is for the command-line tool. We have two command-line tools:

1. PySpark Shell
2. Spark-SQL Shell

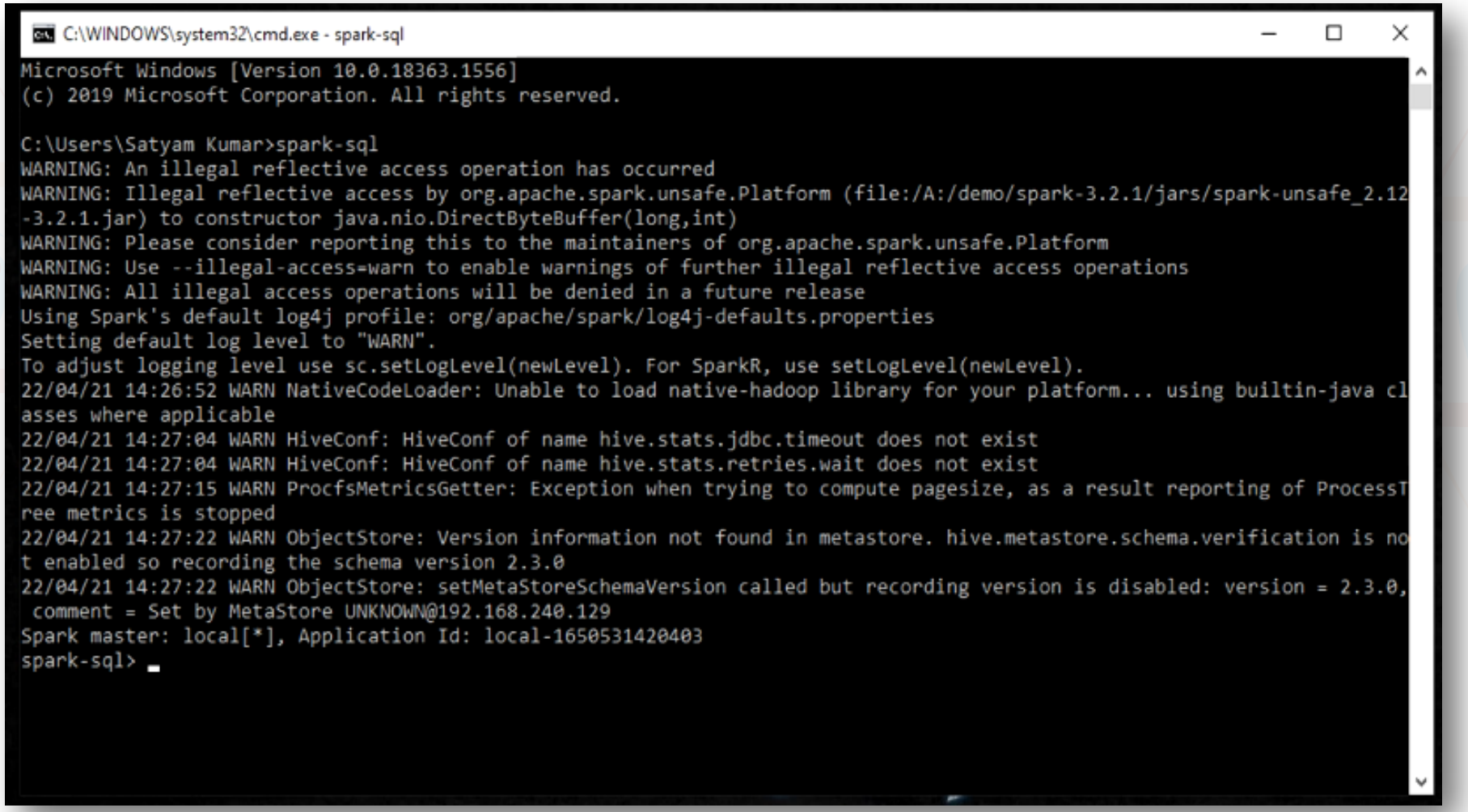
The command-line tools are rarely used on Hadoop Clusters. And we never use them on Databricks clusters.



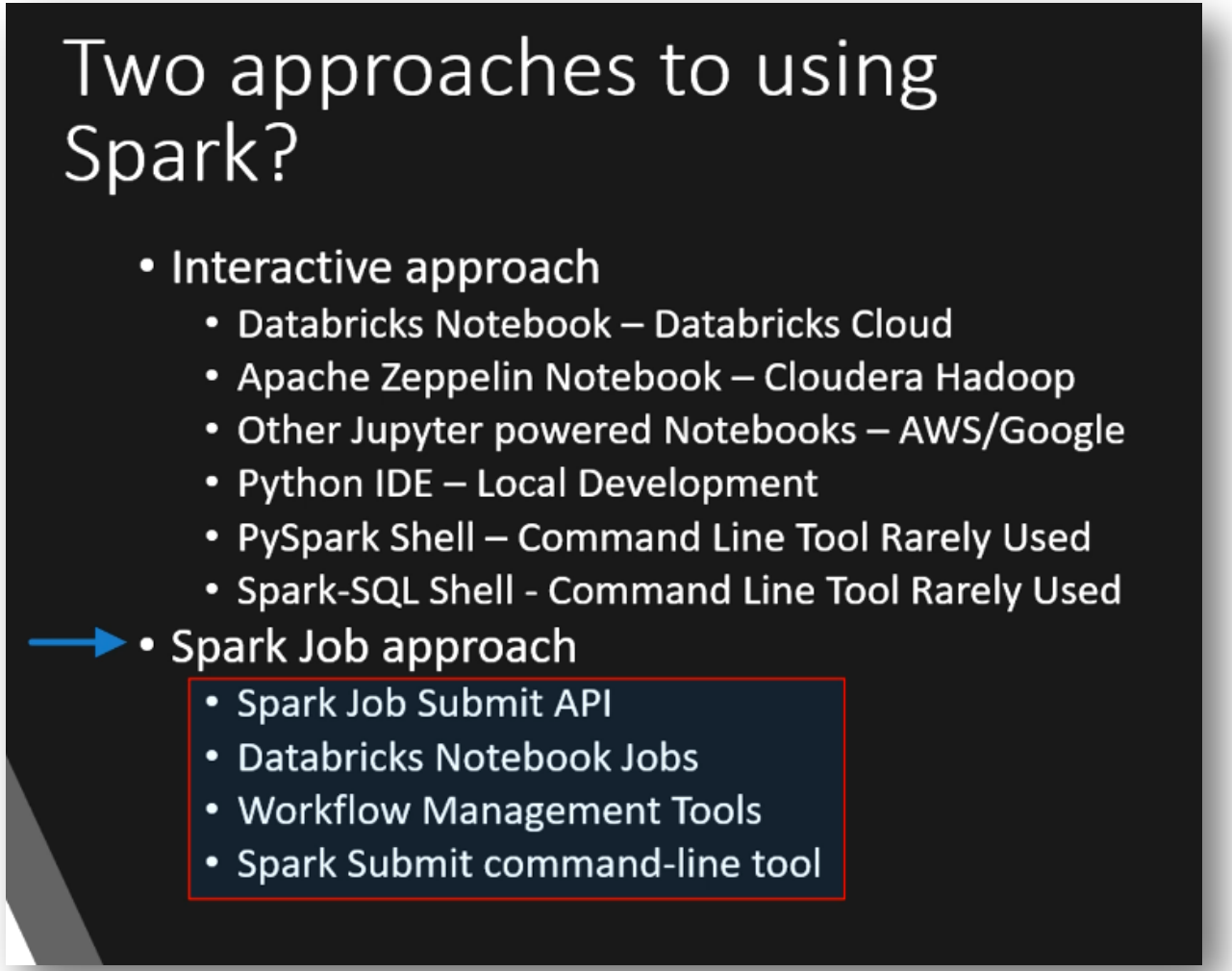
Start your command window and try running the pyspark command as shown below.
A pyspark shell is a command-line tool. You can write some Spark Dataframe code here on the command line and hit enter to run it. It is running on my local machine, so my code will run on the local machine. But if you have a Hadoop cluster, you can configure your command-line tools to run on the cluster. Working with a command-line tool is hard, and we do not use this approach until that is the last option.



We also have a spark-SQL command-line tool.
Start a new command window and run the spark-SQL command as shown below. It will give you a spark-SQL command prompt. You can run spark SQL commands here and see the results. But we do not use this one also. It is pretty hard to work with these command-line tools.



We will package our project files and deploy them on the production cluster. But how do we do that? Well, we run our completed Spark applications on a Spark cluster as a Spark Job or Spark application. We have many ways to run a Spark Job, and these options depend on your Spark environment. There are four standard methods to do that as highlighted in the screenshot below.



Spark Job Approach:

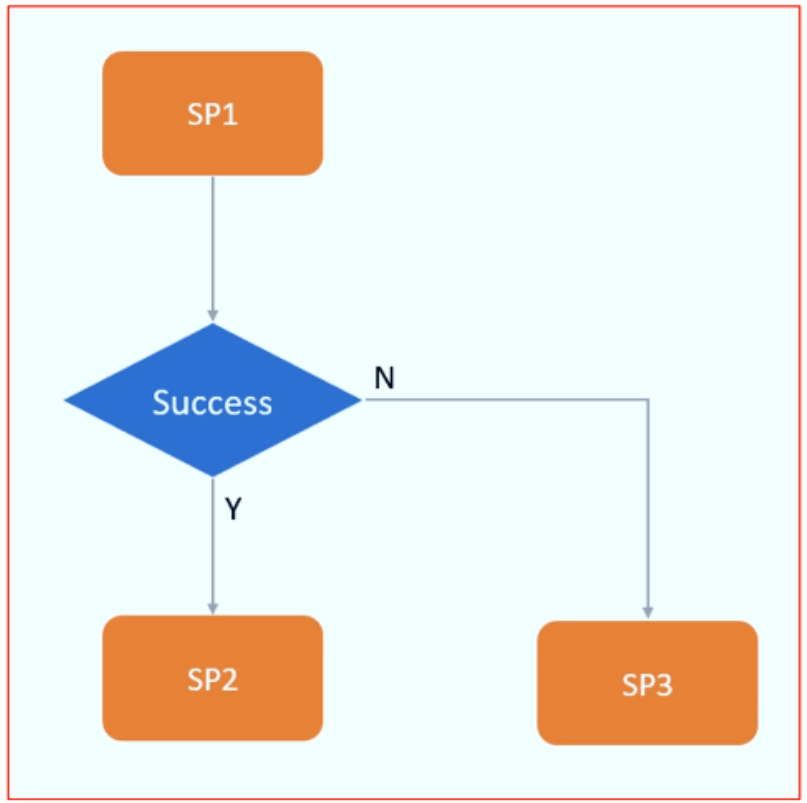
Apache Spark offers an API for submitting an Application to the Spark cluster.
Once submitted, the application runs on the cluster.
However, you are not likely to use APIs. So you can ignore the API option, but it is essential to know that. These APIs are at the core. And other methods internally use the APIs. So Spark offered the API, and many tools were developed to use those APIs.

Then the second most common method is to use Databricks Notebook Jobs.
Databricks offers a user interface to submit your application to the Databricks cluster.
It allows you to submit a Databricks notebook as an application.

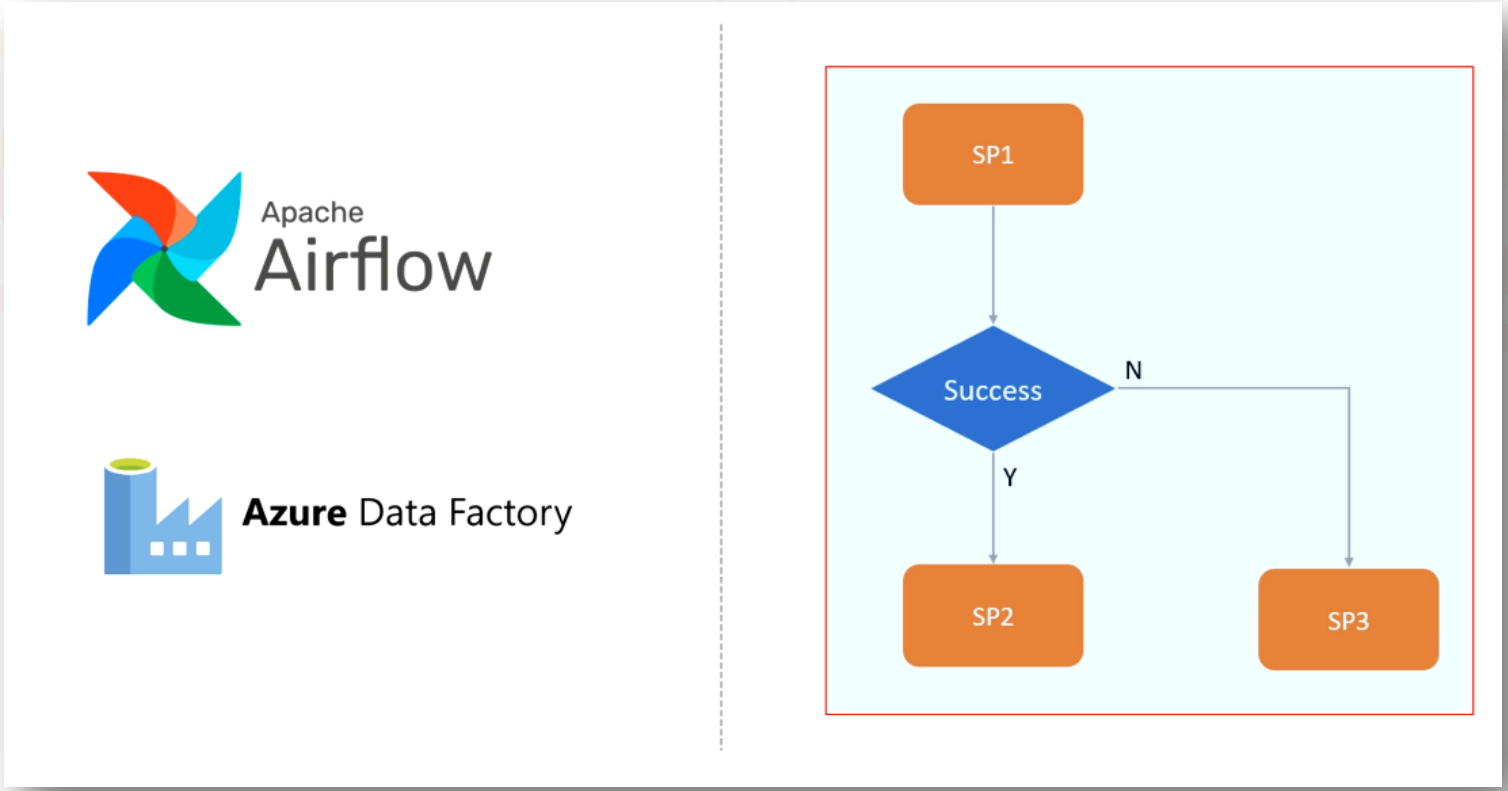
The next option is to use a workflow management tool. We often use workflow management tools when we have more than one Spark application and want to schedule them in a given sequence.

Apache Airflow

Azure Data Factory



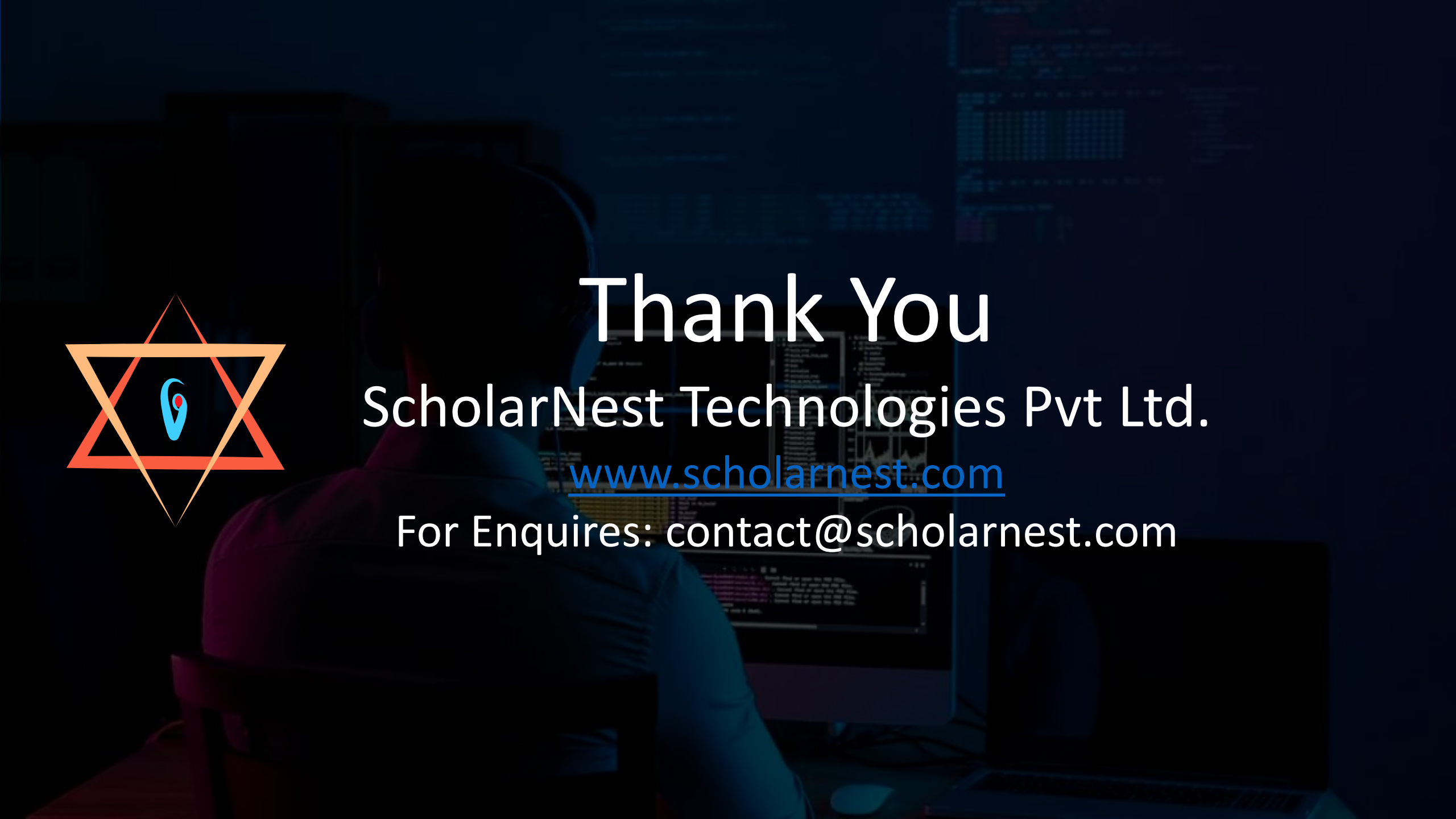
For example, let's assume you have three spark applications (SP1, SP2, and SP3). And you want to create a workflow as highlighted below. In this workflow, you want to run the SP1. Then run the SP2 only when SPI is successfully finished. If SP1 fails, you want to run the SP3. Spark allows you to create applications. But designing workflows is not part of Spark. But we have many tools available to do that.
Apache Airflow is one of the most commonly used tools for doing that.
Cloud platforms also offer you these capabilities. For example, Azure Data Factory allows you to create and manage complex workflows. Behind the scene, these workflow tools will be using the Spark Job submit APIs.



The last option is the spark-submit command-line tool. The spark-submit command-line tool is also popular, and we can use it to submit a packaged Spark application to run on the Spark cluster.
Almost every Spark cluster supports this tool. And that's why this one is the most popular one.

So we know about four methods to deploy and run a spark application on your production cluster.
But when to use what?
We do not use spark-submit APIs. So you can simply discount that from the available options.
We use the spark-submit tool on the Cloudera cluster and Databricks for a single spark application.
That means the spark-submit is an easy-to-use command-line tool, and it is supported by almost all On-Hadoop and On-Cloud clusters. So you can use it everywhere.
But the spark-submit tool does not allow you to create complex workflows and manage interdependencies between multiple spark jobs.
So we do not use them when we have a workflow requirement.

We use other tools such as Apache Airflow and Azure Data Factory for the workflow requirements.
We have many other workflow tools, but your choice depends on your organization.
If your company already implemented a workflow tool that supports spark, you are most likely to use the same.
However, Airflow and Azure Data Factory are most commonly used.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Cluster and Runtime Architecture

Lecture:
Spark Submit and Important Options

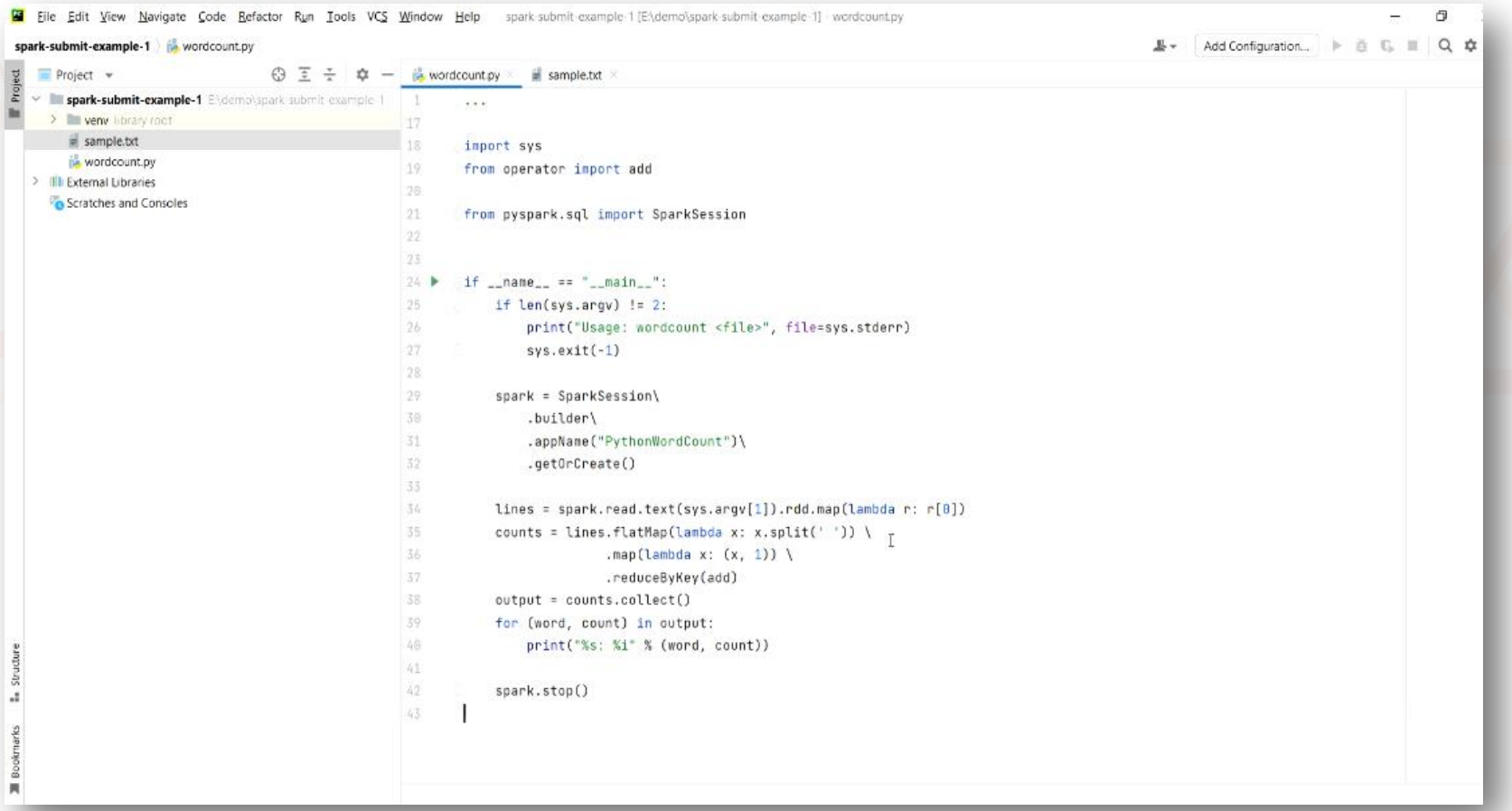
Spark Submit and Important Options

There are two methods to submit a Spark application to the Spark cluster:

1. Single File Spark Application
2. Complex Multi-File Spark Application

Let us see a demo of using the spark-submit command-line tool.

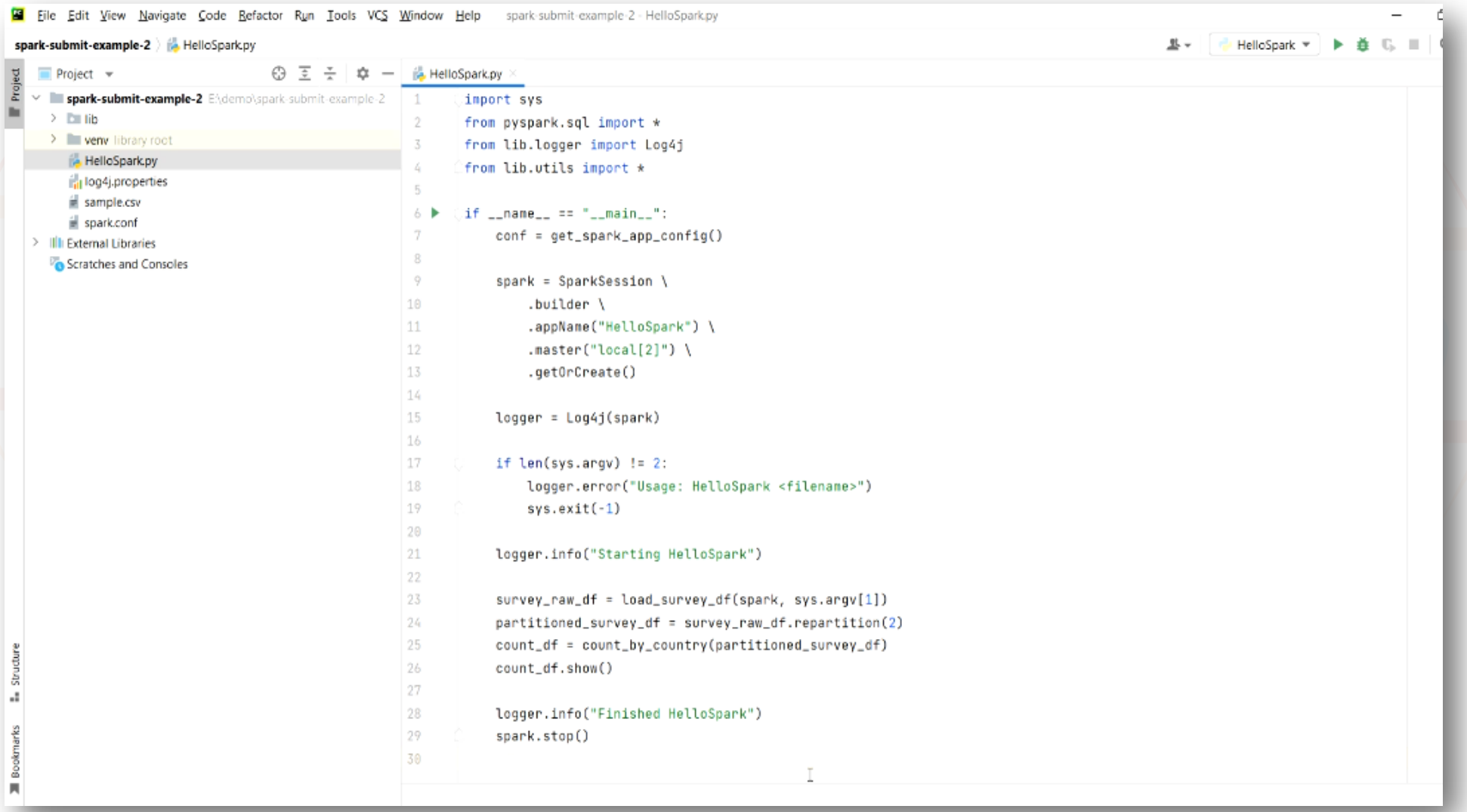
Here is a super simple word count application. **(Ref - spark-submit-example-1)**
We simply took this from Spark examples. You do not need to understand and learn the code at this stage. So let's assume I have a small Spark application that fits into a single Python file.
This program will read a given file and count all unique words in the file. I also have a sample data file. We have a spark application that fits into a single file. Now I want to deploy it on a Spark cluster. We can use the spark-submit tool for doing this, and I will show you how to do it using the spark-submit tool.



But before that, let me show you a different scenario.
Spark projects are not always small enough to fit into a single python file.
A typical Spark project would have a set of the following types of files.

1. Entry point (A Python file with the main method)
2. Libraries (A set of Python files developed by your team)
3. Configuration files (One or more files to configure and customize your application)

Here is another small Spark project. **(Ref - spark-submit-example-2)**
I wrote this example to make sure we follow the structure of a large and complex Spark project. I have a HelloSpark.py file. So this one is my application entry point.



Contents of the project: (Ref - spark-submit-example-2)

I have a main method written here, and my application should start from here. I also have a lib folder. The lib folder has got two python files.

I have only two, but you may have tens of such files in a large project.

I also have two configuration files in my project:

1. spark.conf
2. log4j.properties

These two are configuration files.

I also have a sample.csv data file so I can run it locally. But you can ignore the data file.

It is not part of the application, and I have it for testing my application on the local machine.

A large Spark project will have three types of artifacts:
1. One Main file
2. One or more directories that include other code files.

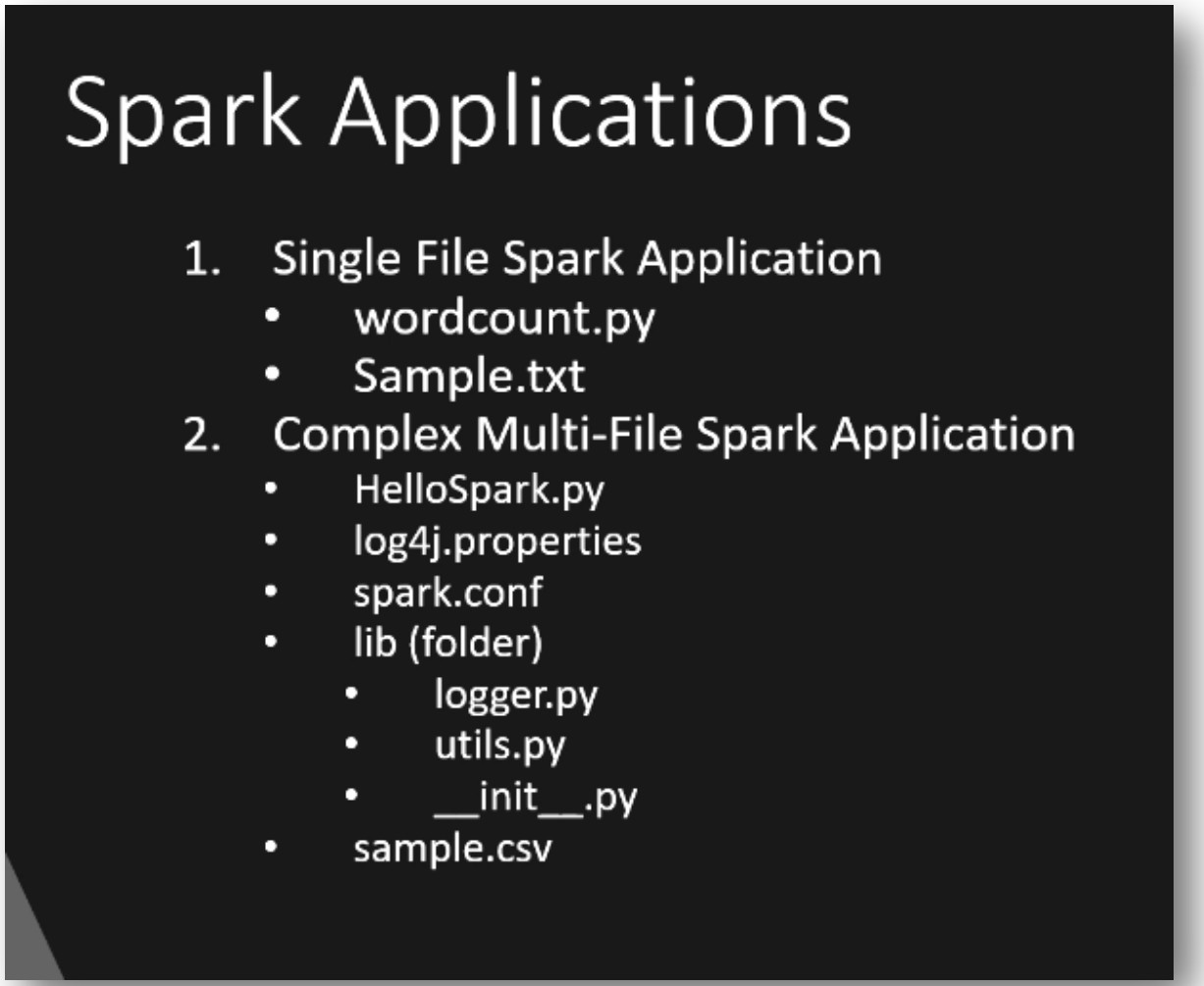
The previous example shows only one directory. But my recent Spark project comes with 20+ such directories and more than 150 source files.

Then you may have a few configuration files.

I have two in my previous project, but your project may have 4-5 such files.

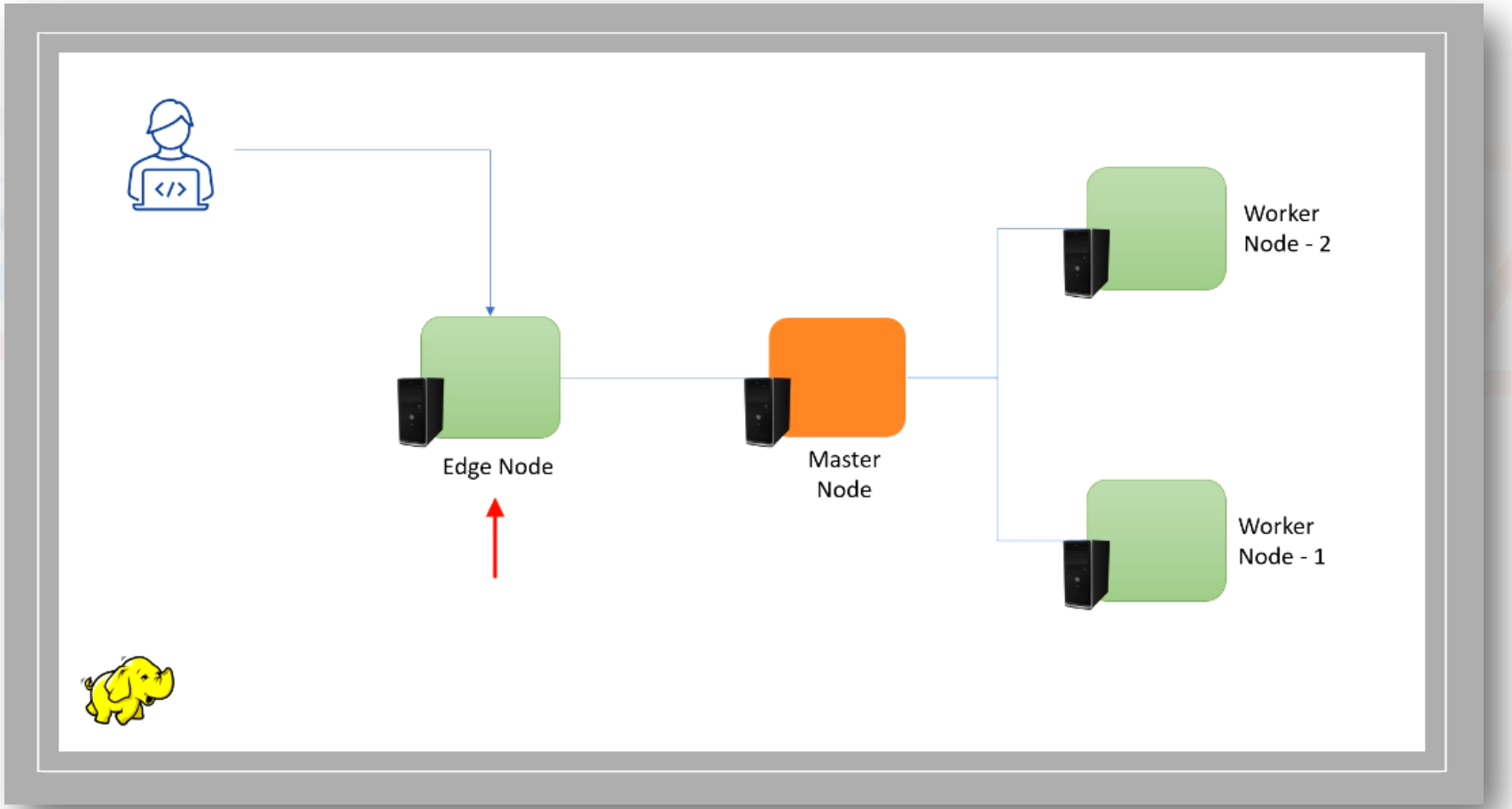
The first one is a single file project, and the second one is a multi-file project. I developed these projects on my local machine, so I have these files. But you could have developed this application on Databricks Cloud.

In that case, you will have a set of notebooks. Deploying and running a notebook project has a different approach.



Next, we want to learn to deploy and run a Pyspark project that we developed on our local machine. But we need a cluster for that.

I have done a four-node Hadoop cluster setup in the Google cloud platform. And it looks like the one shown in image below. So I have one master node and two worker nodes. I have one extra node. This one is not part of the Hadoop cluster. So we do not have any Hadoop services running on this machine. I created this machine to log in to the platform and do a few things from there.

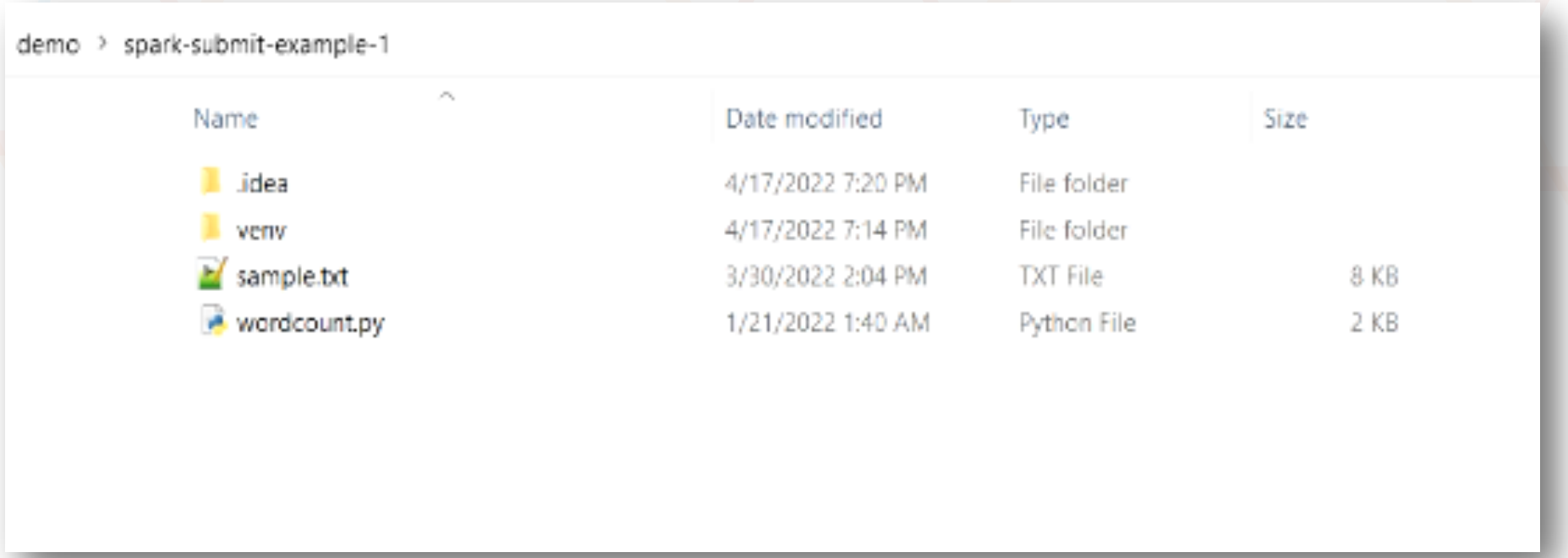


(Reference: Image shown in previous page)
Why do we need an edge node in a Hadoop cluster? We already have one master node and other data nodes. Why do we need a different machine?
Well, it is not safe to directly connect to the master node or the data node. Because we are running Hadoop services on that node.
I do not want any user to log in to those machines and disrupt the cluster services. Only Admin is allowed to log in to those machines.
The developers and operations team should log in to the edge node and work from there.
That's why we created an edge node.
In real-life Hadoop clusters, we create different machines like this one. And we call them to edge nodes. An edge node is a user machine.
We do not have any Hadoop services running on the edge node, but I can run the application on the cluster from the edge node.

So I have a Hadoop cluster. It is a small cluster, but that's more than enough for the demo. I want to deploy my Spark application to this Hadoop cluster. So now we can prepare our project for deployment.

So I have this single file Spark project directory. **(Ref - spark-submit-example-1)**

This application needs a data file and runs a word count on the given data file. I have one sample data file here and a single code file.



For running this application on the cluster, I must do the following things:

1. Upload the code file to the edge node
2. Upload the data file to the storage layer

But you can ask the following questions:

1. Why do I need to upload the program to the edge node?
2. Why upload the data file to the storage layer?

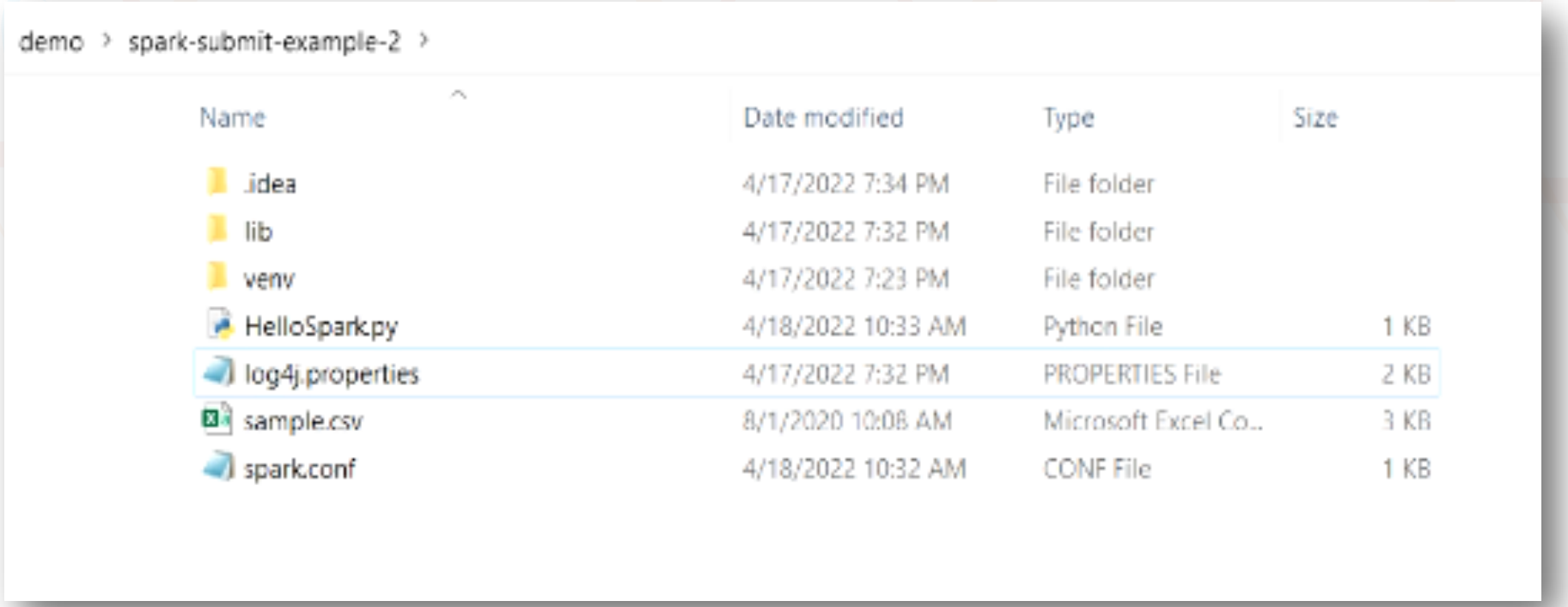
You know the answer to the above questions.
I want to run my application on the cluster.
And I cannot connect to my cluster from my local machine. So I will upload the program file to the edge node and run it from there.

Similarly, Spark is a distributed application, and it reads the data file from the distributed storage layer. In my example, the storage layer is the HDFS. So I must place this data file in the HDFS directory.

Here is my project directory for the large project. **(Ref - spark-submit-example-2)**
Here also, I have two types of files.

1. Application files
2. Data file

I have one main application file, one libs directory, and two configuration files.
And I need to upload all these to the edge node to run my application from there.
Similarly, I must upload my data file to the HDFS location so my Spark application can read the data file from the storage layer.

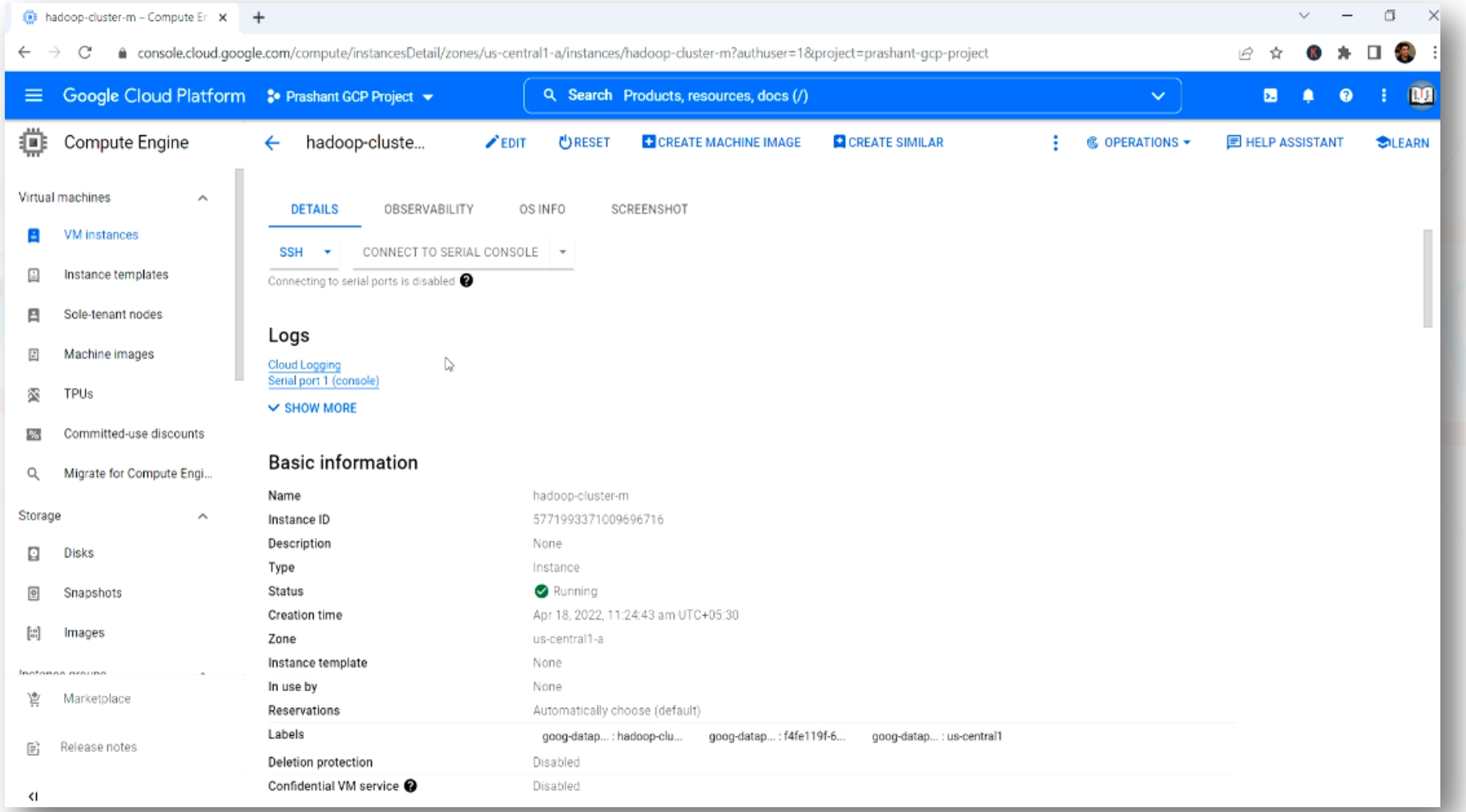


A multi-file project needs some extra preparation. We have three types of files here.

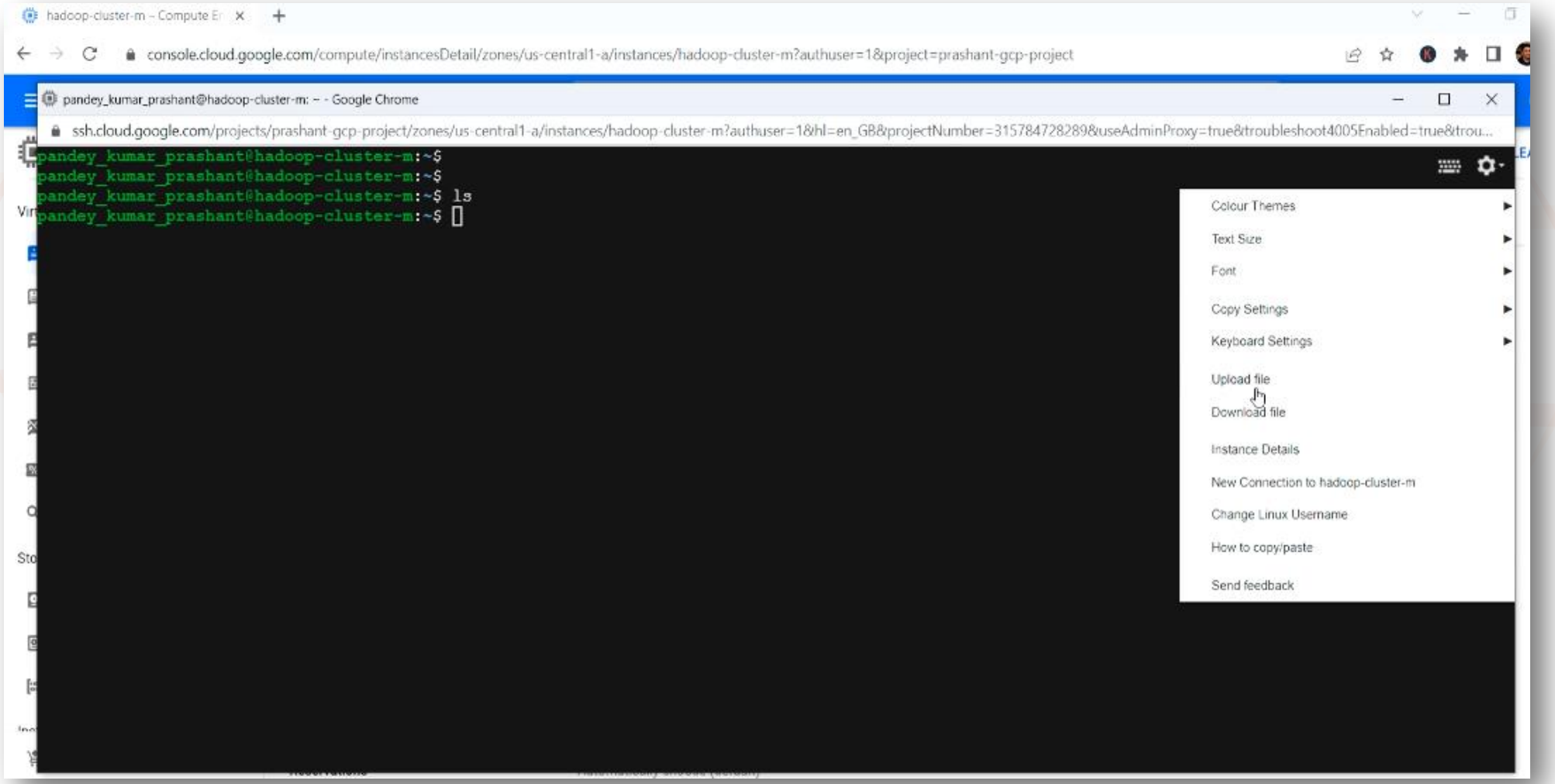
1. Main application file
2. Configuration files
3. Additional application files or libs

The main application file and the configuration files must go as it is.
However, we can compress the additional application files and the libs into a single zip file.
So I will compress the libs directory and make it a zip file. Next, I will upload the main file, two configuration files, and the lib.zip file to the edge node.

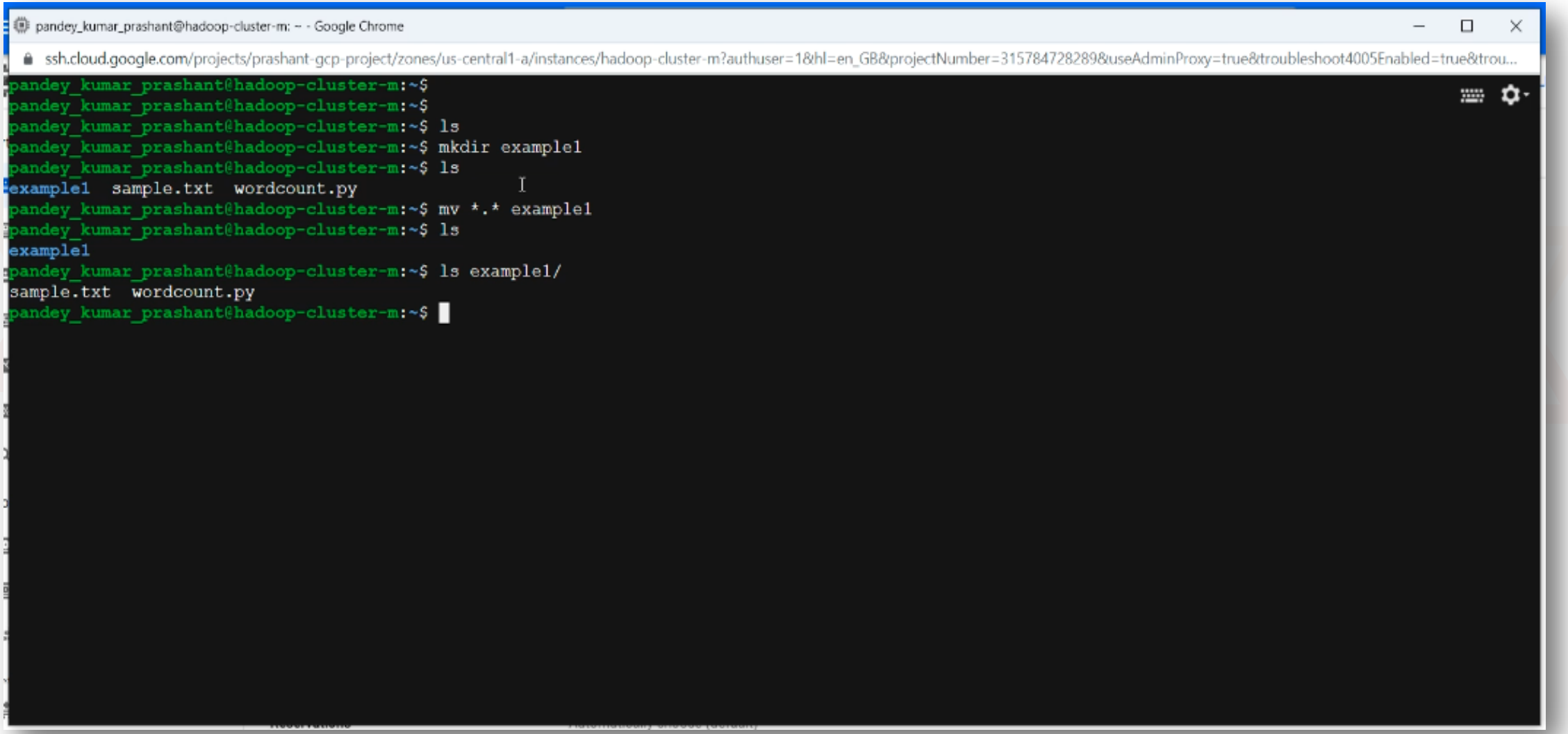
I have a Hadoop cluster, and I created it in the Google Cloud platform for this demo. You do not need to do the cluster setup. Here is the GCP page shown in the image below. This page shows the details of my edge node VM. I can do SSH to this VM and establish a remote connection.



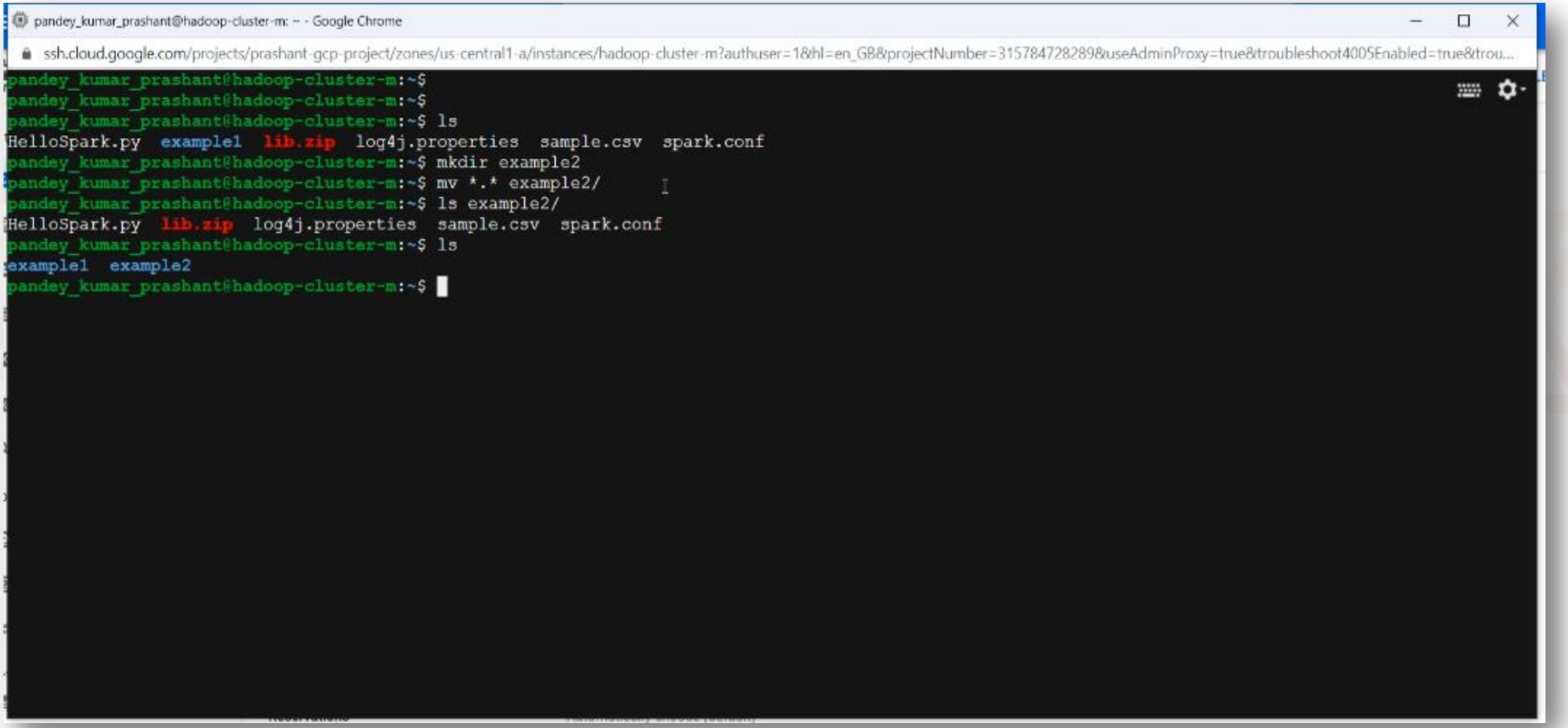
I am now inside the edge node. You can upload your files to the edge node.
I am using the browser-based SSH interface for connecting to the remote machine.
And I am also using the upload menu to upload my files as shown in the image below.



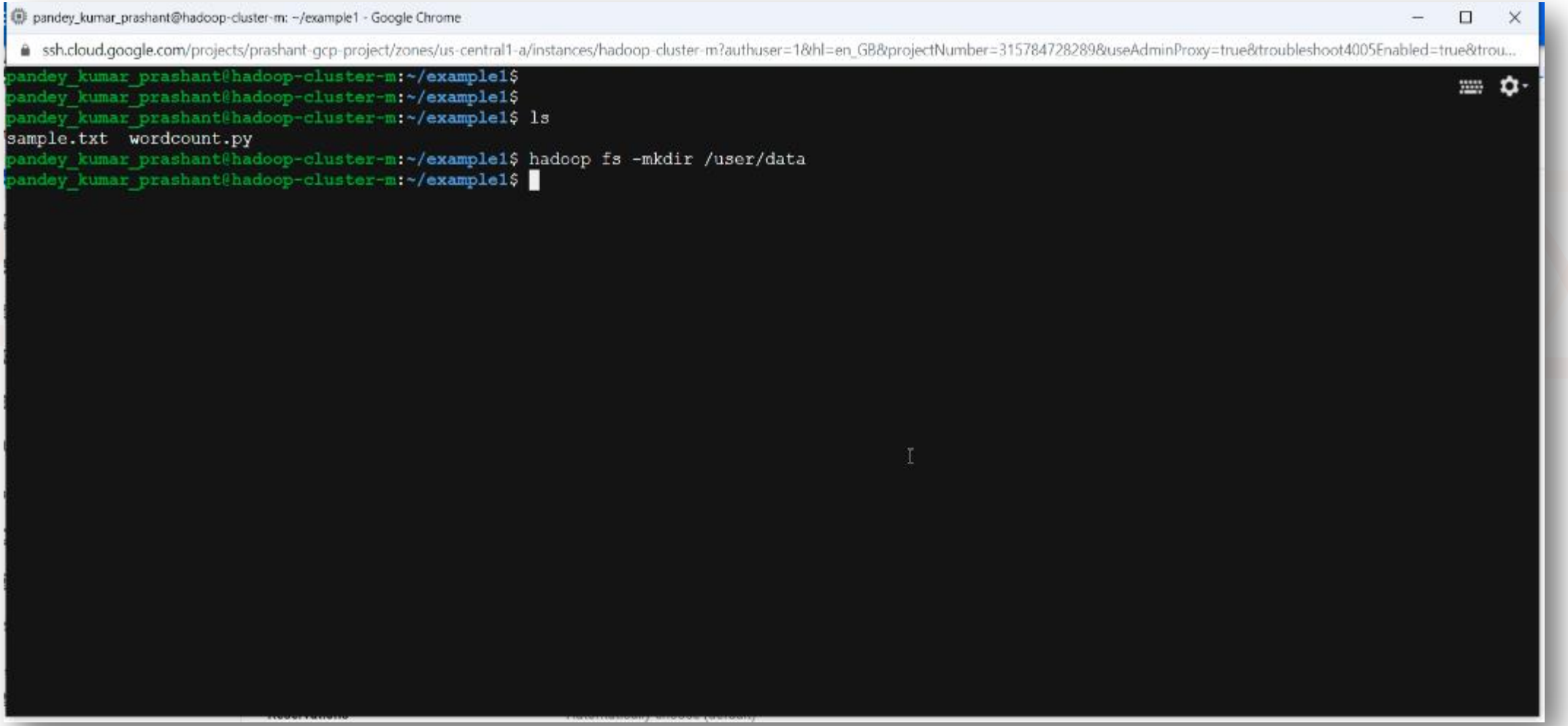
Now, let me create one directory and move all these files into the directory as shown below.



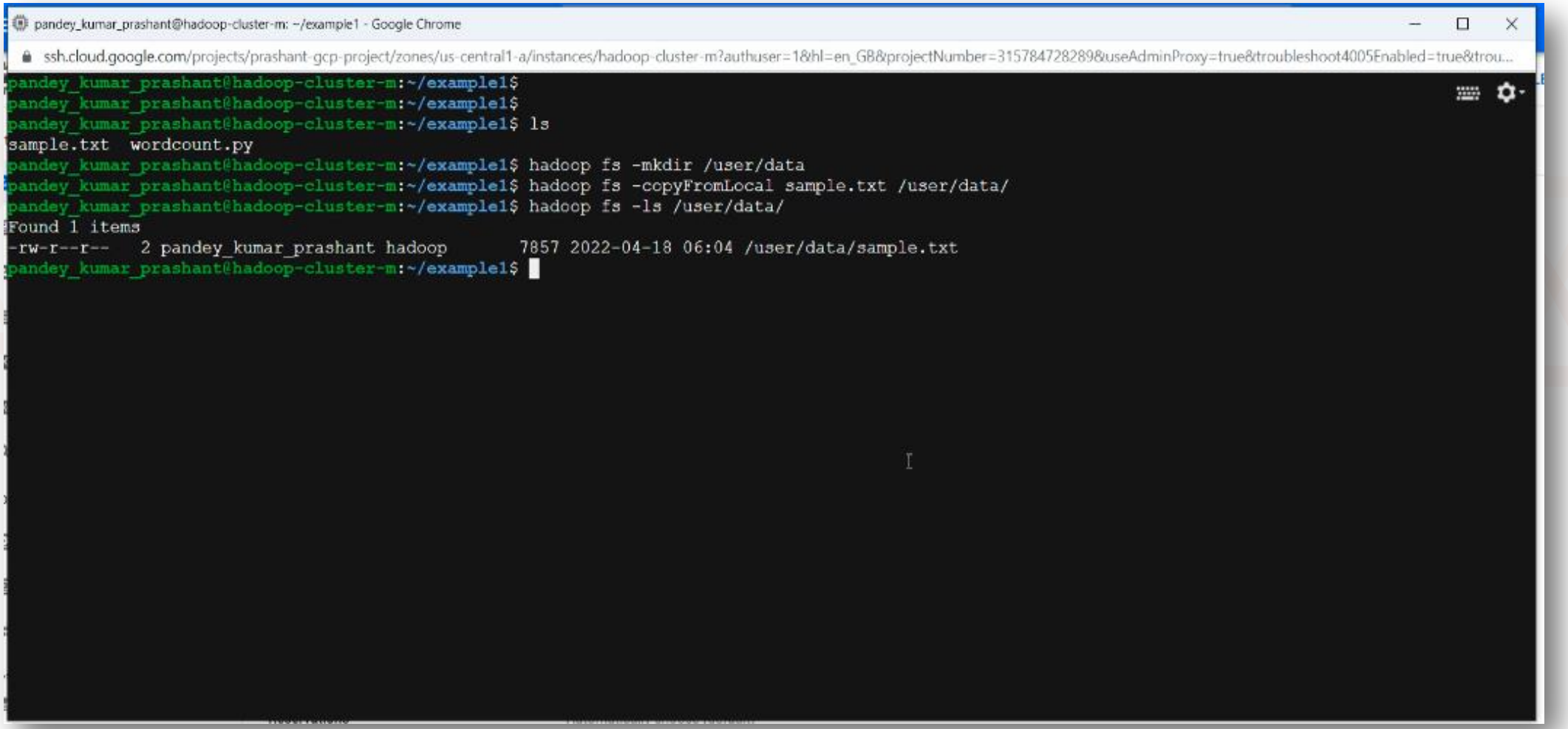
I have all my files for the second example as well. Similar to the previous case, create a separate directory for this example as well and move your files.



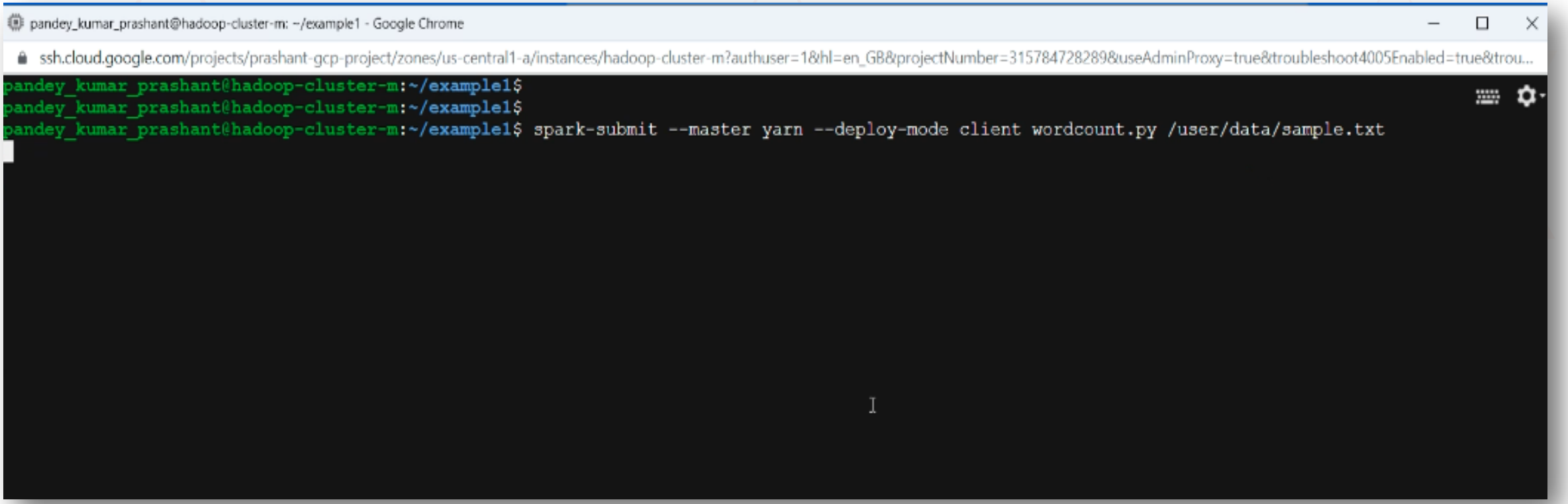
I will work on the first project. So go inside the example1, and you will see that I have only one code file and one data file.
The next step is to copy the data file to a Hadoop directory. So I will create a data directory as shown below.



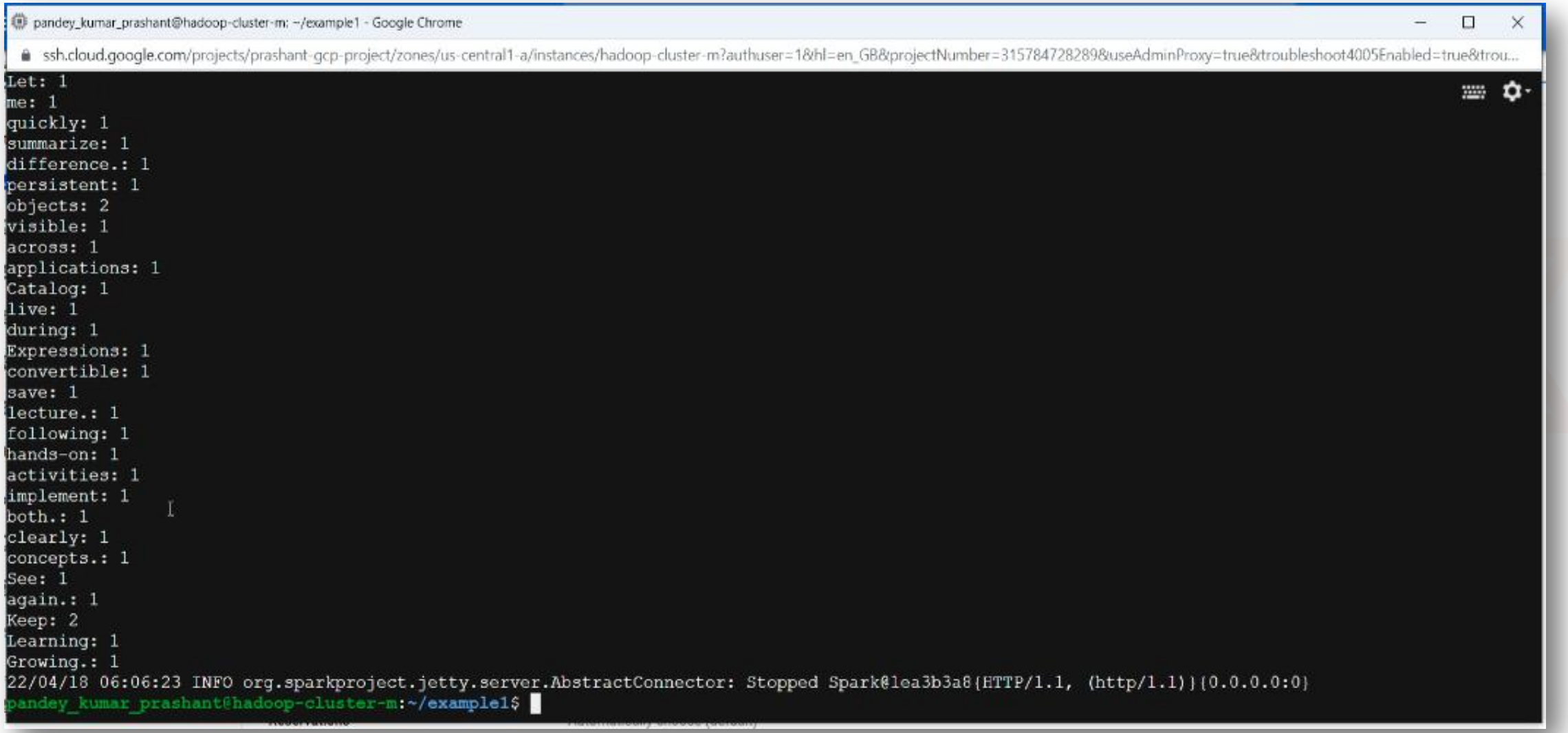
Then I will copy the sample.txt file to the Hadoop directory.
I have used the next command to check if my data file is present there, and I can see the same.



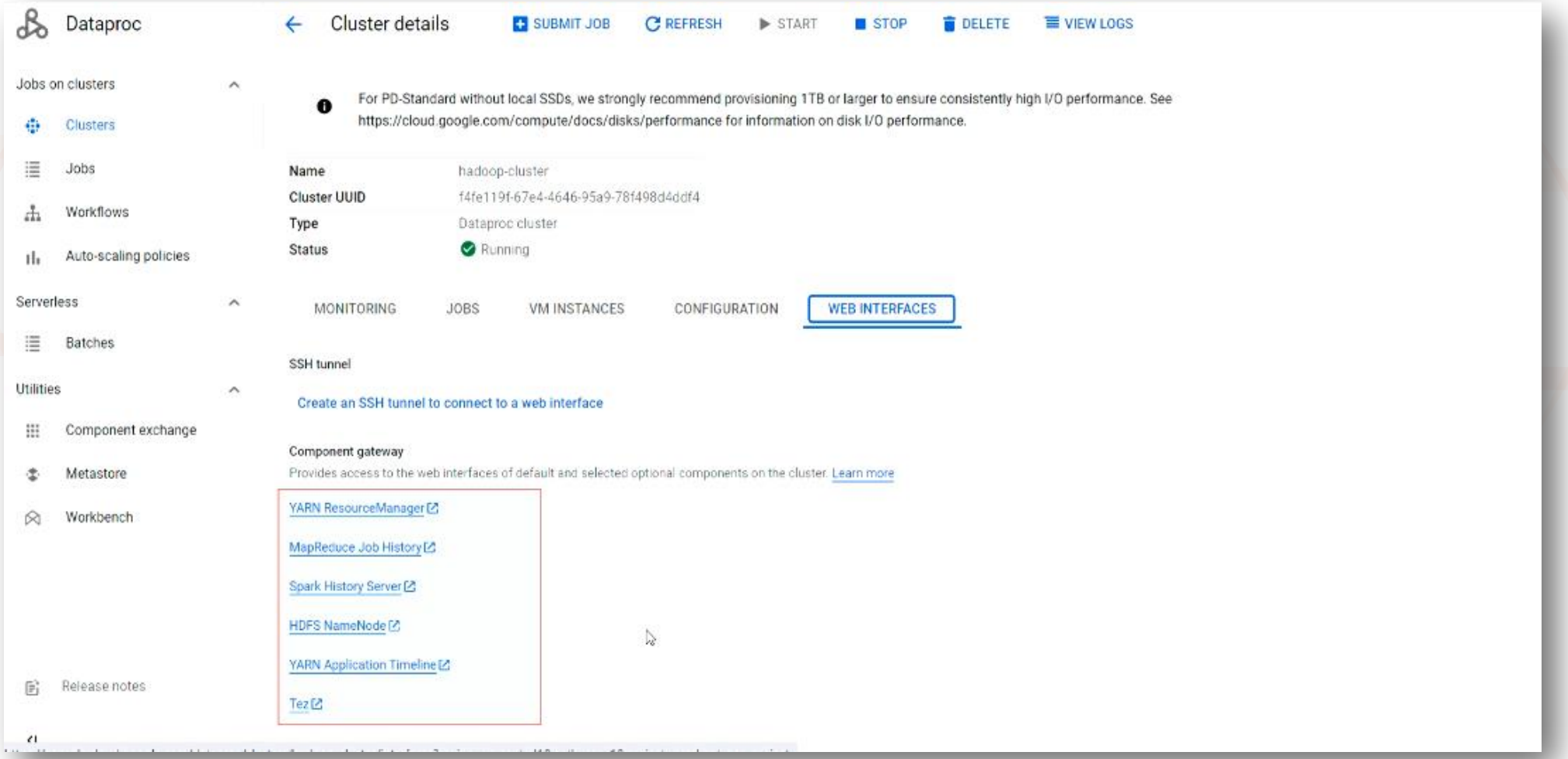
So we are all set to run our word count spark application on a Hadoop Spark cluster. And here is the command shown below for the same.
I am using the spark-submit command. The --master and --deploy-mode are the spark-submit options.
The wordcount.py is my application code file, and the user/data/sample.txt is the command-line argument for my word count application.



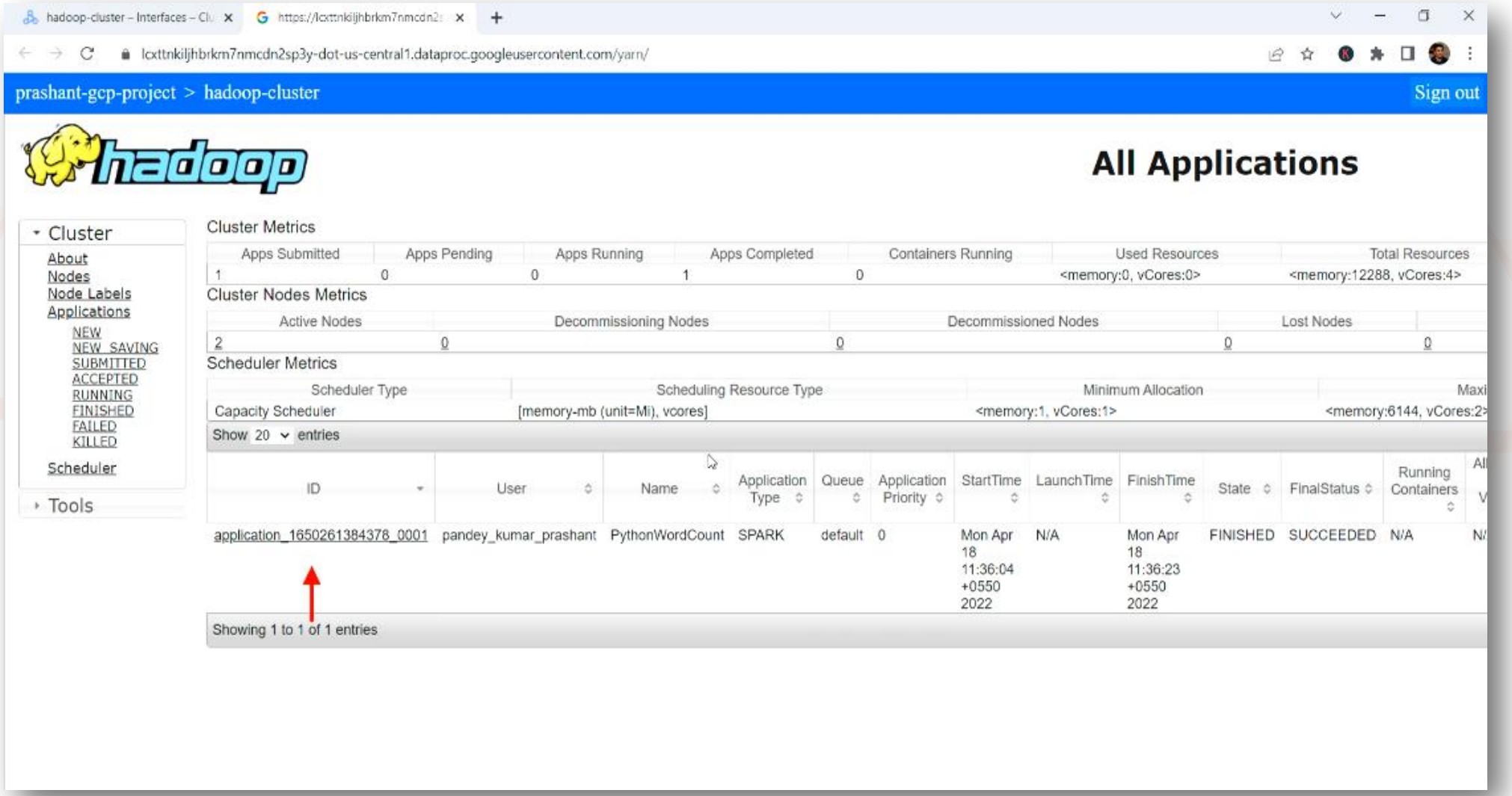
If you run the command, you can see the output. It shows all the unique words and their count.



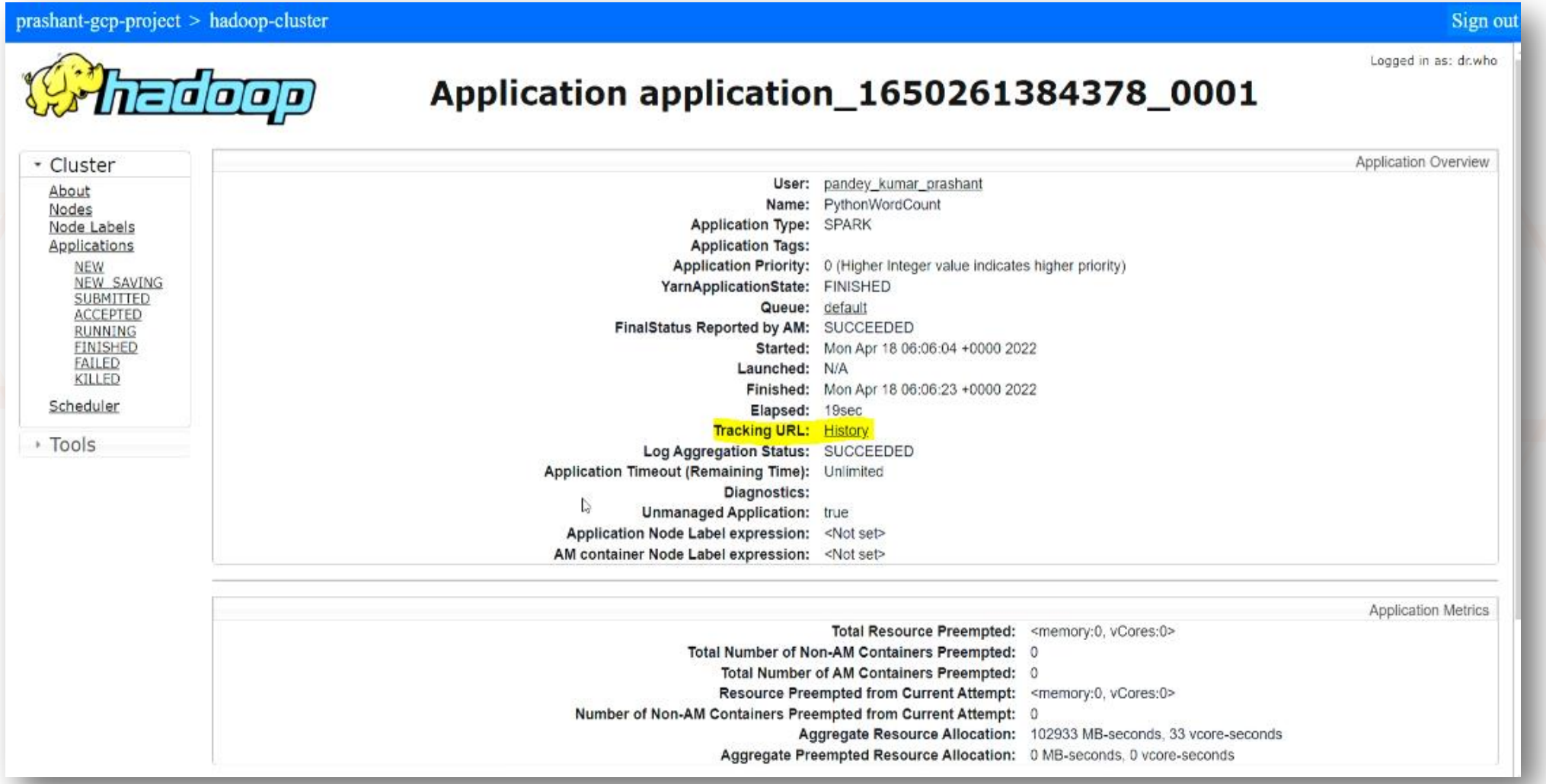
Hadoop cluster comes with many additional facilities to know about your applications.
Here is a list of some tools highlighted below.
The first and the most important one is the YARN resource manager.
You can use the YARN resource manager to see the list of applications.



Here is my application. You can see the application id, user name, application name, application type, and status. Click the application id, and you can get more details.



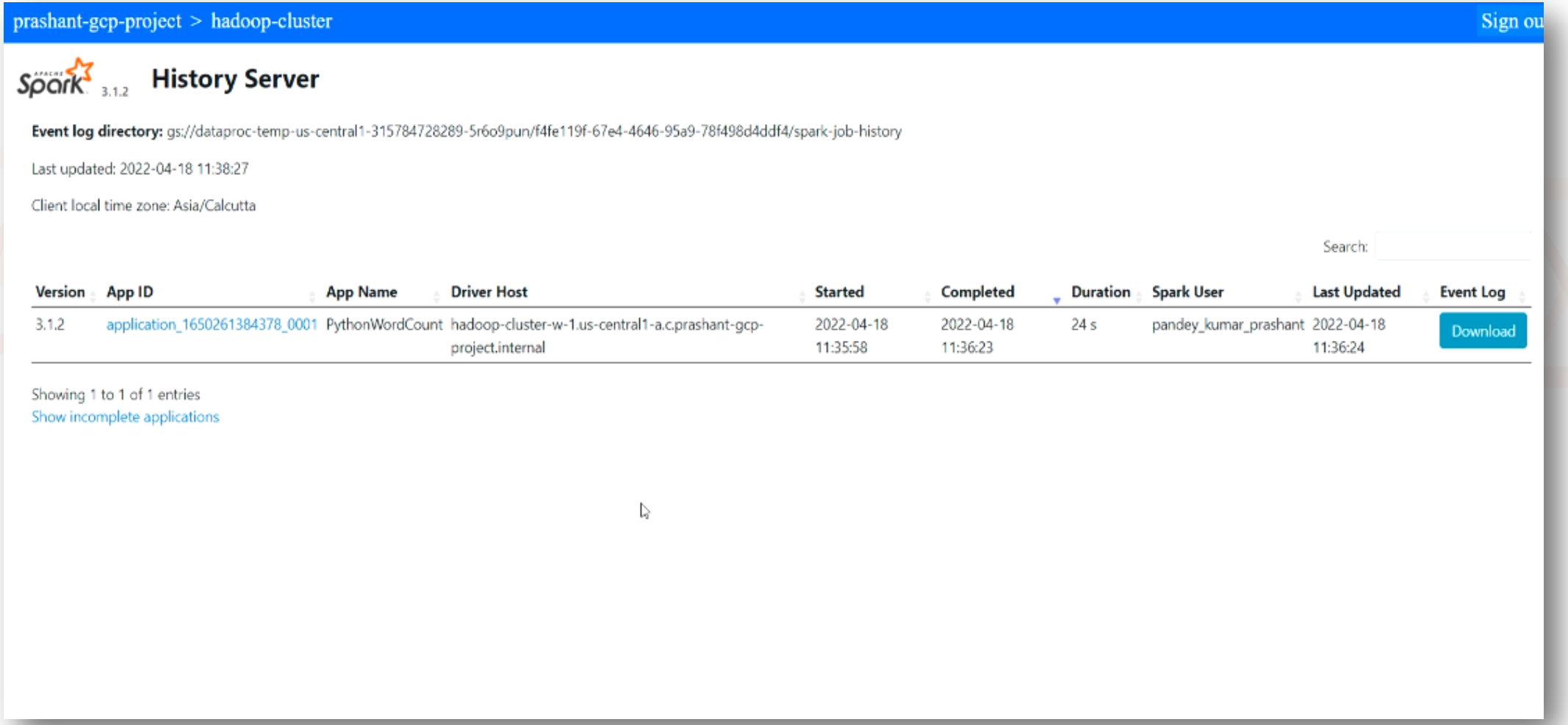
You can see the application log from here. However, Tracking URLs is the most important thing for a Spark application. Click the highlighted link, and it will take you to the Spark history server.



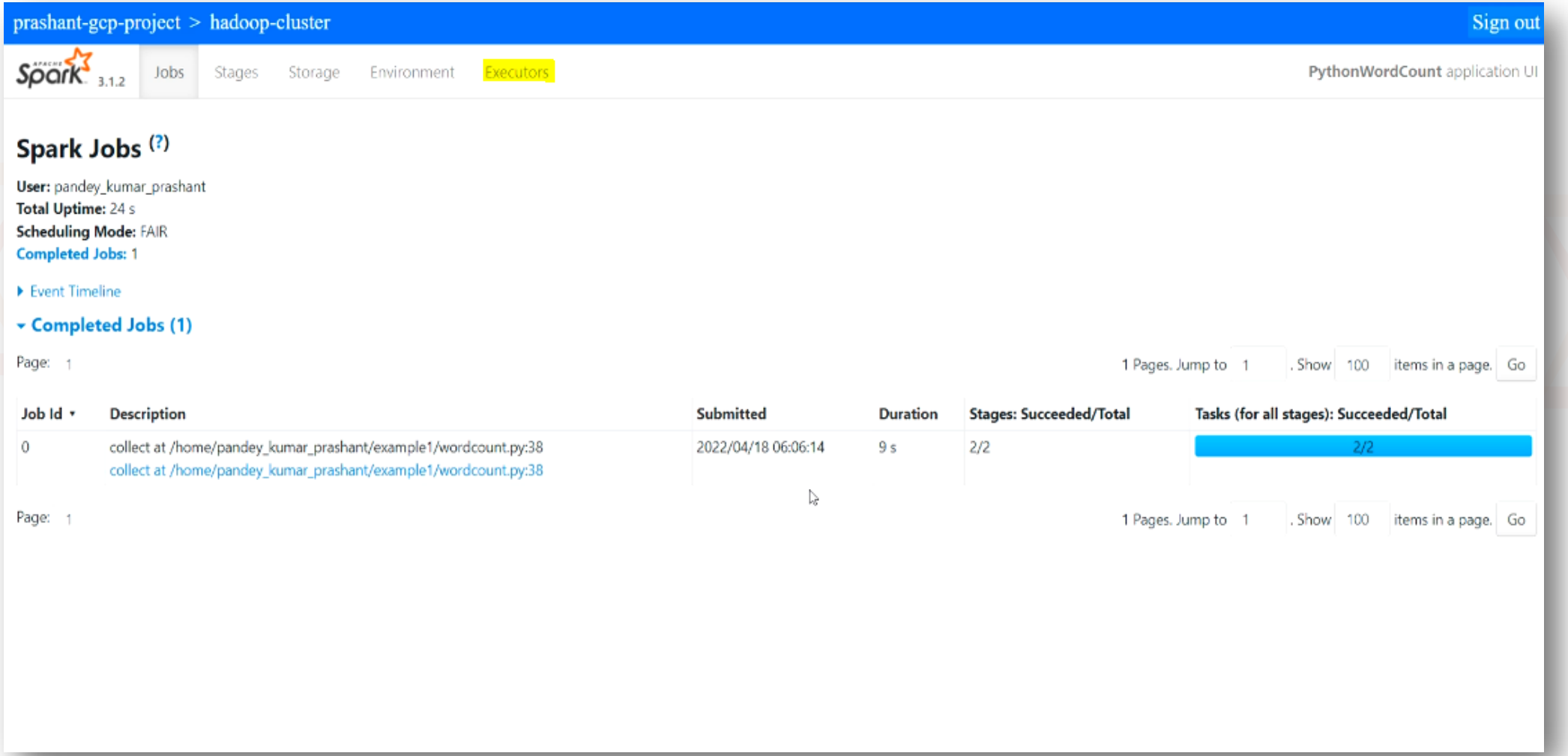
So this one is a Spark tool. The YARN resource manager page was a Hadoop tool. But this one is a Spark tool.

You will see a list of all your Spark applications here. I ran only one application, so I see only one here.

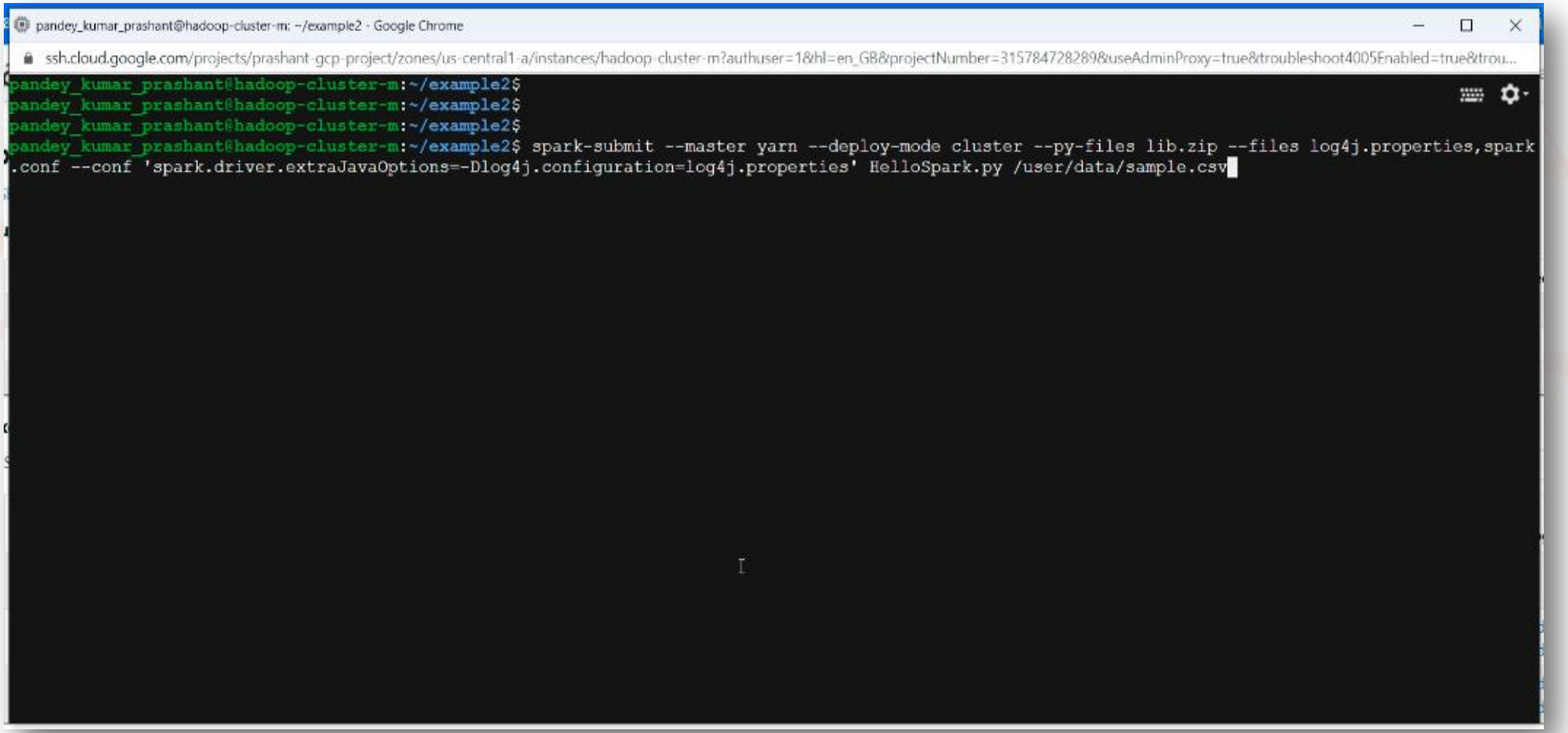
Click the application id, and it will take you to the Spark web UI.



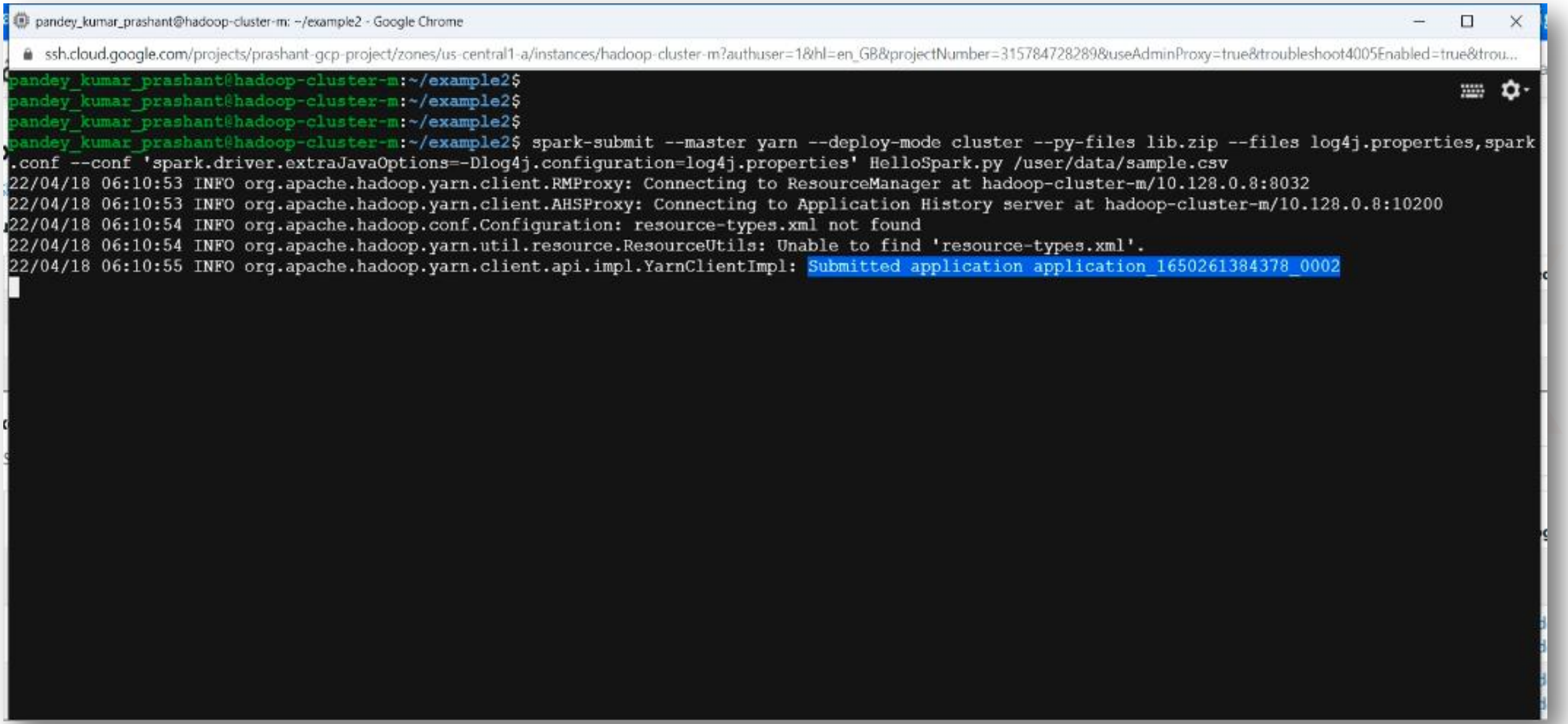
Spark web UI is your go-to place to know everything about your spark application. I will cover the Spark Web UI in more detail. You can see the executor information here.



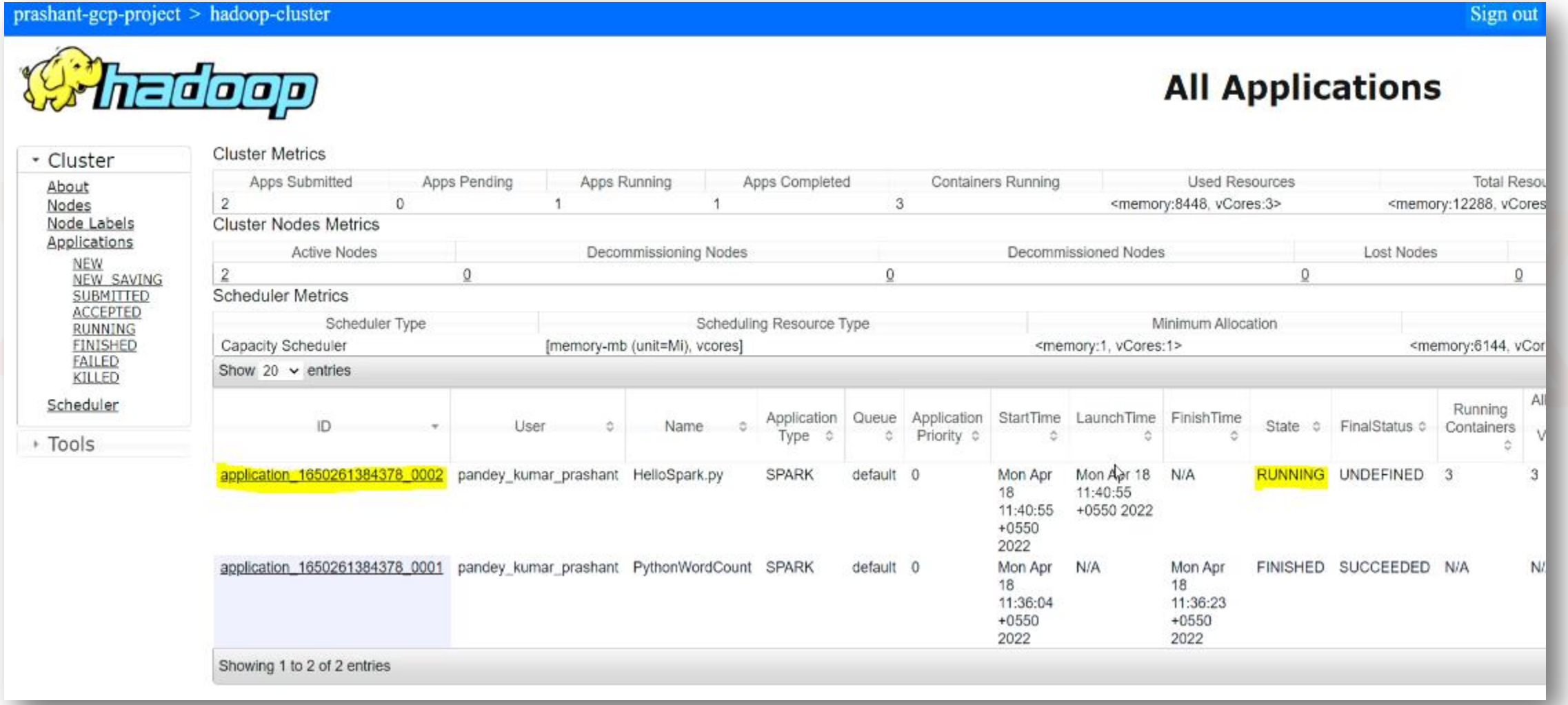
Now go to the second example and run it.
First, let me copy the data file to the HDFS location.
The, we want to submit this multifile project to the Spark cluster and run it. For that we need a spark-submit command as shown below. I am using many spark-submit options here. I have --master, --deploy-mode, --py-files, --files, and finally, the --conf. The last one is the main application file and the command line argument.



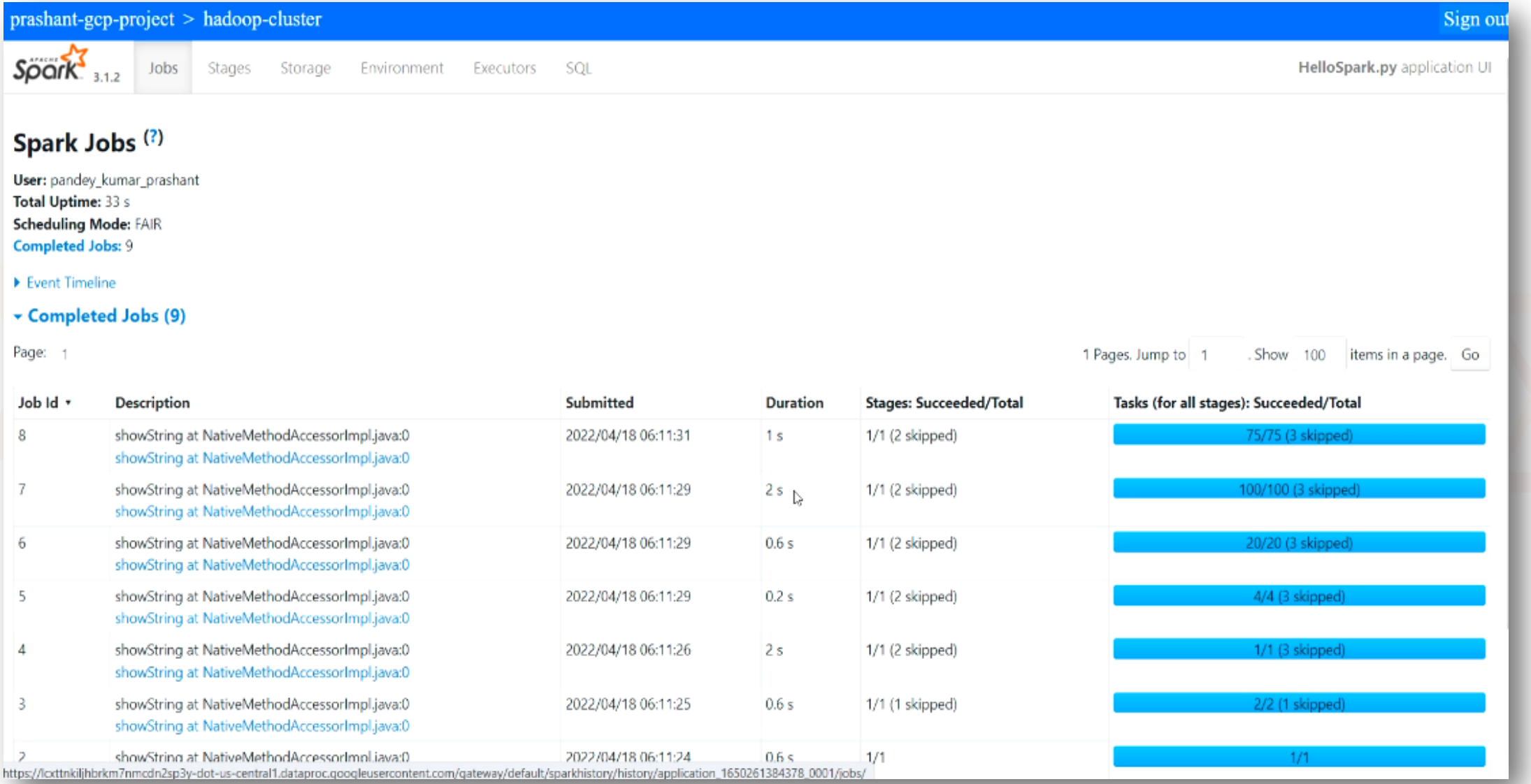
So my application is now running, and here is the application id.
You can go to YARN Resource Manager and open the Spark web UI to see more details.



We see two applications. I want to see the second one highlighted below. It is still running. But that's fine. You can go and check out the Spark UI.

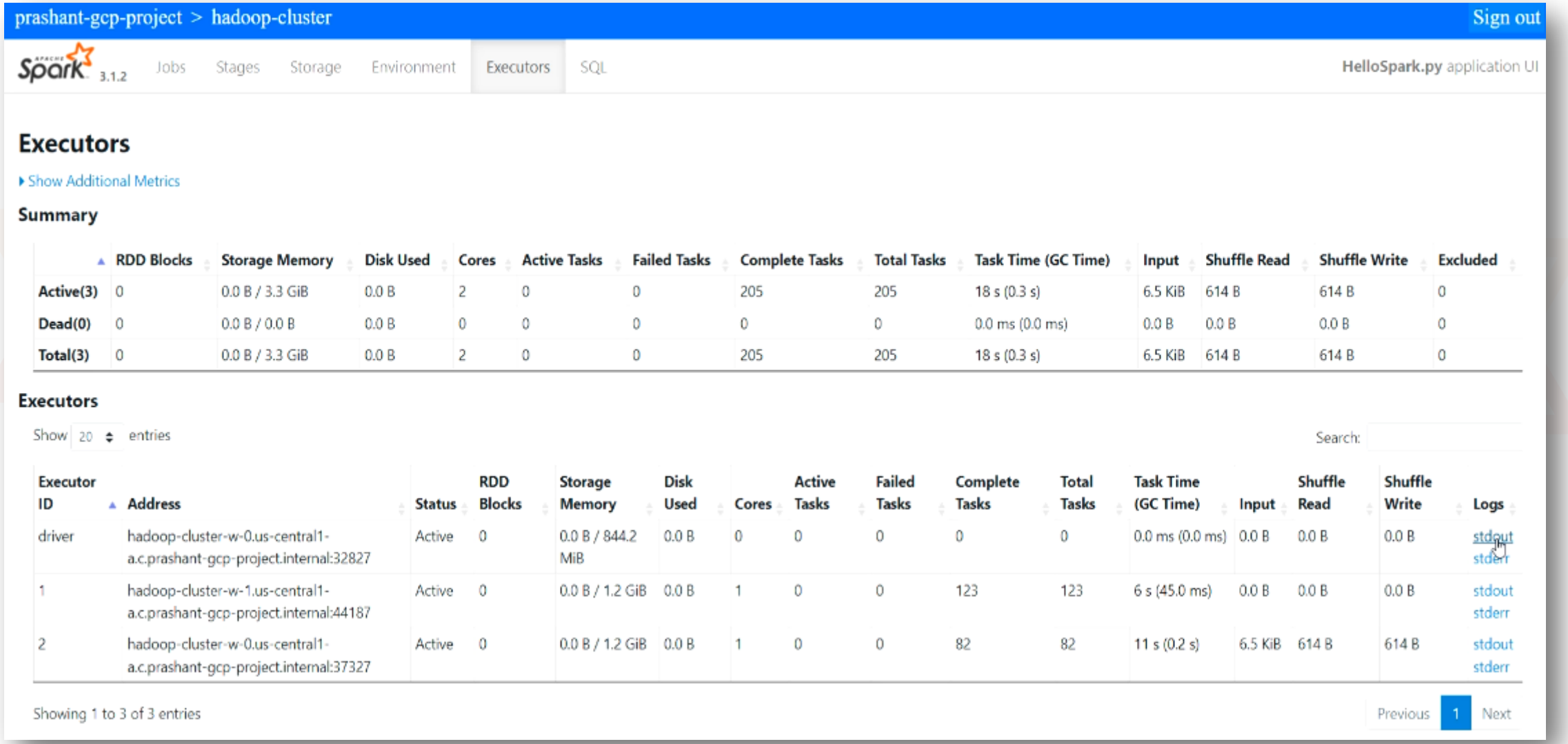


Here is my Spark UI for the second application.



Here is the executor information.

So my application was again executed using one driver and two executors.

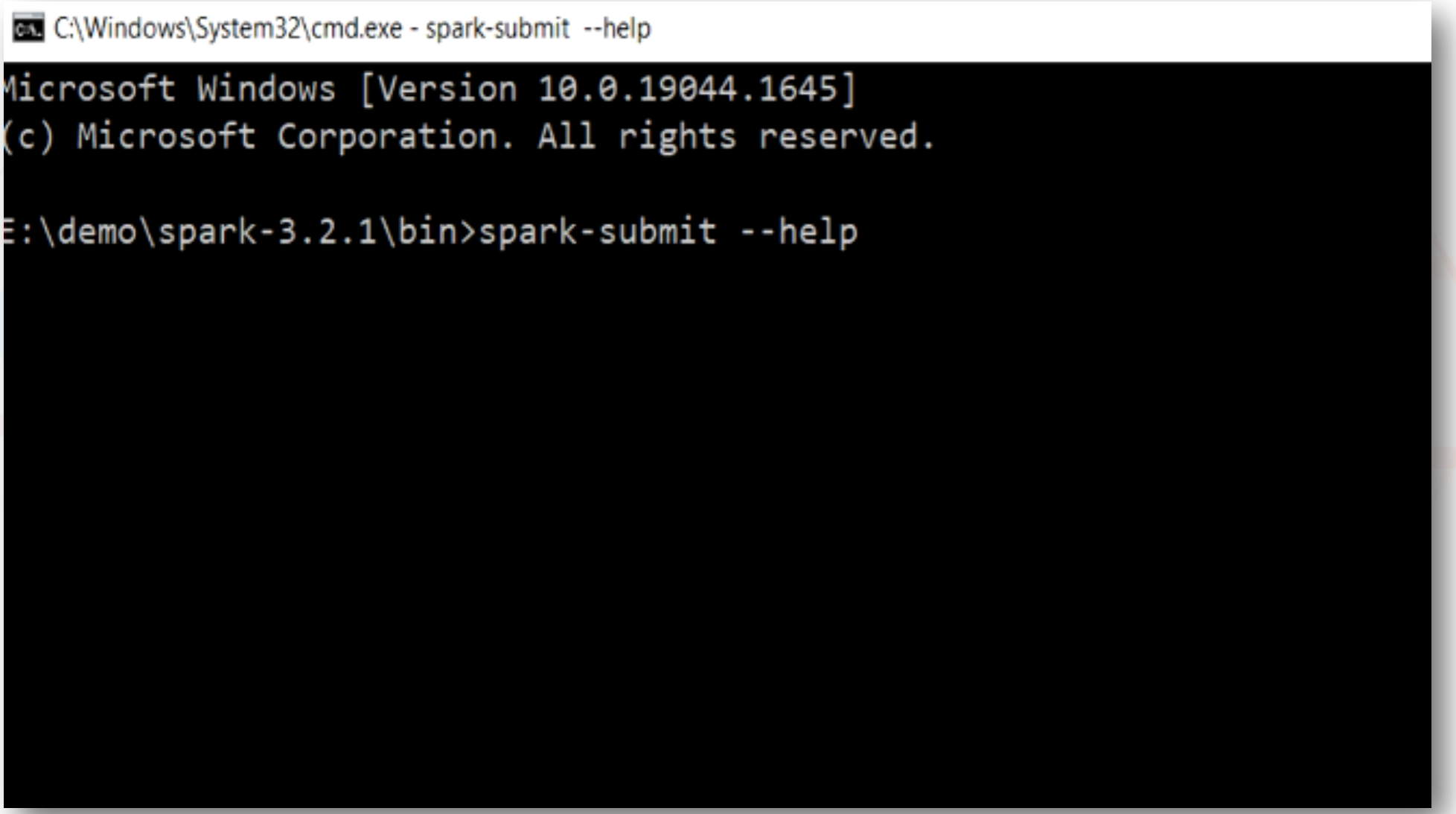


The spark-submit works like a shell command. You can give one or more options to configure the spark-submit behaviour. Finally, you will supply the program file.
If your spark program takes one or two arguments, you can specify the program arguments in the end.
The spark-submit will run your spark application in the Spark cluster.
The spark-submit is a powerful tool, and it offers you many configuration options. We used only a few configurations in this example.

```
spark-submit --master yarn
             --deploy-mode client
              wordcount.py /user/data/sample.txt
```

```
spark-submit --master yarn
             --deploy-mode cluster
             --py-files lib.zip
             --files log4j.properties,spark.conf
             --conf 'spark.driver.extraJavaOptions=-Dlog4j.configuration=log4j.properties'
              HelloSpark.py /user/data/sample.csv
```


You can use the spark-submit –help command to see the complete list of configurations.



I classify the available options into the following groups for better understanding.

```
General options
------------------------------------------------------------------------------
  --master MASTER_URL         spark://host:port, mesos://host:port, yarn,
                              k8s://https://host:port, or local (Default: local[*]).
  --deploy-mode DEPLOY_MODE   Whether to launch the driver program locally ("client") or
                              on one of the worker machines inside the cluster ("cluster")
                              (Default: client).
  --name NAME                 A name of your application.
                             search for the maven coordinates given with --packages.
  --py-files PY_FILES         Comma-separated list of .zip, .egg, or .py files to place
                              on the PYTHONPATH for Python apps.
  --files FILES               Comma-separated list of files to be placed in the working
                              directory of each executor. File paths of these files
                              in executors can be accessed via SparkFiles.get(fileName).
  --conf, -c PROP=VALUE       Arbitrary Spark configuration property.
```



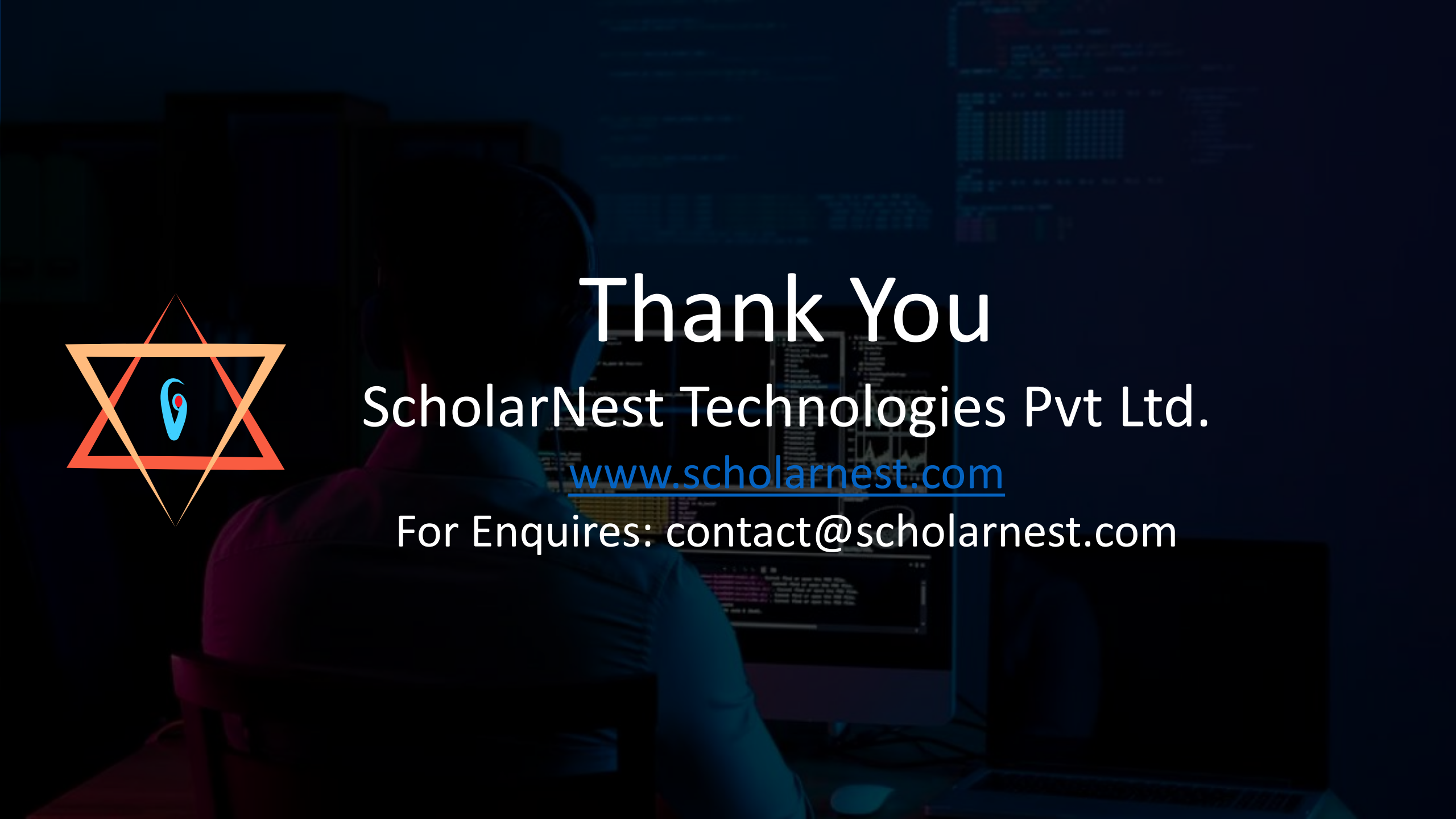
```
Dependency Management Options
------------------------------------------------------------------------------
  --jars JARS                 Comma-separated list of jars to include on the driver
                              and executor classpaths.
  --packages                  Comma-separated list of maven coordinates of jars to include
                              on the driver and executor classpaths. Will search the local
                              maven repo, then maven central and any additional remote
                              repositories given by --repositories. The format for the
                              coordinates should be groupId:artifactId:version.
  --repositories              Comma-separated list of additional remote repositories to
```



```
Resource Allocation Options
--------------------------------------------------------------------------------
  --driver-memory MEM         Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
  --executor-memory MEM       Memory per executor (e.g. 1000M, 2G) (Default: 1G).
  --driver-cores NUM          Number of cores used by the driver, only in cluster mode
                              (Default: 1).
  --executor-cores NUM        Number of cores used by each executor. (Default: 1 in
                              YARN and K8S modes, or all available cores on the worker
                              in standalone mode).
  --num-executors NUM         Number of executors to launch (Default: 2).
                              If dynamic allocation is enabled, the initial number of
                              executors will be at least NUM.
```



```
Security Options
-----------------------------------------------------------------------------
  --principal PRINCIPAL         Principal to be used to login to KDC.
  --keytab KEYTAB               The full path to the file that contains the keytab for the
                                principal specified above.
```

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

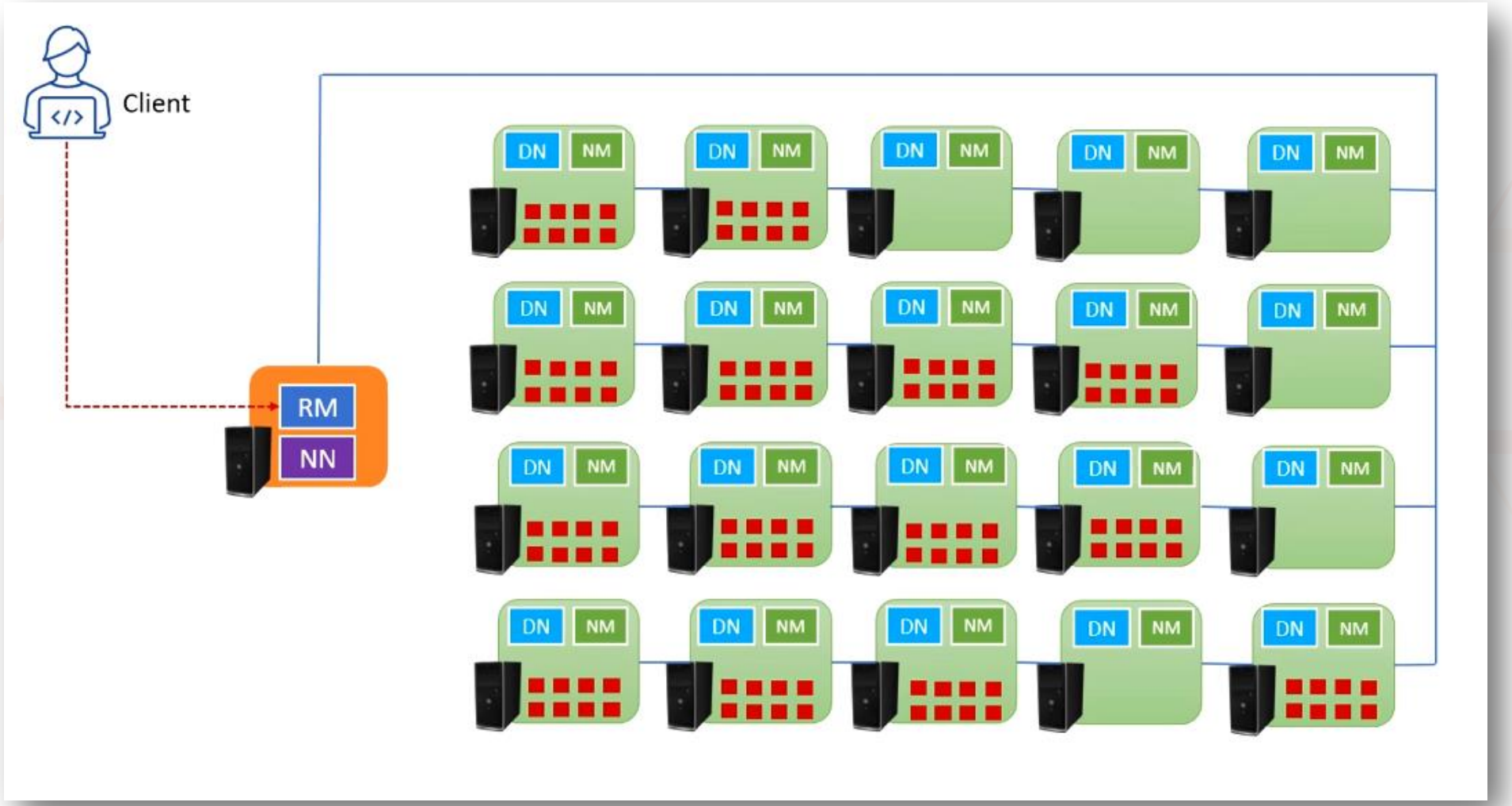
Databricks Spark Certification and Beyond

Module:
Spark Cluster and Runtime Architecture

Lecture:
Spark Deployment Modes

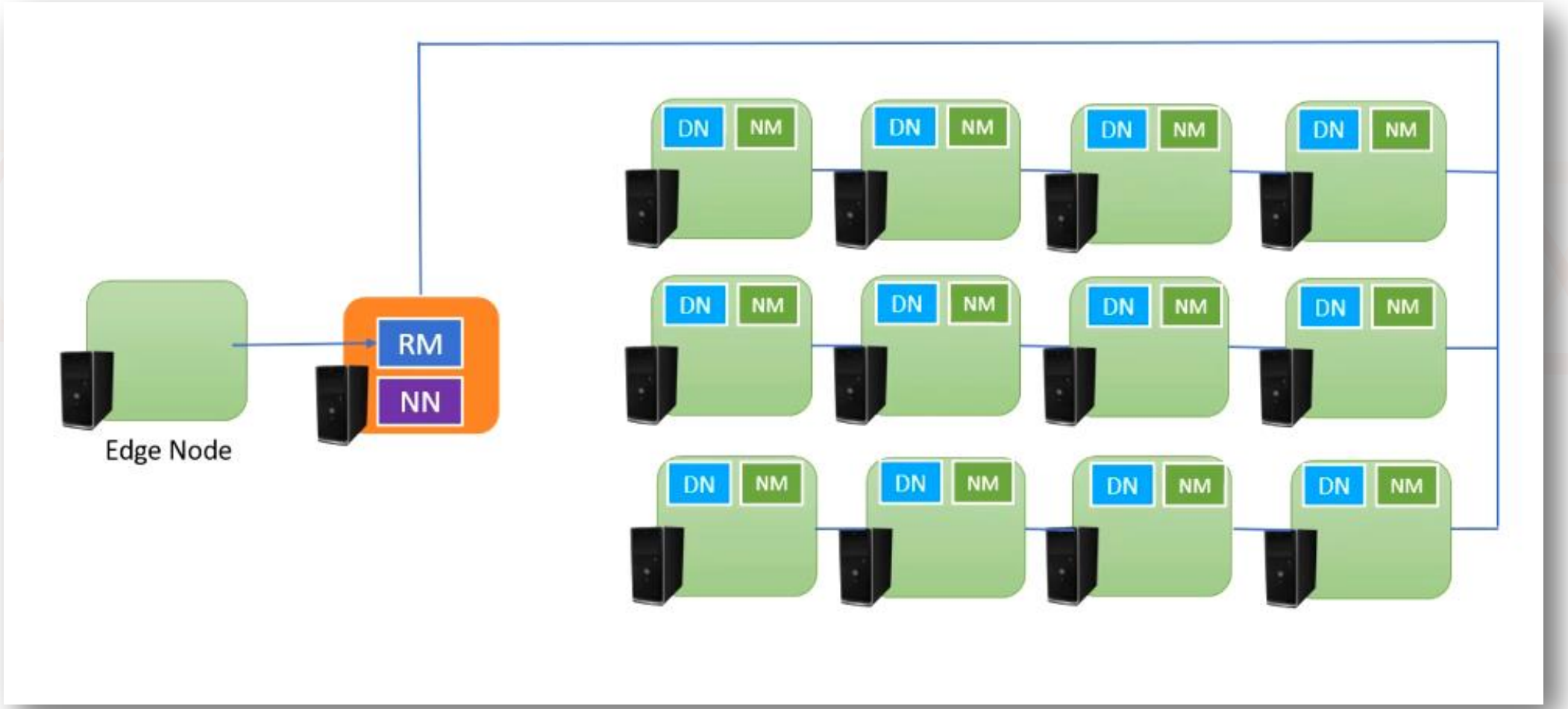
Spark Deployment Modes

I hope you remember this diagram. I explained what happens when you submit your map/reduce application to the YARN resource manager. We learned the process of running a Hadoop Map/reduce program.

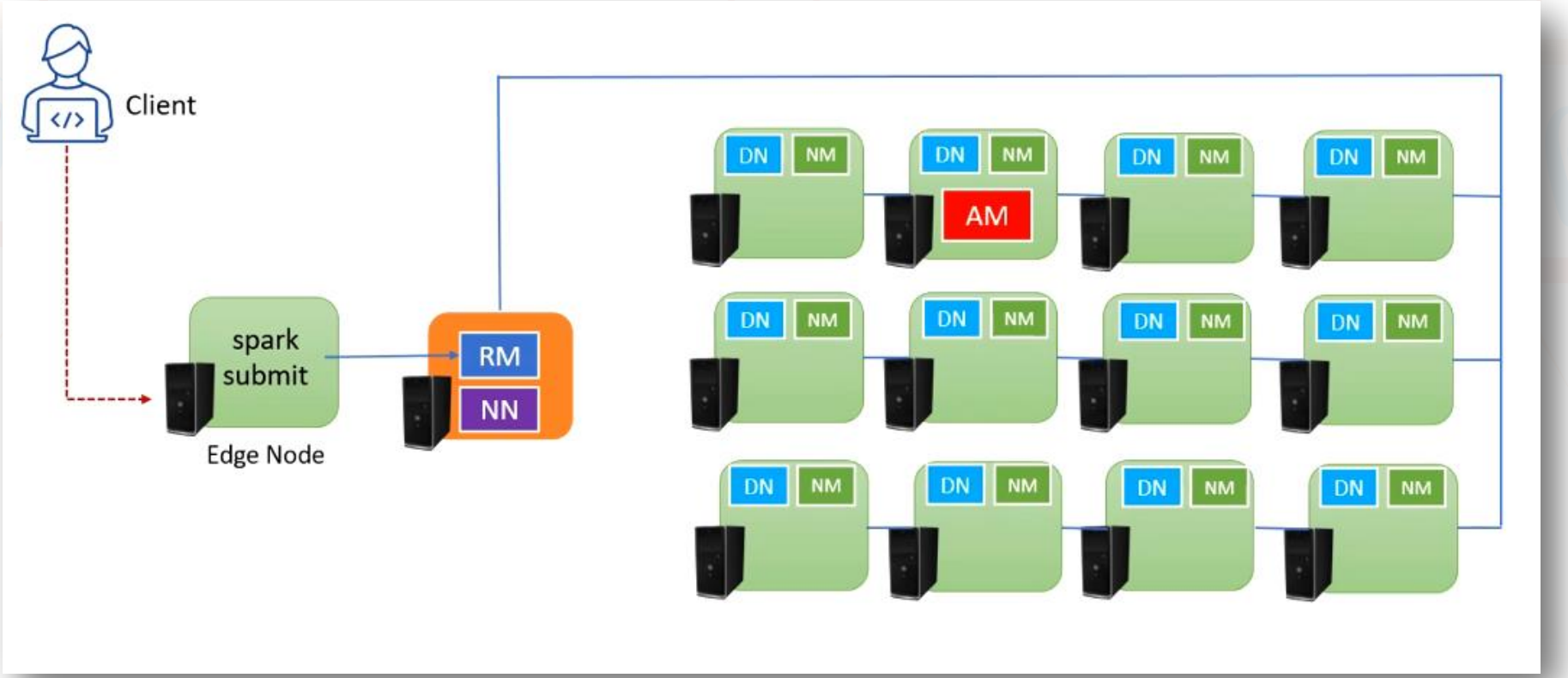


We submit a Map/Reduce program to the RM. The RM coordinates with the NM to start an AM container.
Your M/R program starts running in the AM container. But your program implements the M/R framework. So the M/R framework will take control of the AM container and request the RM for more containers.
The RM will coordinate with multiple NMs and provide some containers to the AM.
Now AM will run your Map function in that container.
These Map functions run in parallel to process your data blocks.
When all Map functions are complete, the M/R framework will start a Reduce function in a new container.
The framework will also collect the map outputs and pass them to the reduce function.
The reduce function will consolidate the output and show the results. That's How Hadoop M/R used to run on the Hadoop cluster. Spark also applies the same approach.

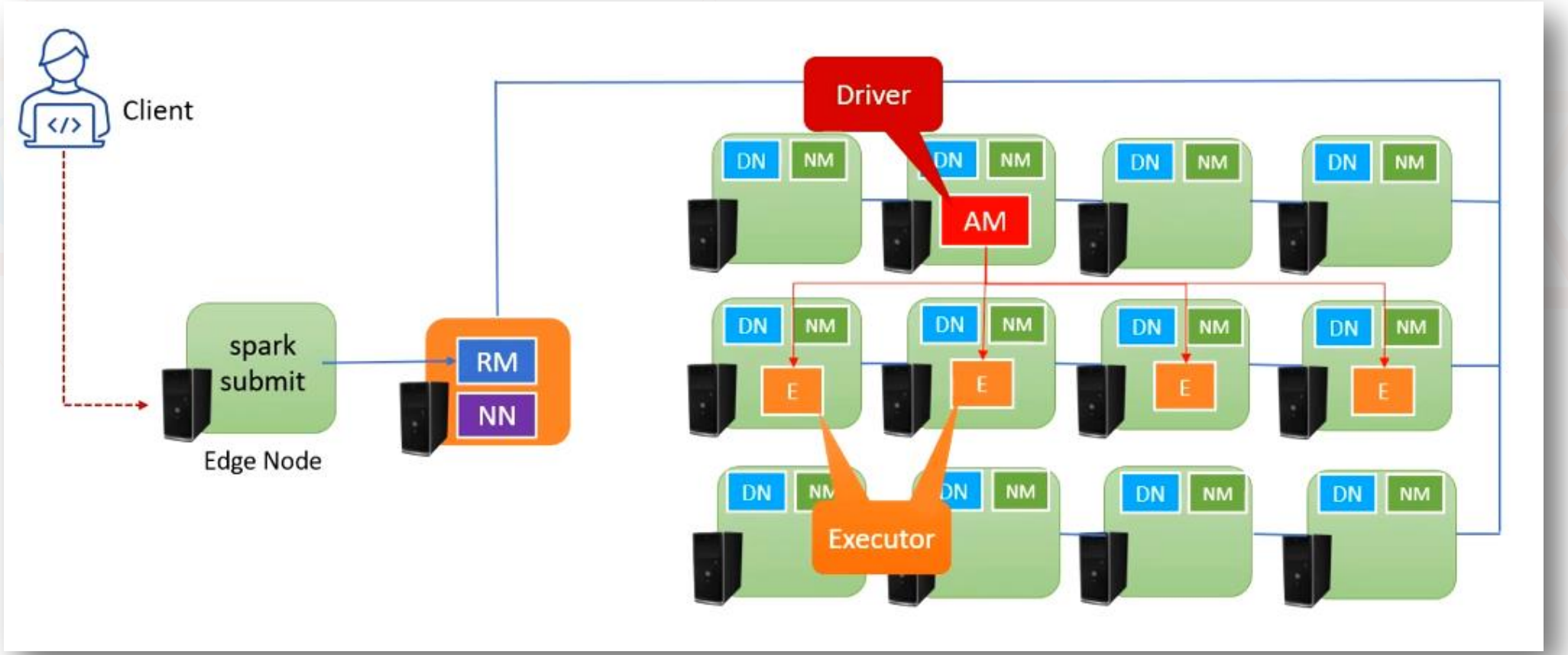
I am showing a YARN Hadoop cluster, and it is configured to run Spark applications. So we have Spark framework also installed on this cluster. You may have a Spark Standalone or Kubernetes managed cluster.
But will have a Resource Management service at the master and a node management service for each worker in all cases. The names of the master and the worker service could be different. However, they work in the same way. So if you understand how Spark runs on the YARN cluster, you know it works in the same way on the other clusters.



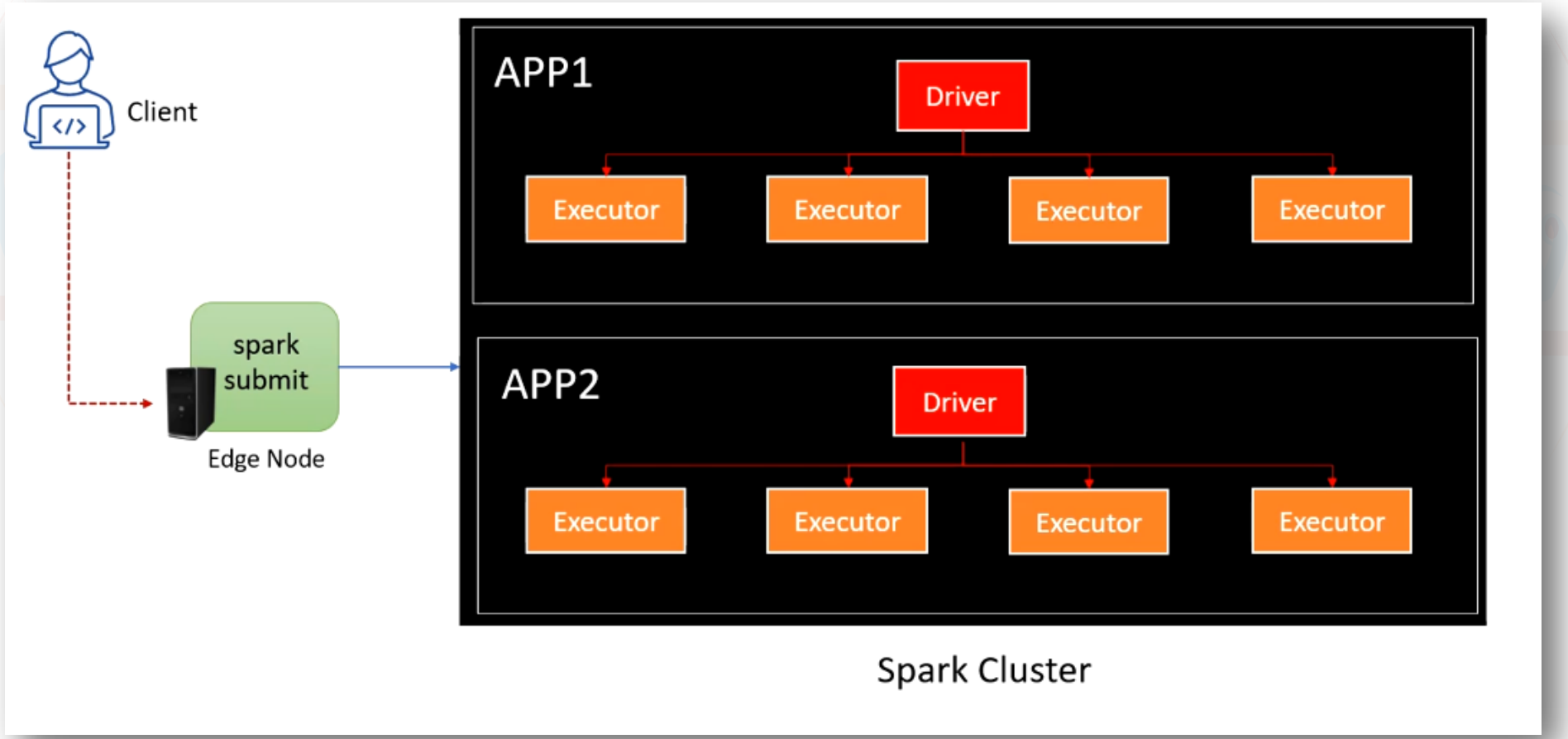
So you will log in to your edge node and use the spark-submit tool to submit your Spark application to the RM. The RM service will coordinate with a NM and allocate an AM container for your application. The AM container starts your application's main method. Your application's main method creates a Spark session object that triggers the Spark Framework. The Spark framework knows that your application must run in parallel.
And for running your code in parallel, you will need some more containers. So the Spark framework will reach out to the RM once again and request more containers for parallel execution. The RM will work with some NMs to allocate a few more containers and pass the details back to the AM. Now AM will start parallel execution of your application in those new containers.



The driver process creates small tasks and runs those tasks in the other containers. These other containers are known as executors. Every Spark application, once submitted to the cluster, runs as one driver and a bunch of executors. The driver starts first, then it will work with the cluster manager to start the executors.
So always remember the notion of the Spark driver and the executors.
You can simply hide the cluster and worker details from this diagram and talk about the driver and executor.
So from now on, I will hide all the cluster details such as cluster manager, node manager, AM container, and all those things.



Assume, I have one APP1 driver and some APP1 executors, and they are doing the necessary work. While they are running, I submit another application, APP2. So a new driver for APP2 will start in a new AM container. This new driver will again request executor containers. The RM will help the driver with new executor containers, and the APP2 driver will assign tasks to these executor containers. Every Spark application runs with its driver and a set of executors. Each Spark application is separate, and it has nothing to do with the other applications.
A Spark application can have one and only one driver, but you can have zero or more executors.

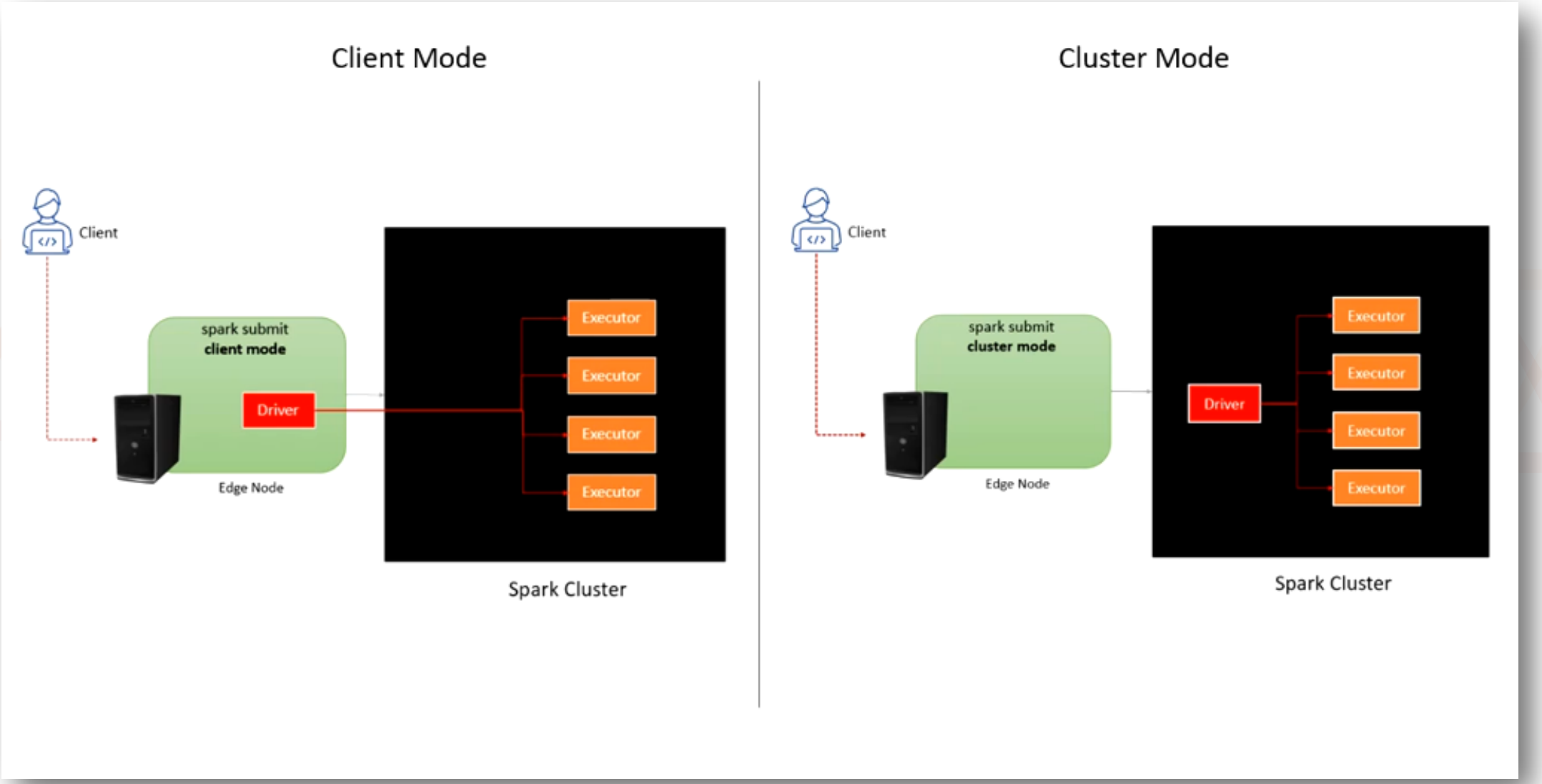


The spark-submit tool comes with many configuration options.
One of the options is the --deploy-mode
This configuration can take one of the following values.

1. client
2. cluster

The default value is the client. So if you do not specify the --deploy-mode. The spark-submit assumes the client mode.
We used cluster mode in the earlier lecture to submit our spark application.
And cluster mode is the recommended value.
You should prefer to use the cluster mode while submitting your spark applications.

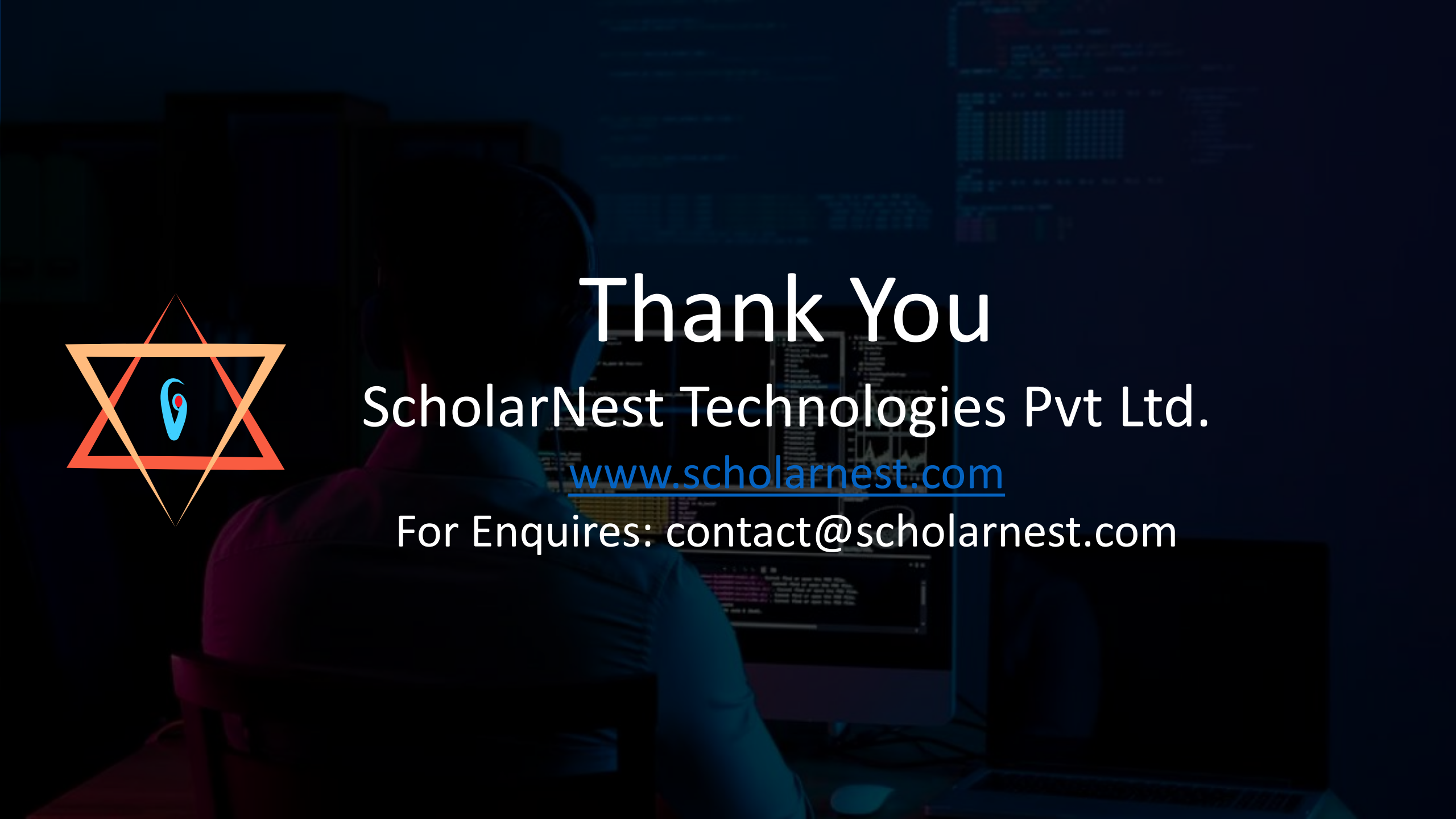
Here is the difference between Client mode and Cluster mode.



The deploy mode determines the location of your Spark driver.
So when you submit an application using the **cluster mode**, your driver will start in the AM container on one of the cluster machines.
We submit the application to the YARN RM, and the RM will start an AM container to run your driver.
The driver runs in the cluster, and it works with the cluster resource manager to start the executors.
You can also set the deploy mode to the **client**.
In that case, the spark-submit will start your driver in a local JVM process.
What does it mean?
Your driver is not running in the cluster.
It is running at your edge node as an independent Java application.
Everything remains the same, except the driver sits outside the cluster and manages your application from outside.

But why do we have two options?
The cluster mode is recommended, and that makes sense also.
I mean, we have a cluster, and everything related to the Spark application should run on the cluster.
So the cluster mode makes good sense.

But why do we have a client mode?
The Spark creators designed the client mode for being able to develop interactive tools.
For example, Spark notebooks, pyspark shell, and Spark SQL tools are designed to run your spark code in client mode. They need to show you the output. So they need the driver running locally to fetch the output.
These tools can't have control when a driver is running in the cluster.
So they always use the client mode and run the driver locally to manage the driver themself.
So the client mode is suitable for interactive work.
But once we finish creating an application, we should package it and deploy it on the cluster.
In those cases, we do not need interactive work.
So we prefer to use the cluster mode to submit our finished applications.
Why? Because we do not want to remain logged in to the edge node.
We simply want to log in to the edge node, submit our application, and log out.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

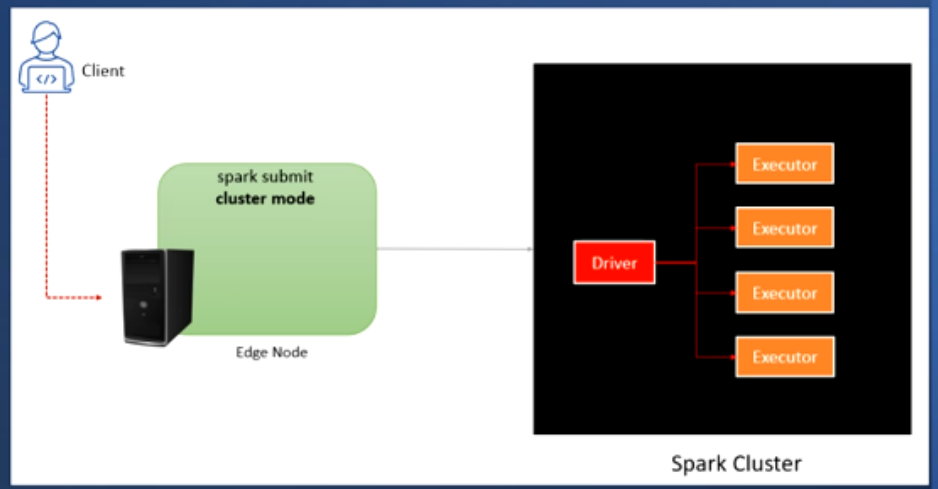
Module:
Spark Cluster and Runtime Architecture

Lecture:
Resource Allocation to Applications

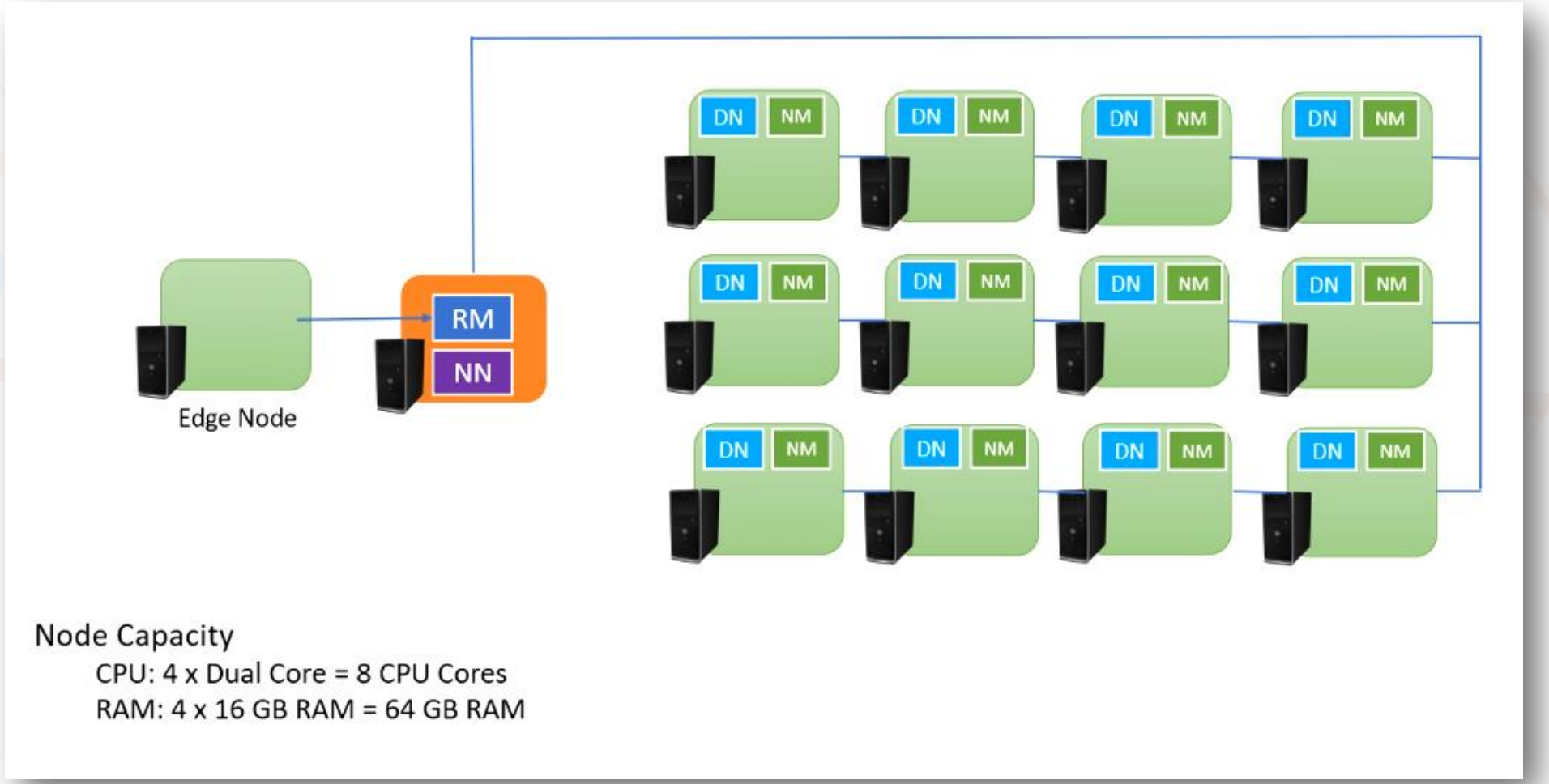
Resource Allocation to Spark Application

I showed you the following spark-submit resource allocation options. Once you understand the notion of Spark driver and the executions, these options start making sense. Now, let us see these options in detail.

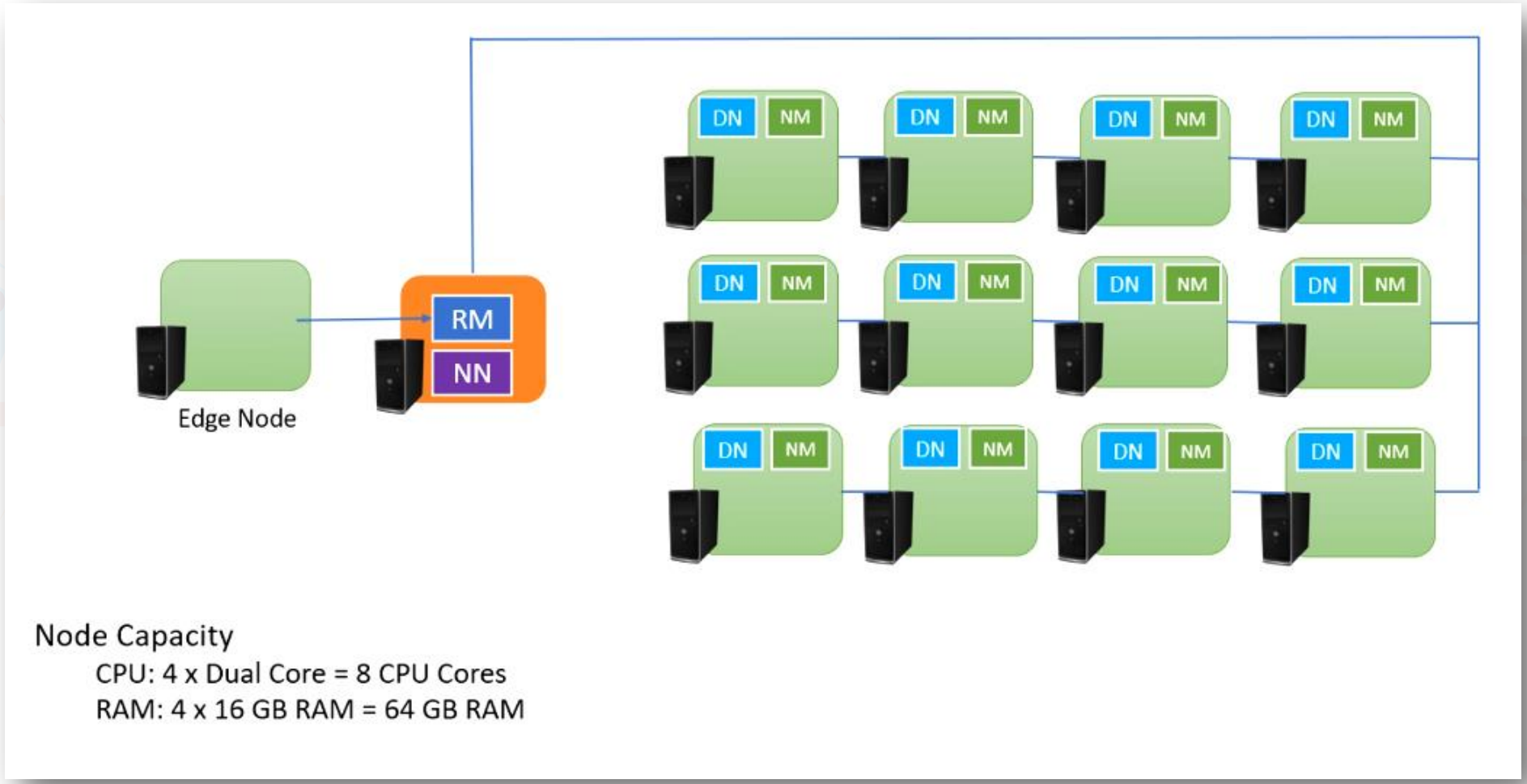
```
Resource Allocation Options
---------------------------------------------------------------------------------------
  --driver-cores NUM          Number of cores used by the driver, only in cluster mode
                              (Default: 1).
  --driver-memory MEM         Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
  --num-executors NUM         Number of executors to launch (Default: 2).
                              If dynamic allocation is enabled, the initial number of
                              executors will be at least NUM.
  --executor-cores NUM        Number of cores used by each executor. (Default: 1 in
                              YARN and K8S modes, or all available cores on the worker
                              in standalone mode).
  --executor-memory MEM       Memory per executor (e.g. 1000M, 2G) (Default: 1G).
```



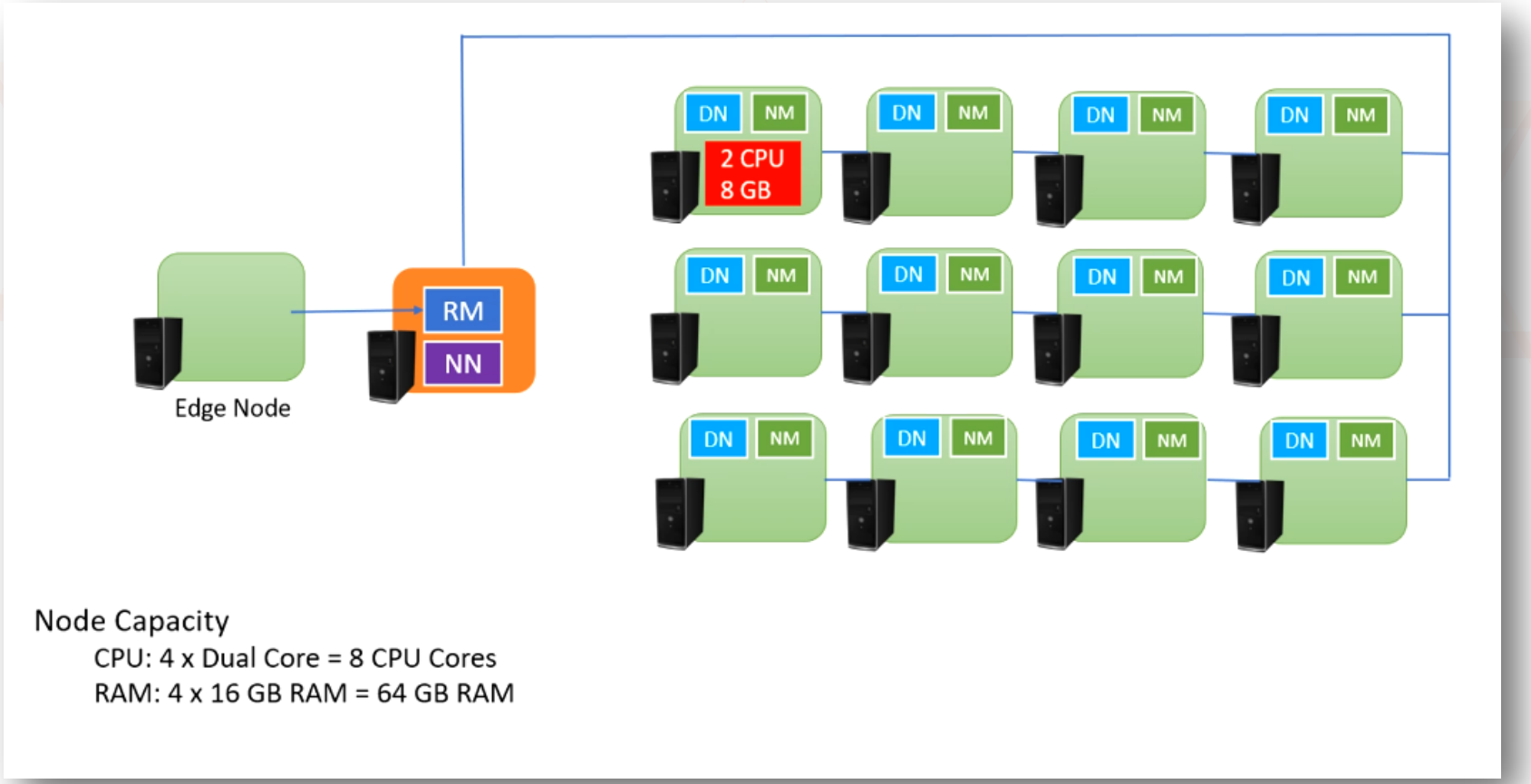
Every spark application runs one driver in a container. In simple terms, a container is a closed resource box on a worker machine. Assume you have a Spark cluster. So you have one master node and a bunch of worker nodes.
These nodes are nothing but a machine. It could be a physical machine, or it could be a virtual machine.



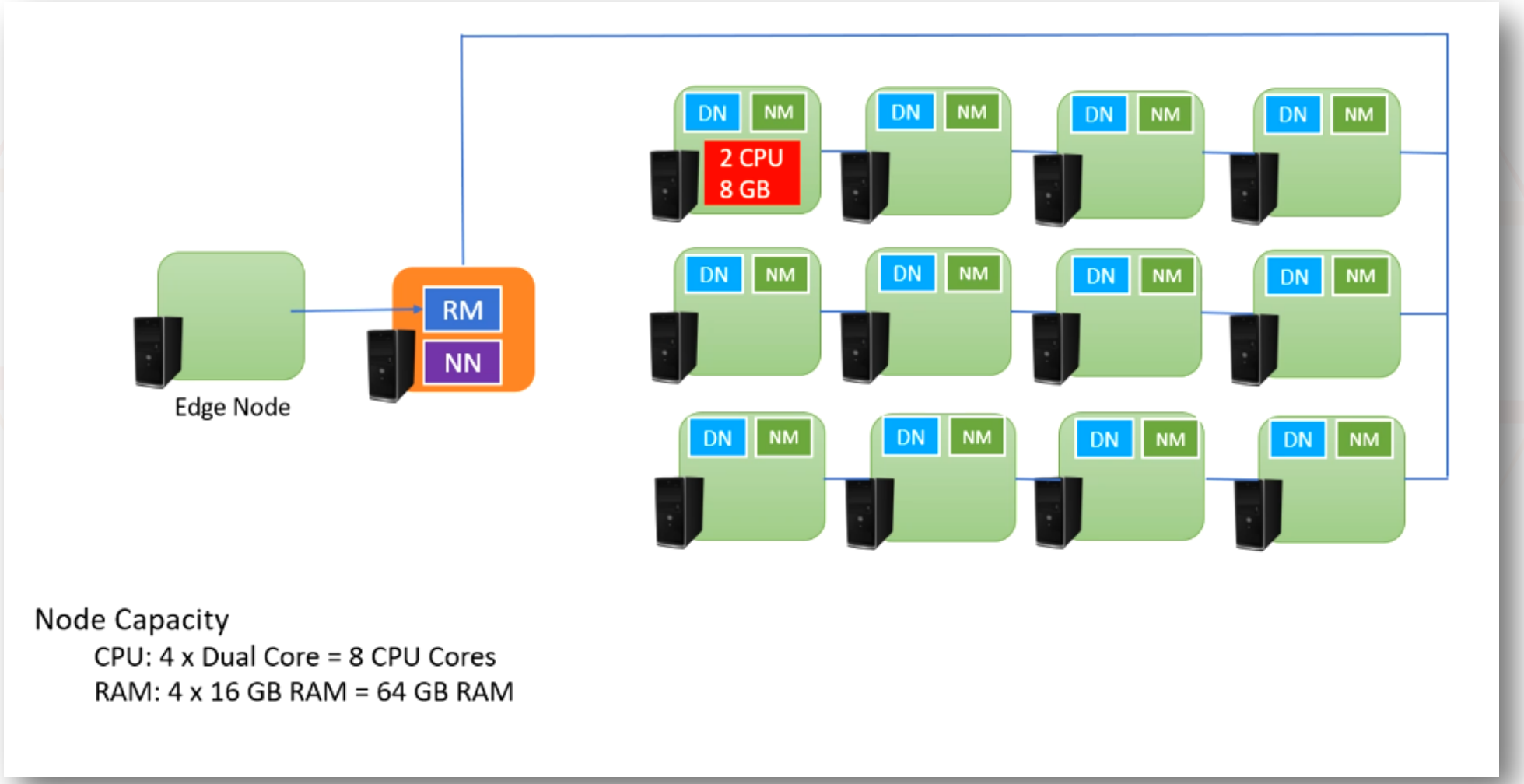
We assume these nodes are computer systems. Each computer system comes with some CPU and Memory attached to it. Let's assume each node of this cluster is the same and has 8 CPU cores and 64 GB of memory.



How does the Spark cluster manager allocate resources for my Spark driver and executor?
The cluster manager will create a container and dedicate some CPU and RAM to the container.
Let's assume I got a container with 2 CPU cores and 8 GB of Memory. You can think of the container as a runtime environment inside the worker node. Your application code runs within the runtime container. It uses 2 CPU cores and 8 GB of Memory. It cannot use anything more than that. Why?
Because the cluster manager gave you only 2 CPU cores and 8 GB memory.



So resource container is a virtual box within a worker node. It comes with some CPU and Memory. Your driver or an executor must run within this virtual box and manage all its work with the given CPU and RAM.



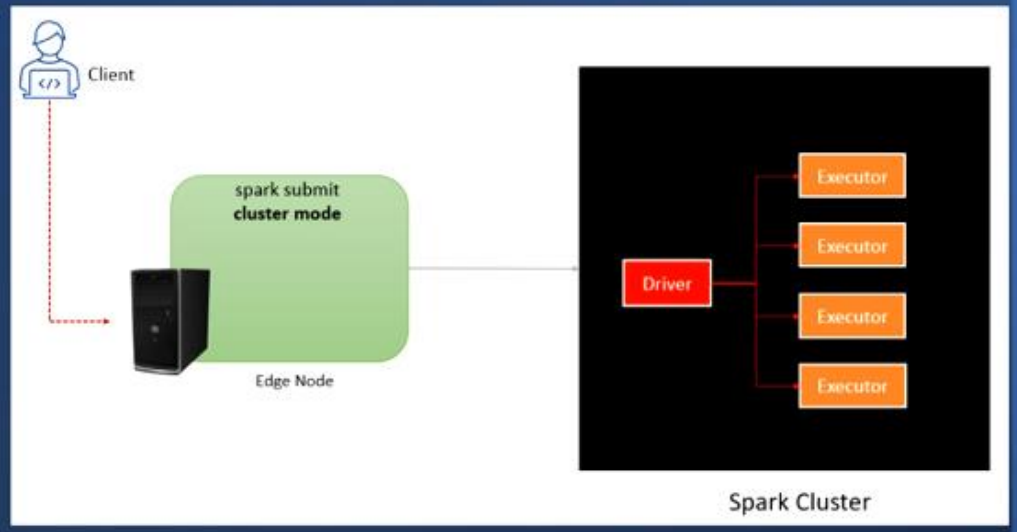
How does a cluster decide how much CPU and Memory to give to a container?
I mean, what is the size of the container? Who decides that? It is your job.
We designed the Spark application.
We know what we are doing in the application and how quickly we want to finish the work.
And how much CPU and Memory are required to do it.
So we ask the cluster manager.
How do we ask? Using the spark-submit options.

You can use --driver-cores to ask the number of CPUs for your driver container.
The default is one. So if you do not ask, the cluster manager will give you one CPU core.
One CPU should be enough for most of the drivers.

Similarly, you can use the-- driver-memory to ask for the driver container memory.
The default value is 1 GB. But you can ask for more.
We typically ask for 2 to 4 GB of memory for the driver in most applications.
But how much to ask is a topic for another lecture.
We have quite a few other things to learn before you can understand how many resources will be required for your application.

We have three options for the executors.

```
Resource Allocation Options
----------------------------------------------------------------------------------------
  --driver-cores NUM          Number of cores used by the driver, only in cluster mode
                              (Default: 1).
  --driver-memory MEM         Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
  --num-executors NUM         Number of executors to launch (Default: 2).
                              If dynamic allocation is enabled, the initial number of
                              executors will be at least NUM.
  --executor-cores NUM        Number of cores used by each executor. (Default: 1 in
                              YARN and K8S modes, or all available cores on the worker
                              in standalone mode).
  --executor-memory MEM       Memory per executor (e.g. 1000M, 2G) (Default: 1G).
```



We use --num-executors to ask the number of executors.
The default value is two. So if you do not ask, RM will give you only two executor containers.
But you can ask for more. You can ask for 10, 20, 50, or whatever you want.
But remember, asking doesn't mean you will get it.
All these options are for asking the RM for resources.
What you actually get will depend upon what is available to the RM.
You can ask for 100 executors. But your cluster may be small, and RM might not have enough capacity to create 100 executors.
So you won't get it. You will get what best you can but not what you asked.

The next option is for asking the executor cores. The default is one. So if you do not ask, each executor will get only one CPU core. You can ask more. The best practice is to ask for 4-5 CPU cores for an executor.
So you can use --executor-cores options to request the CPUs for your executor container.
What if I ask for 10 CPUs?
In our example, each worker comes with 8 CPU cores. A container lives on a worker node. You are asking for 10 CPU cores. All we have is 8 CPU cores on a worker.
So you will not get 10 CPU cores.
The RM cannot give it to you because there are not enough CPUs at a worker.
And a container cannot expand across workers.
Every container lives on a single machine. You cannot combine two workers and create a larger container. A container must live on a single worker.
So, you must know your total cluster capacity and your worker node capacity before asking for resources. There is no point in asking for more than the available capacity.

The last option is to ask for the executor-memory.
The default value is 1 GB, but you can ask more.

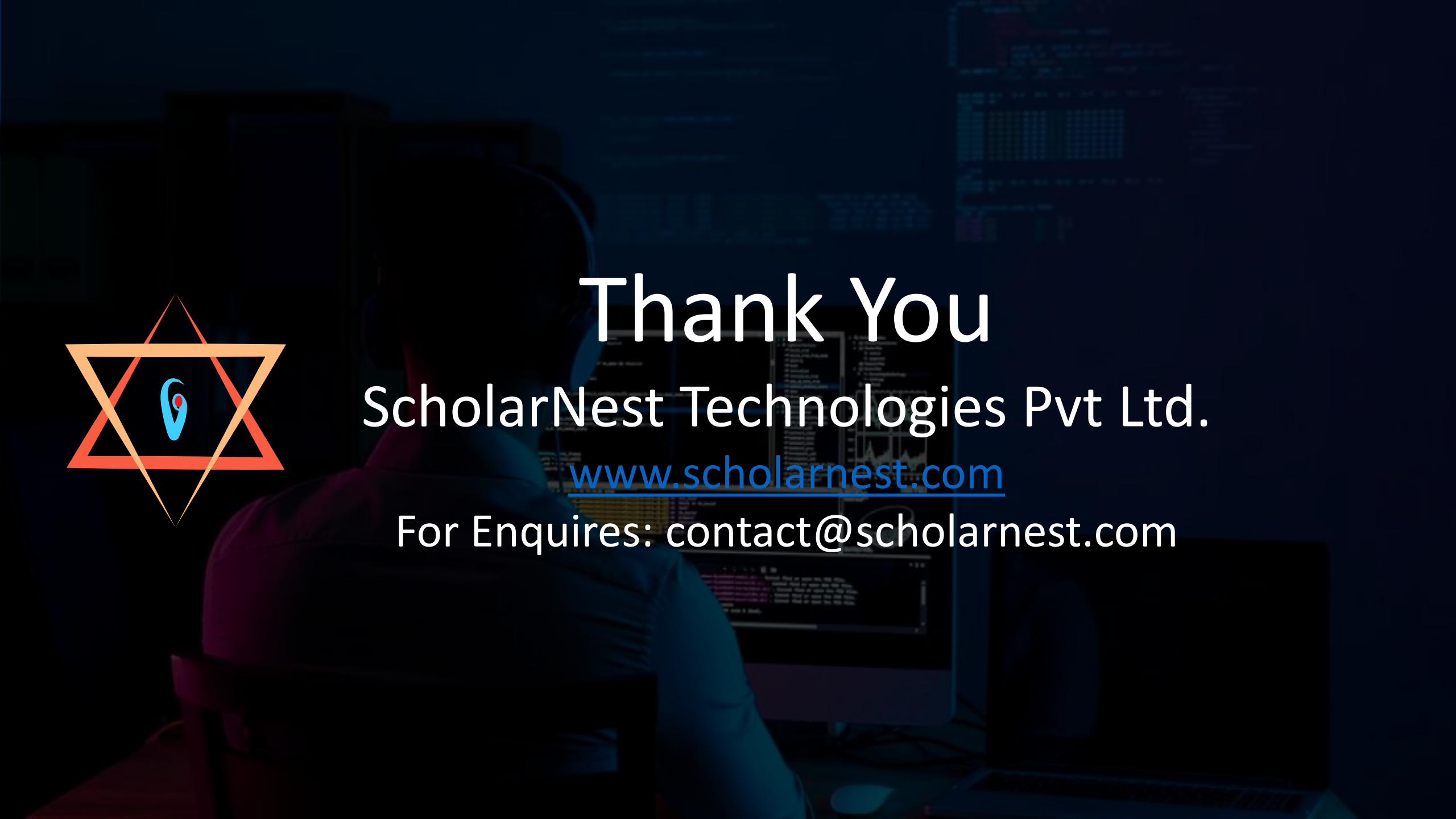
We run Spark applications on two types of Spark clusters:

1. Permanent cluster

   The permanent cluster scenario is mainly used with on-premise Hadoop clusters.
   In a typical case, your company might create a large Hadoop cluster and keep it running all the time. You and other teams will create Spark applications and run them on the same Hadoop cluster. You may have a shared and permanent cluster running all the time.
   And you may have a requirement to deploy your application on a shared, long-running existing cluster. All these spark-submit options are helpful for the permanent cluster scenario.

2. On-Demand cluster

   It is primarily used in the cloud platforms. In this scenario, you will create a cluster, deploy your application on the cluster, and wait for your application to finish and terminate the cluster. All this can be automated. You can create an on-demand cluster specifically for your application, run your application and delete the cluster when your application finishes. The resource allocation for these on-demand cluster works differently. Because we are creating a cluster specifically for our application.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Cluster and Runtime Architecture

Lecture:
Cluster and Runtime Architecture

Spark Cluster and Runtime Architecture

So far in this course, you learned a lot about the following things:

1. Spark cluster and its architecture
2. Spark Application DAG and its execution plan

In this lecture, I will combine both of these to learn about how the Spark application runs on the cluster.

I will quickly revise some concepts you already learned and then tie everything together.
So let's start with a quick revision of some Spark cluster-related concepts. Here is a list of some Spark cluster concepts:

1. Spark Cluster, Master, Worker, and Resource Container
2. Application Master, Driver, and Executor
3. Driver Cores and Memory
4. Executor Cores, Slots, and Memory

And here is a list of Spark application concepts:

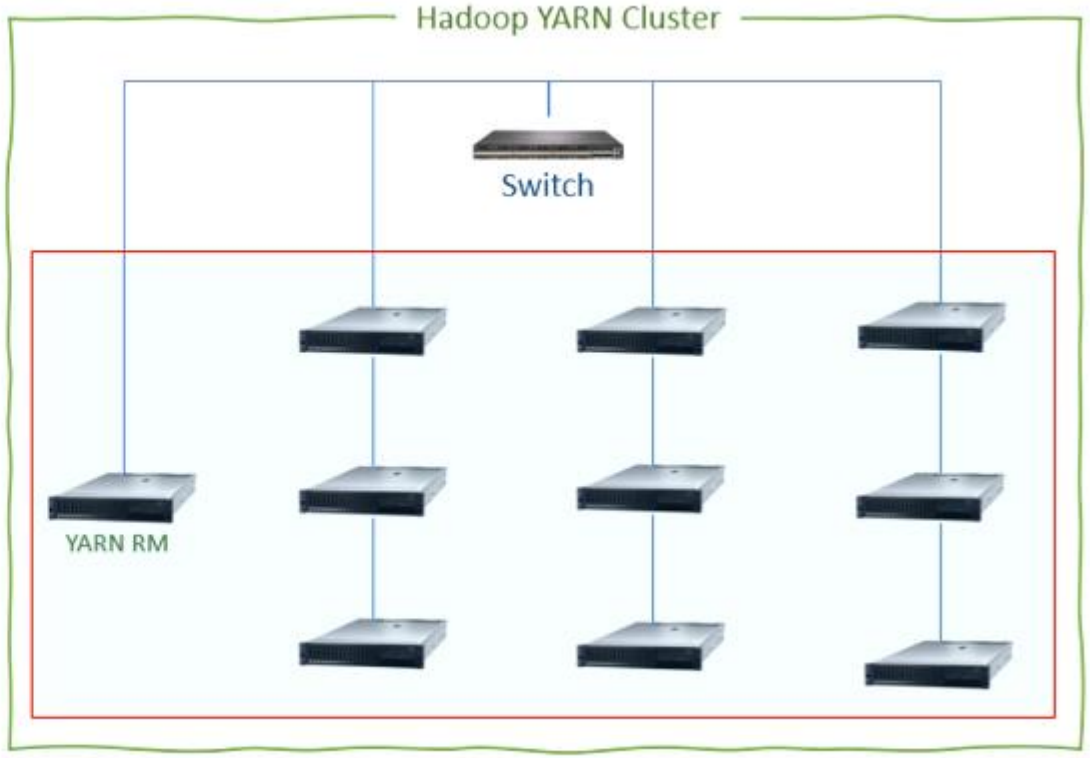
1. Spark Execution Plan and DAG
2. Job, Stages, and Tasks

I hope you have already learned these concepts because I already covered them in the earlier lessons. But let me quickly revise these things, and we will start with the Spark Cluster.

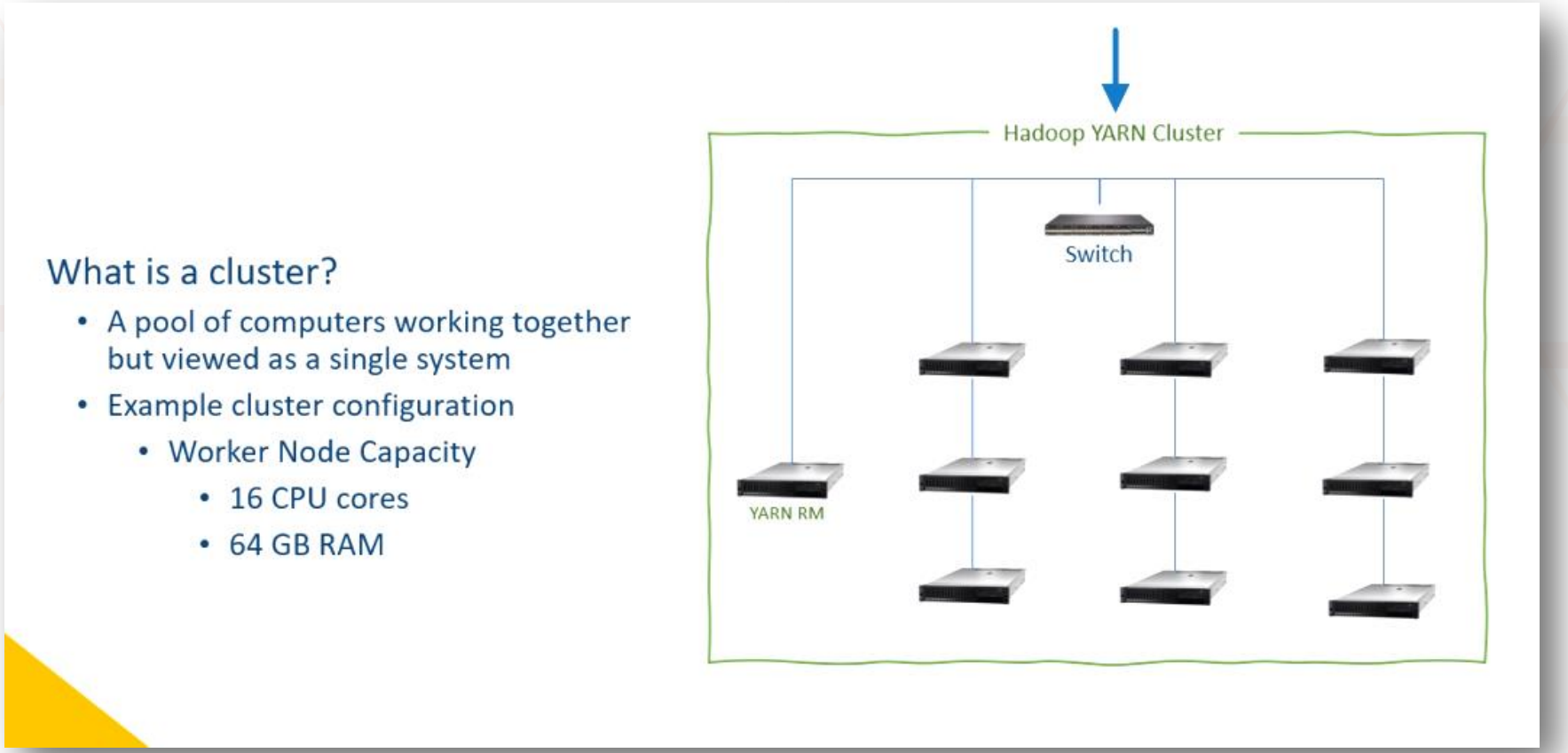
Spark application runs on a cluster. And a cluster is nothing but a pool of physical computers. For example, I have a cluster of 10 machines. Each machine in this example cluster has 16 CPU cores and 64 GB of RAM. They are networked, and we created a cluster of these ten machines.

What is a cluster?

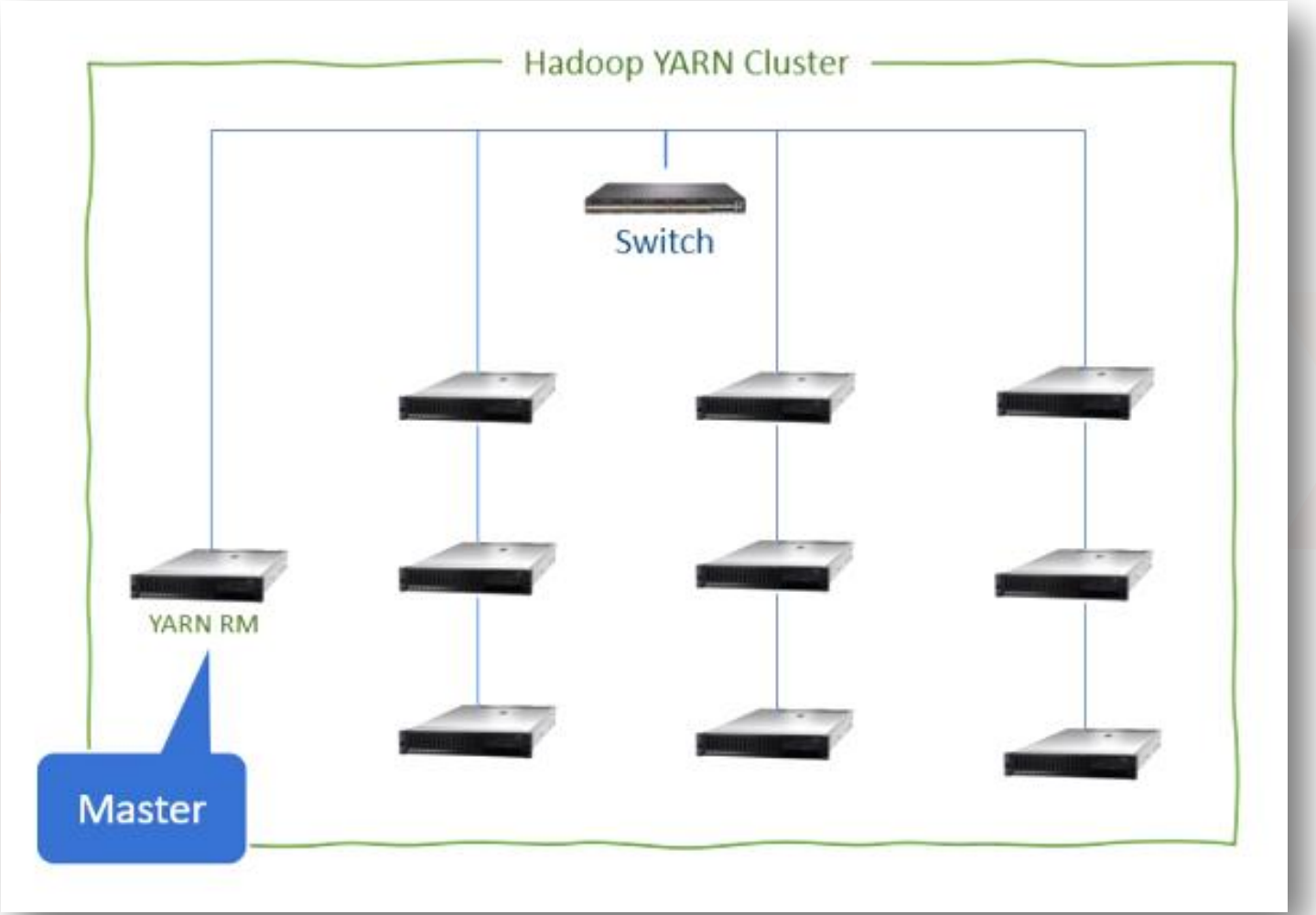
- A pool of computers working together but viewed as a single system
- Example cluster configuration
  - Worker Node Capacity
    - 16 CPU cores
    - 64 GB RAM



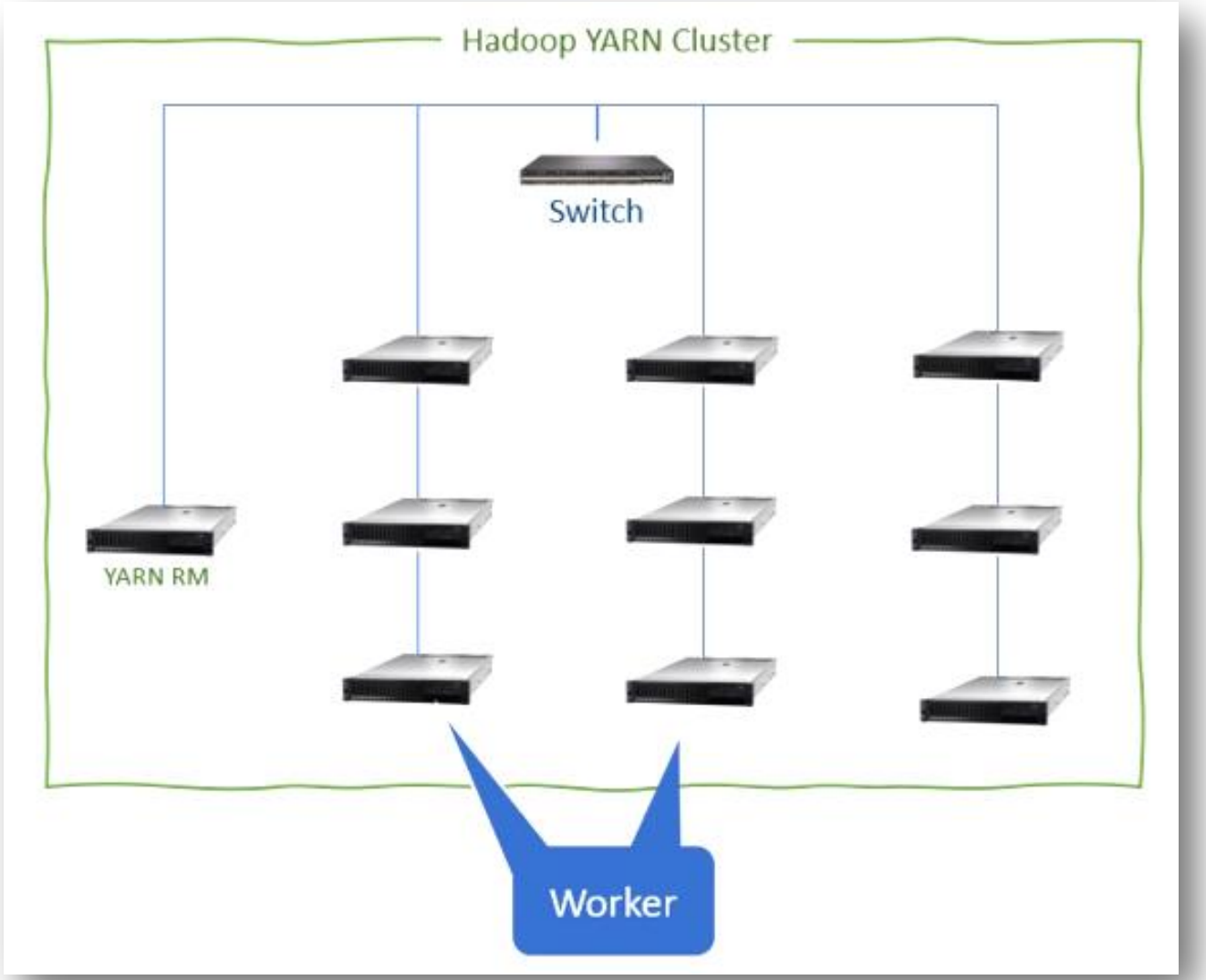
However, just networking is not enough to create a distributed computing cluster. You also need to install and configure some cluster management systems. We have many cluster management systems such as Hadoop YARN, Spark Standalone, and Kubernetes. But for simplicity, let's assume I created this cluster using Hadoop YARN. So I have a Hadoop cluster. The entire pool of computers is termed a cluster.



We have one dedicated machine known as the Master node

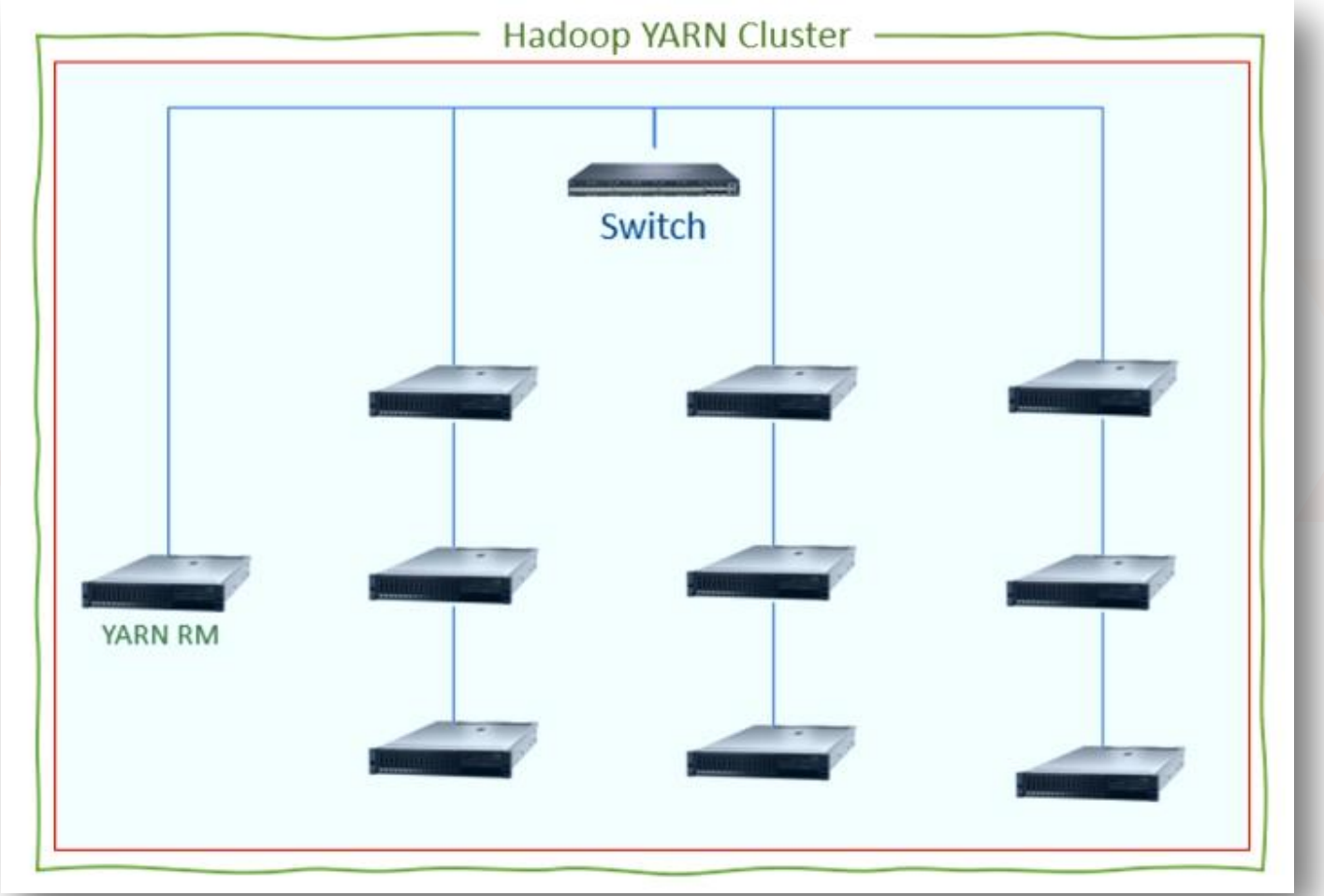


And the rest of the individual machines are known as Worker nodes.

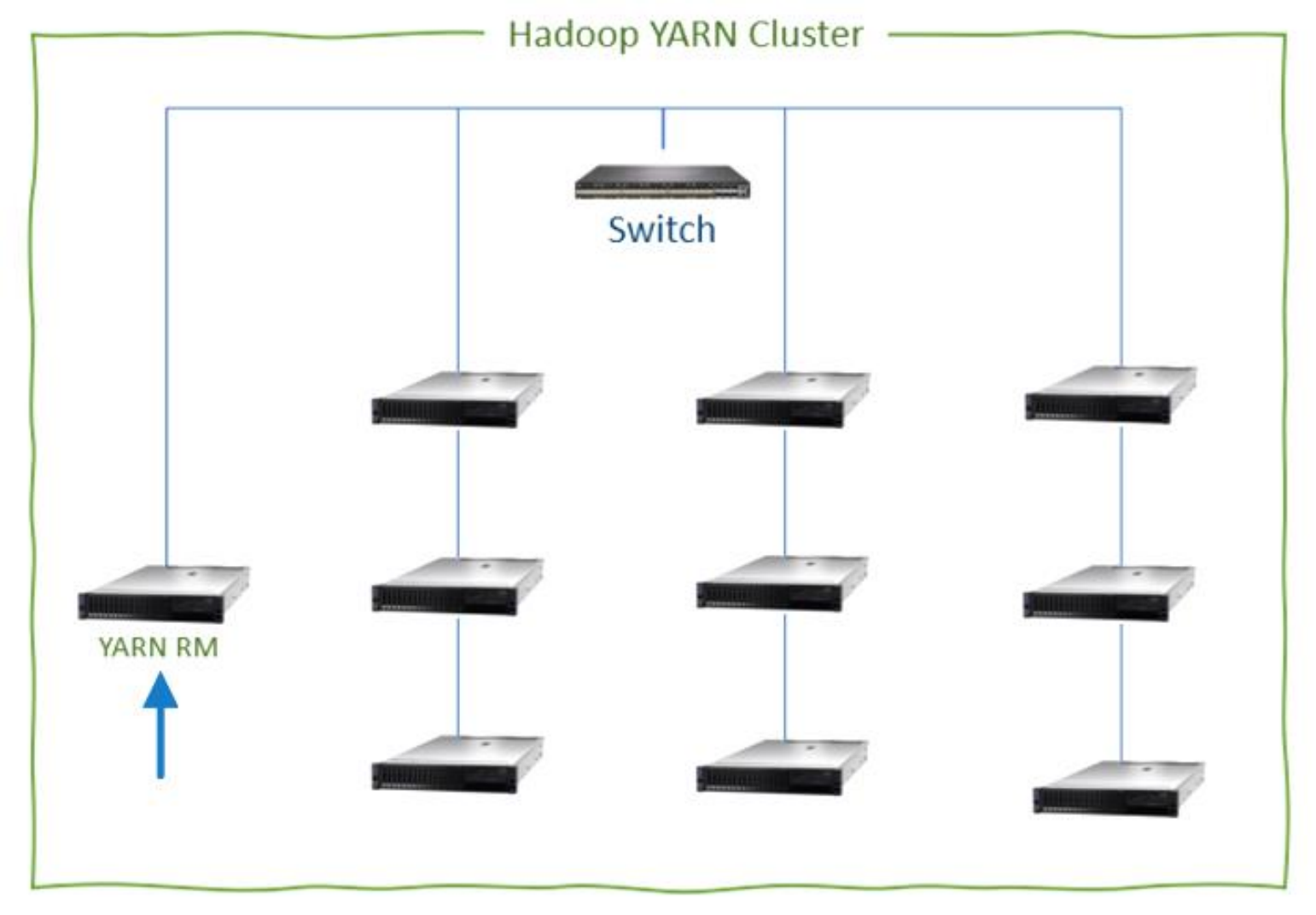


So, we have covered three concepts till now.

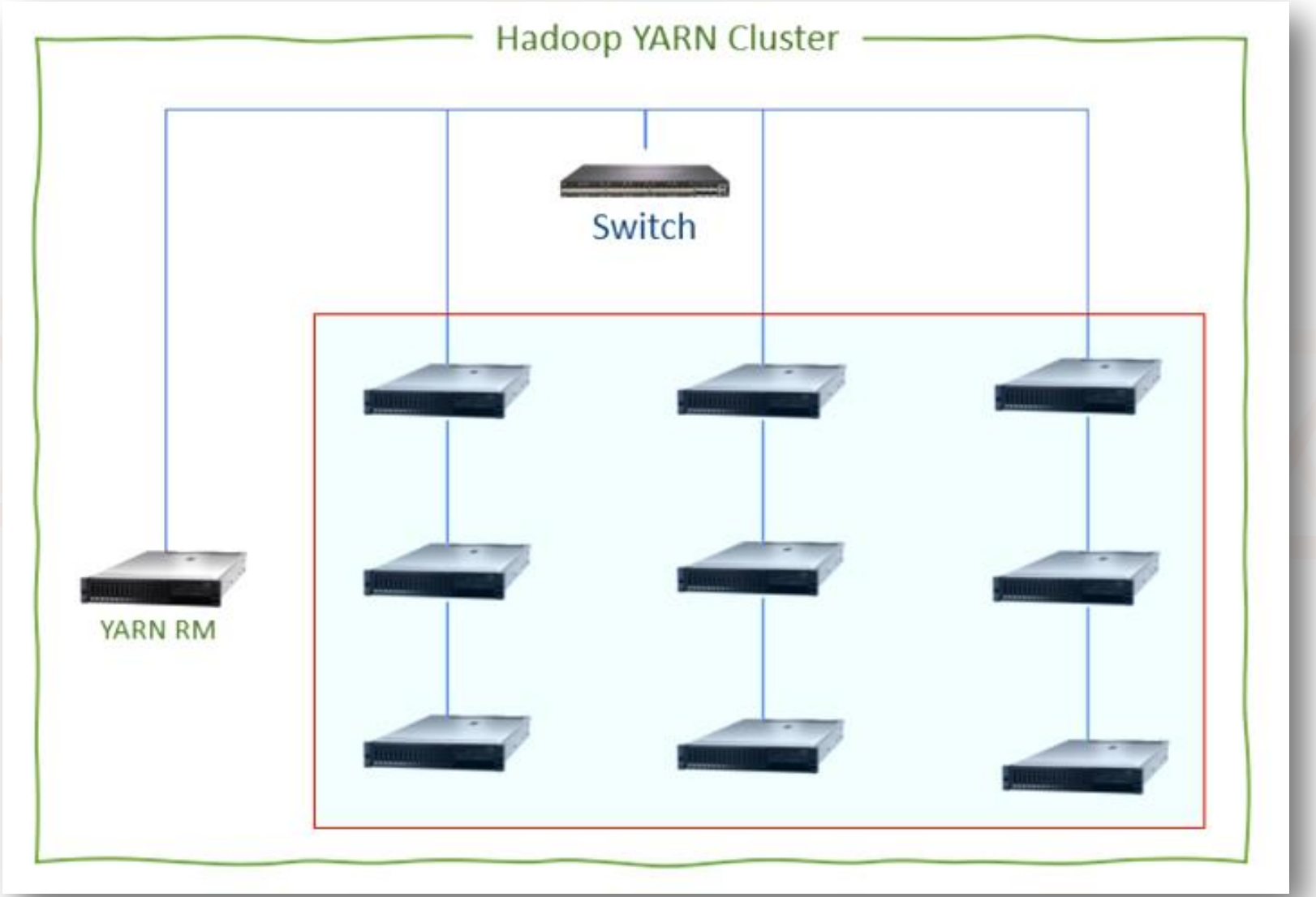
1. **Cluster:** A cluster is a pool of resources tied together by a cluster management system such as Hadoop YARN.



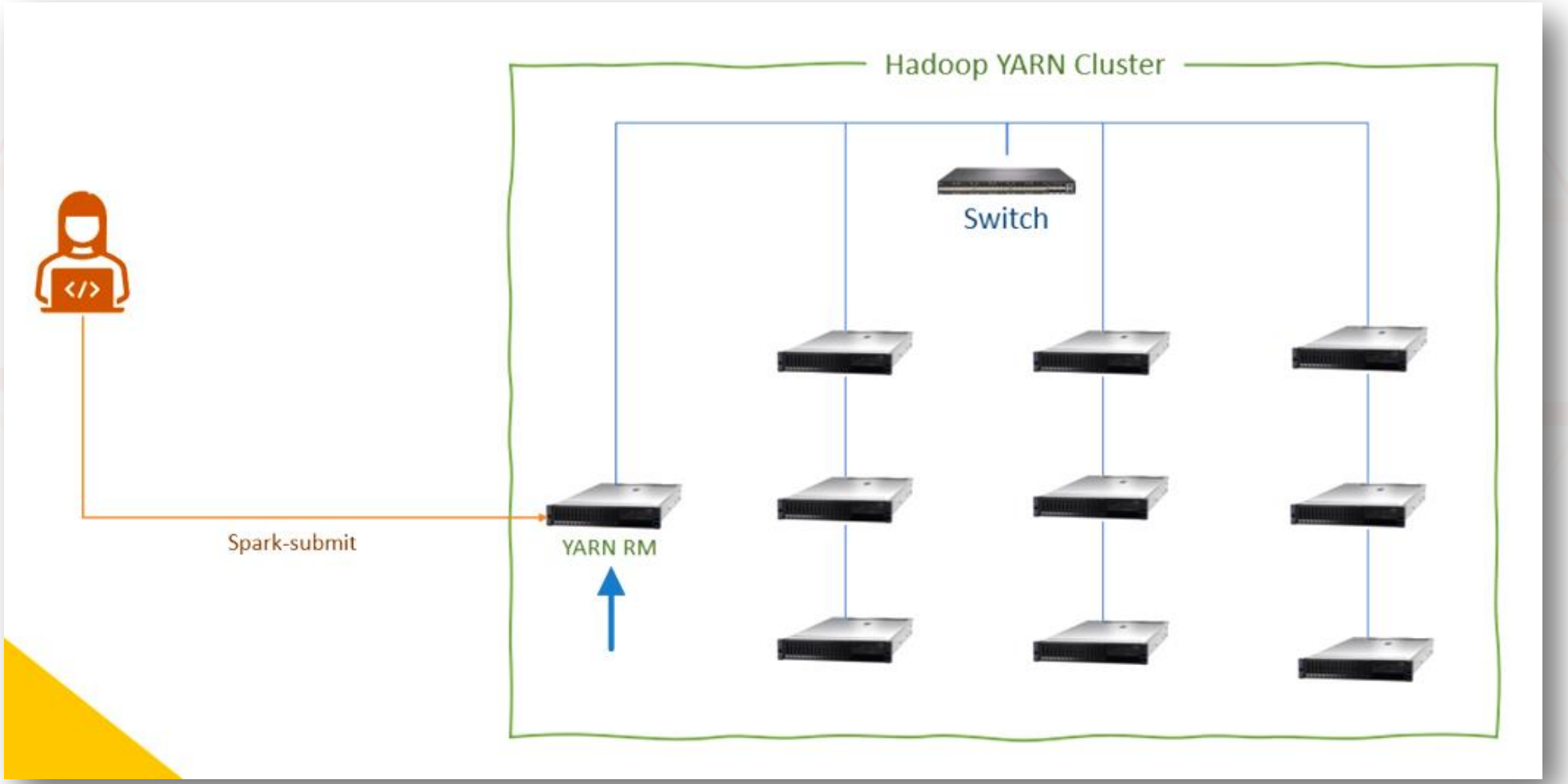
2. Master Node: Almost every cluster needs one dedicated machine to run cluster management services, and that machine is the Master Node.



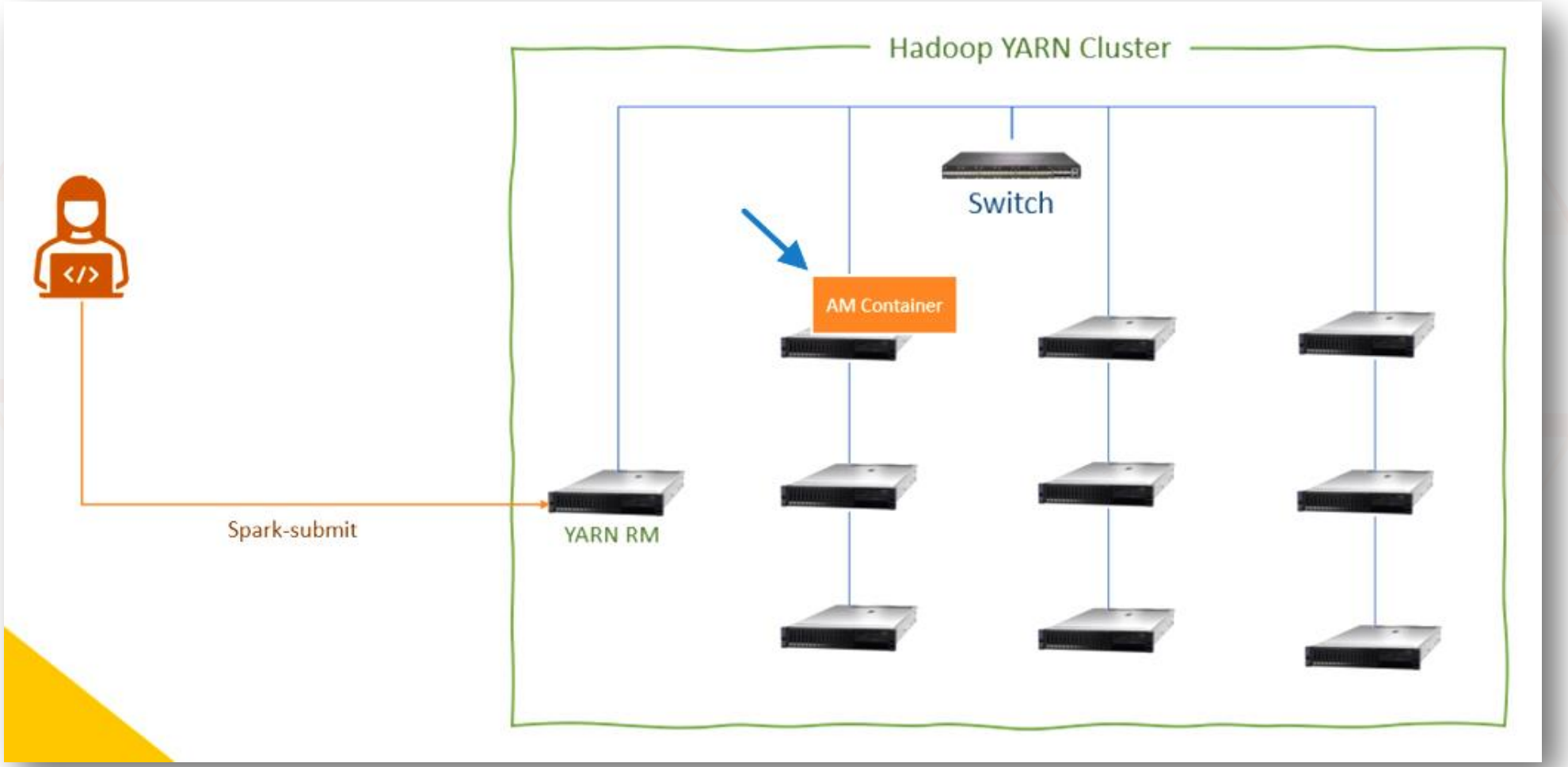
3. **Worker Node:** All other machines in the cluster are Worker nodes.



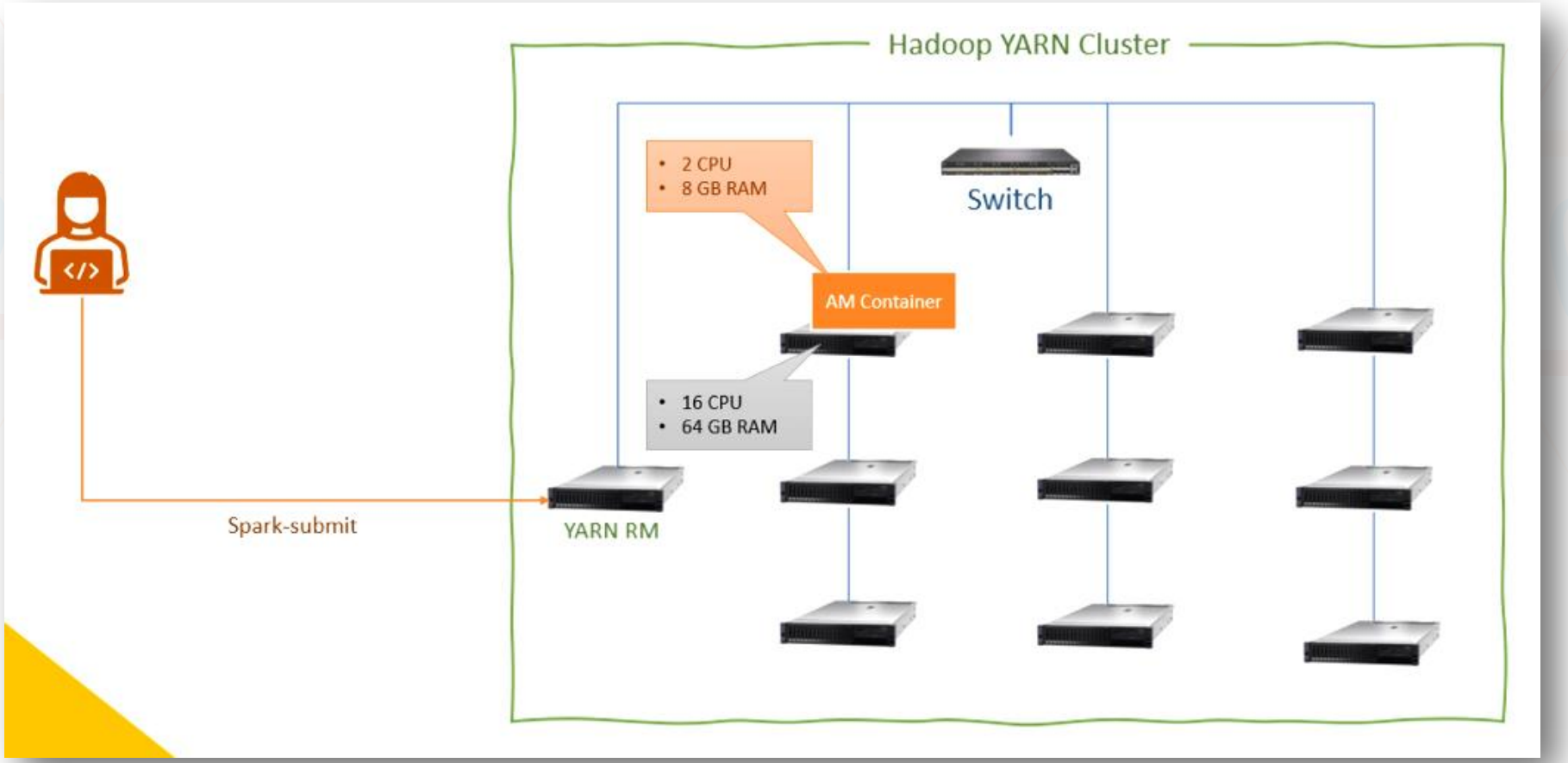
Now I want to run a Spark application on this cluster. So I will use the spark-submit command and submit my spark application to the cluster. My request will go to the master node. In our example, the master node is a YARN resource manager. So My request will go to the YARN resource manager.



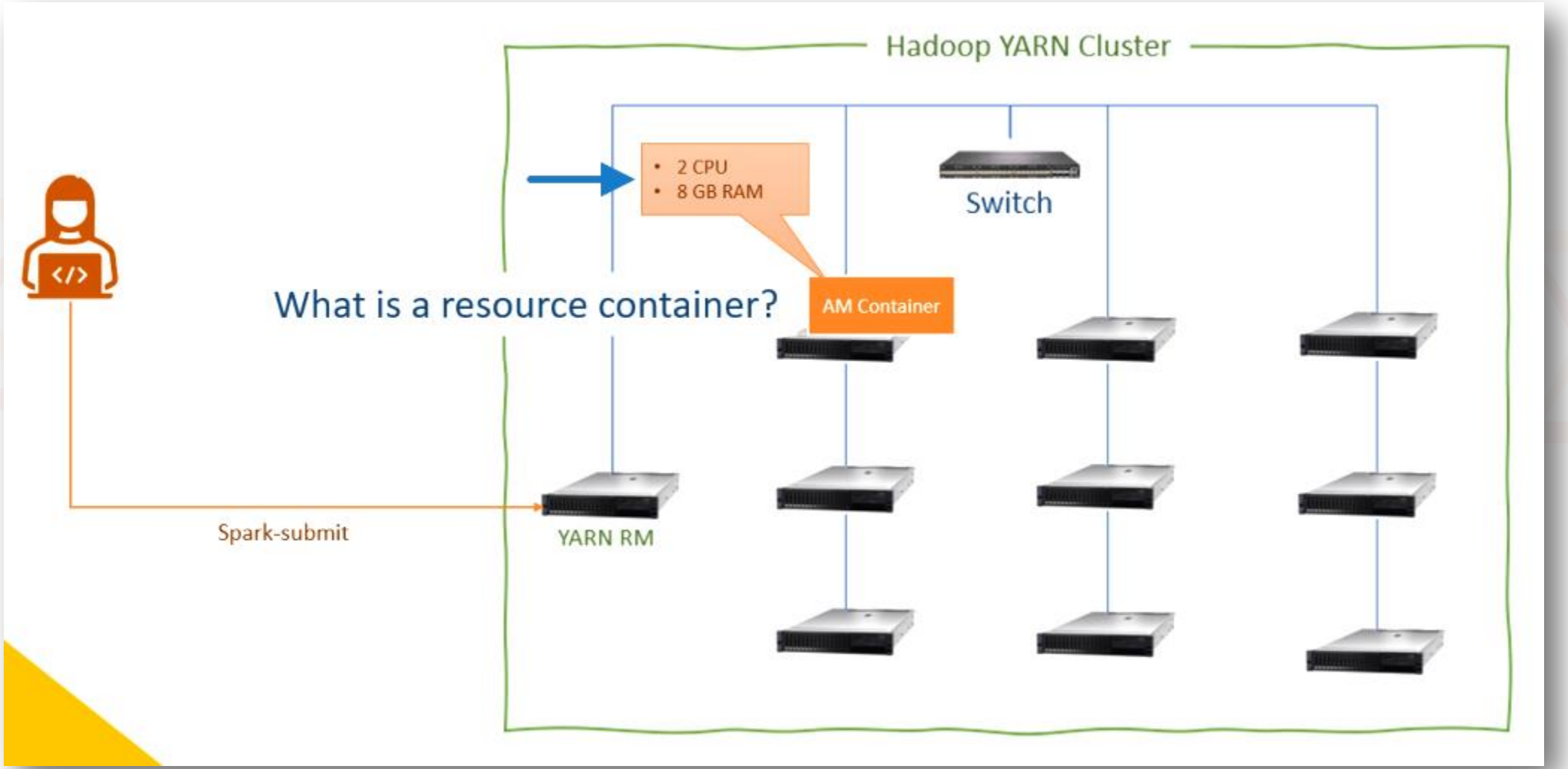
The YARN RM will create one resource container on a worker node and start my application's main() method in the container. But what is the resource container? A resource container is an isolated virtual runtime environment. It comes with some CPU and memory allocation.



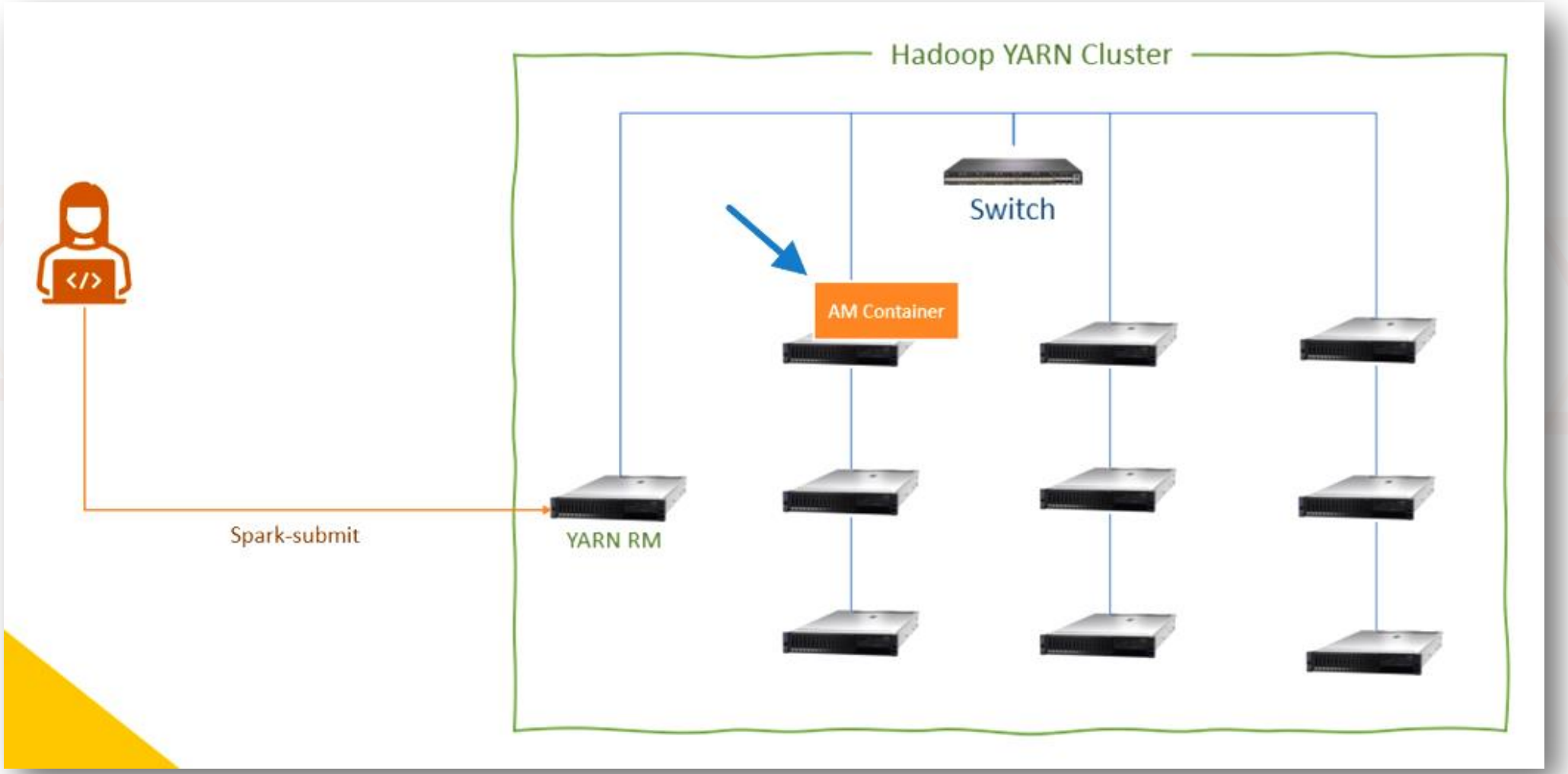
Let's assume YARN RM gave 2 CPU Cores and 8 GB memory to this resource container and started it on a worker node. The worker node has got 16 CPU cores and 64 GB of memory. But YARN RM took 2 CPU cores and 8 GB memory and gave it to my container.
Now my application's main() method will run in the container, and it can use 2 CPU cores and 8 GB of memory.



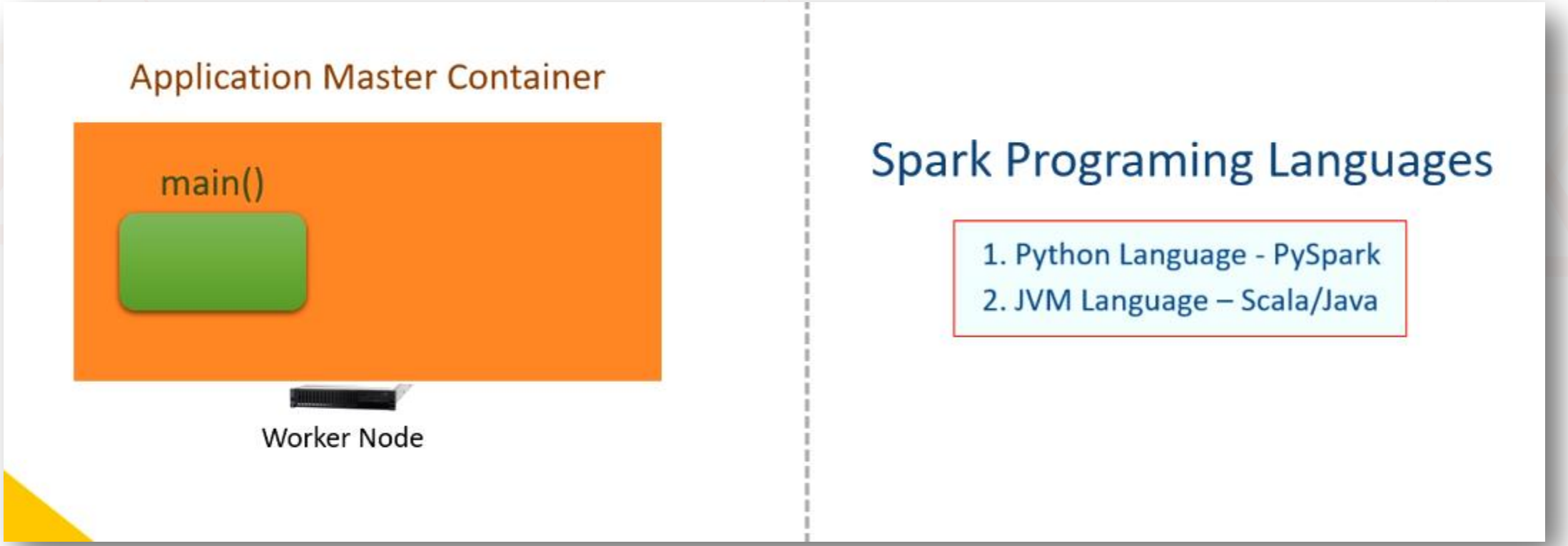
So, now we understand what is a resource container. It is a virtual runtime capacity box with some CPU and Memory. In our example, the first resource container is given 2 CPU cores and 8 GB of memory.



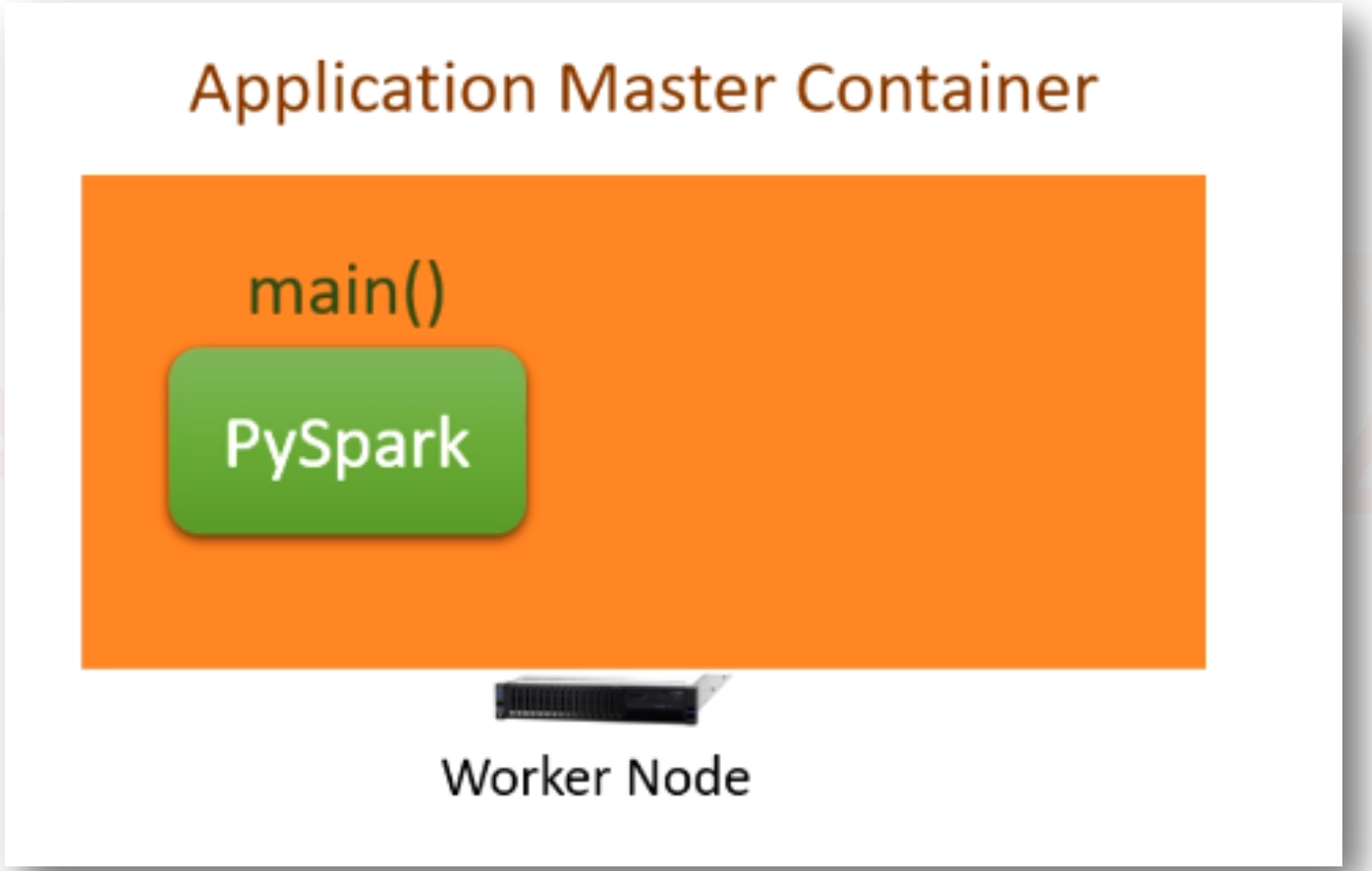
So we submit a Spark application to the cluster manager, and it starts our Spark application's main method inside a resource container.
This one is my application's first resource container, known as Application Master Container.



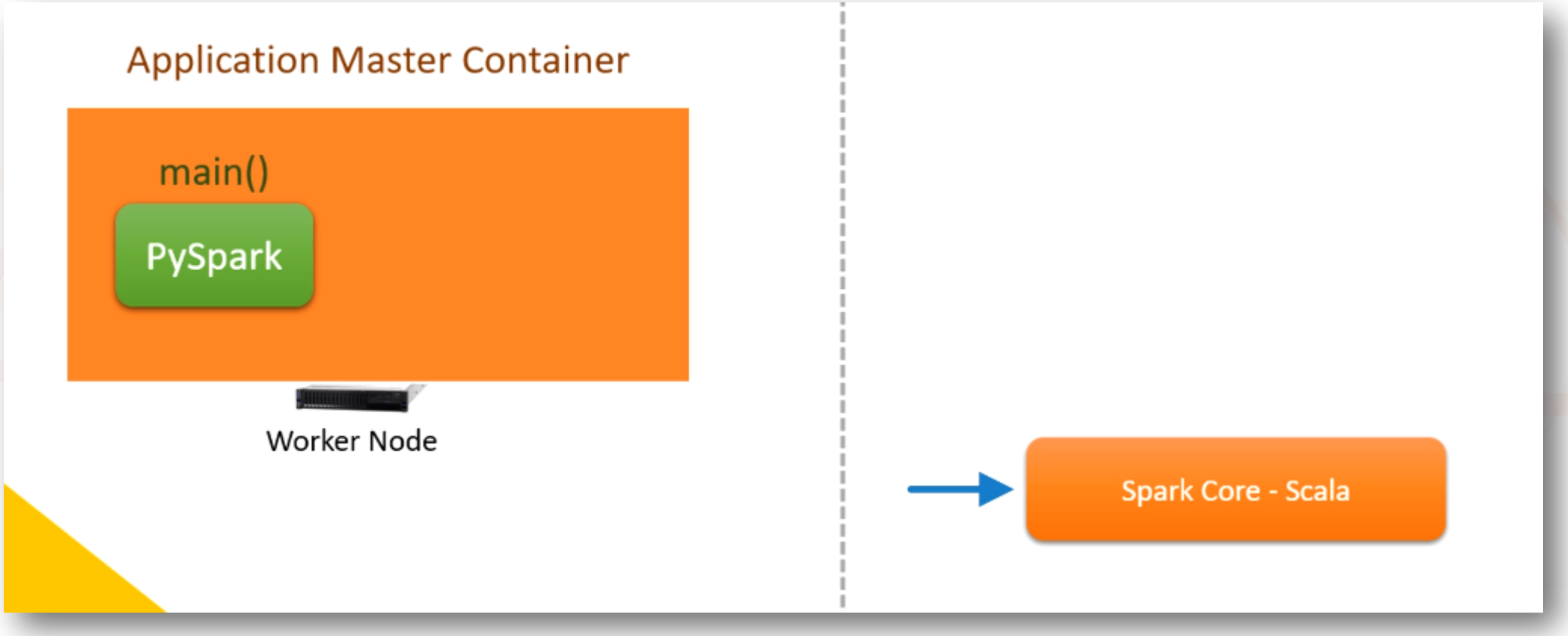
Now let's go inside the AM Container and see what happens there.
The container is running the main() method of my application. And we have two possibilities here. My main method could be a PySpark application, or it could be a Scala or Java application. We write Spark applications in two flavours: Pyspark and JVM-based languages such as Scala or Java.



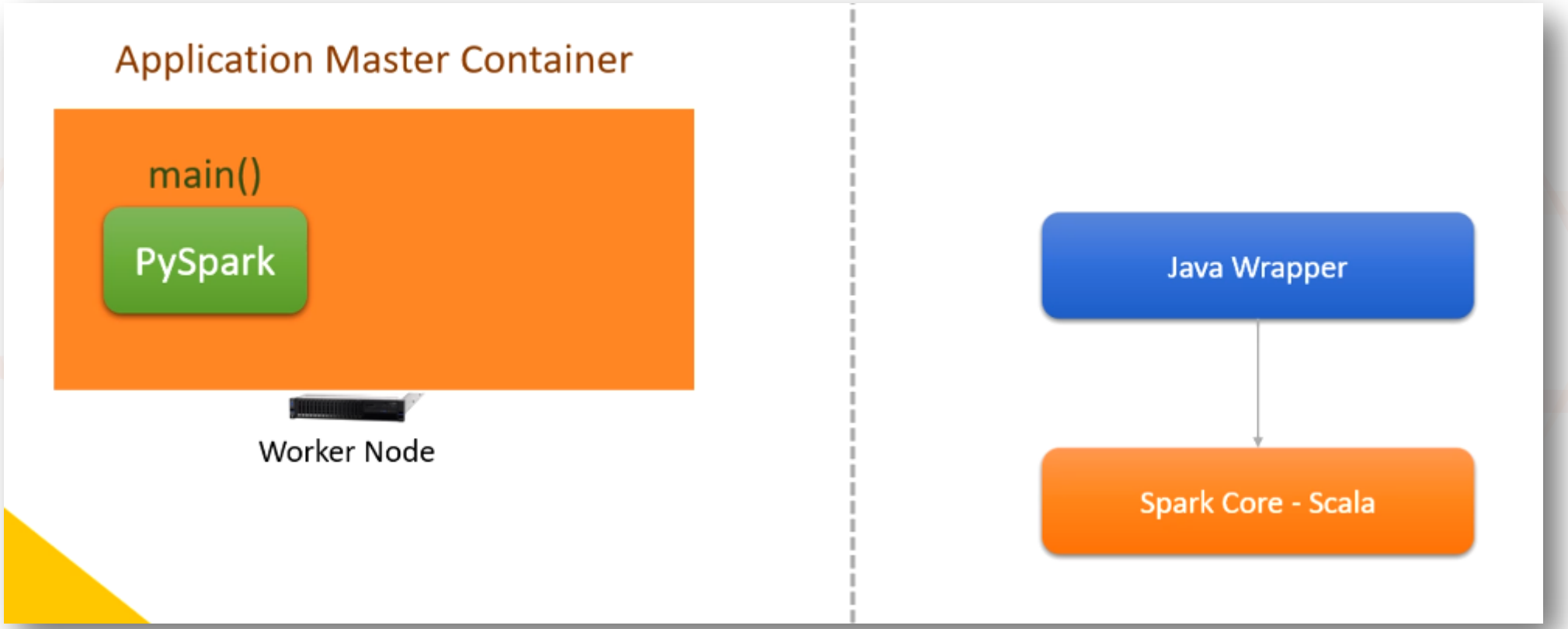
So let's assume my application is a PySpark application.
But Spark is written in Scala and runs in the Java virtual machine.



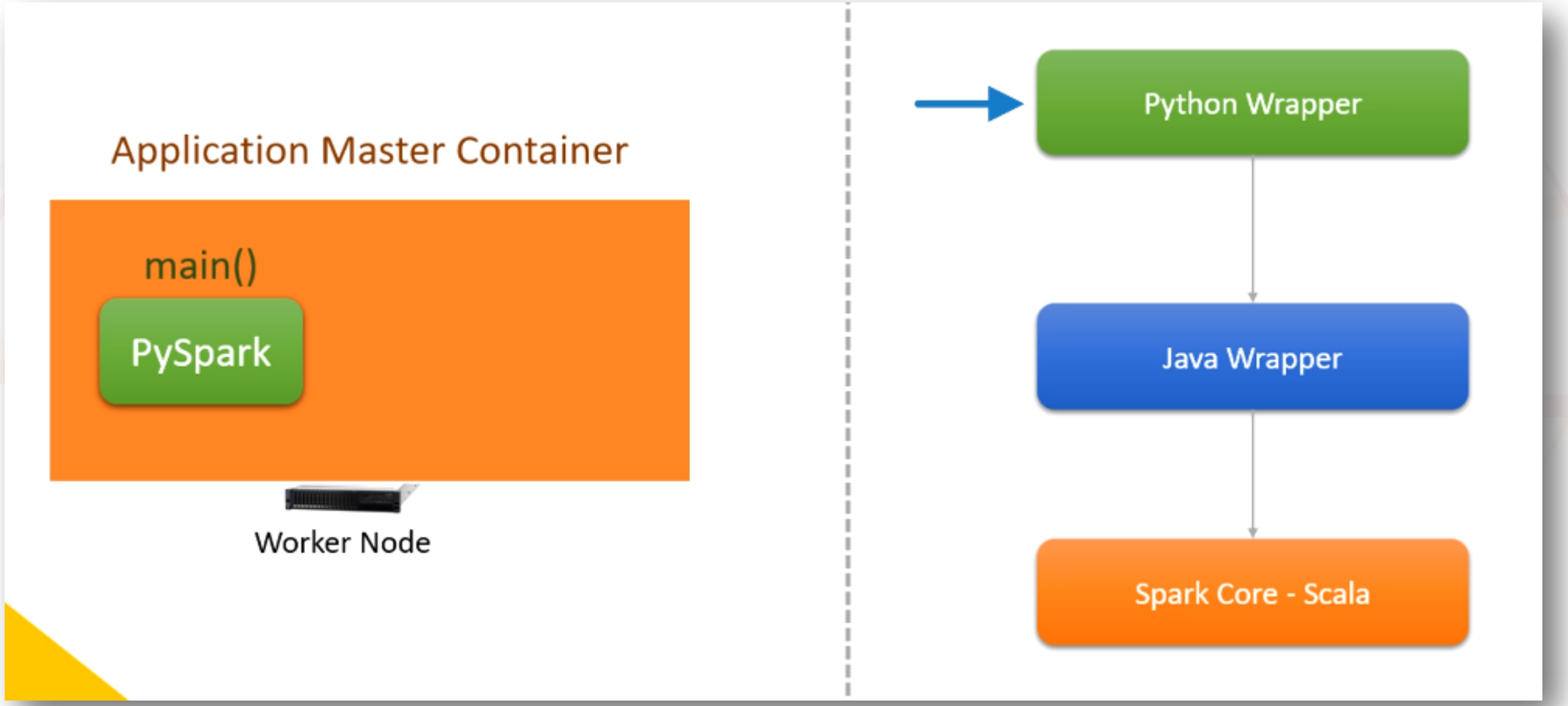
Spark was written in Scala. Scala is a JVM language, and it always runs in the JVM.



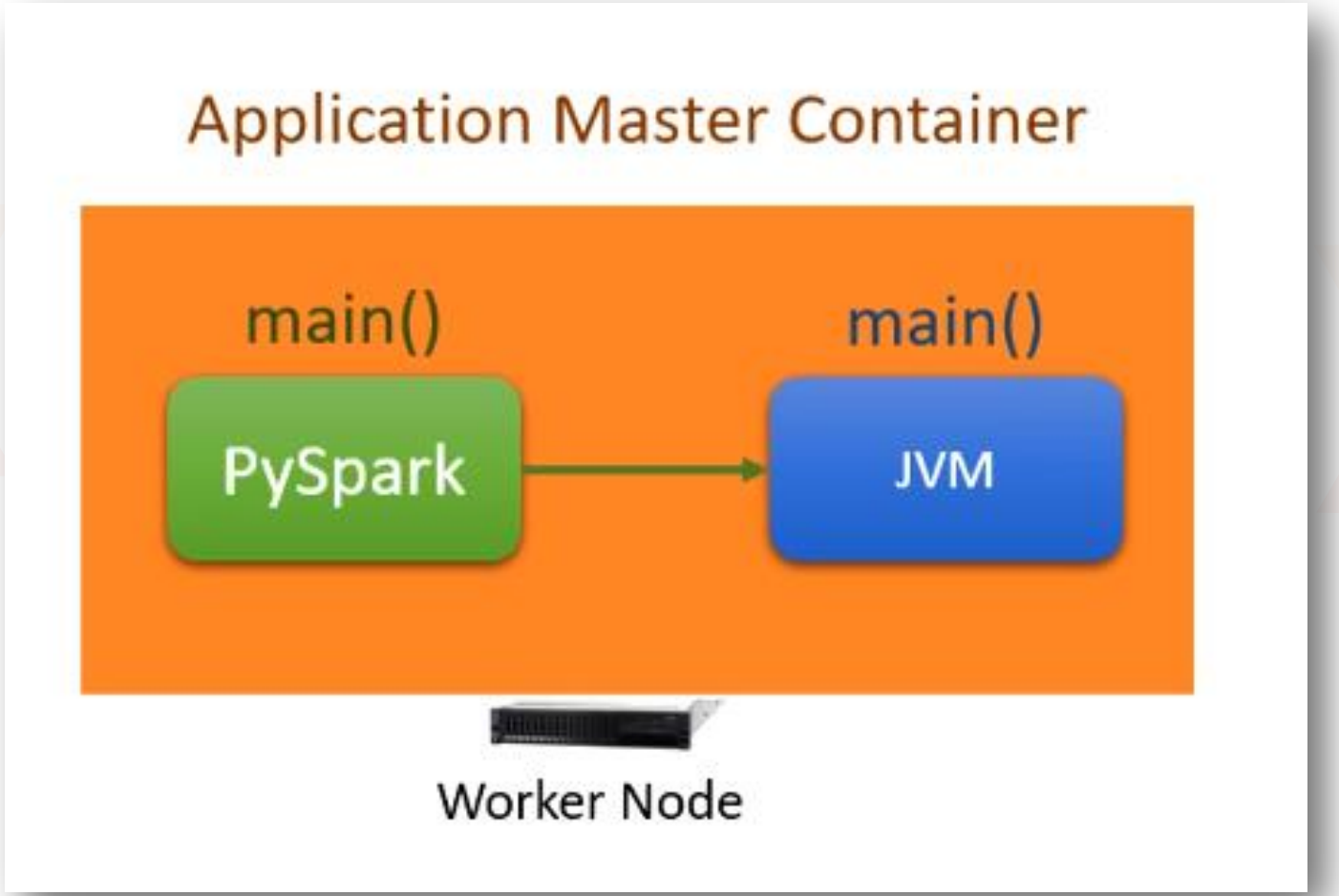
But the Spark developers wanted to bring this to Java and Python developers. So they created a Java wrapper on top of the Scala code.



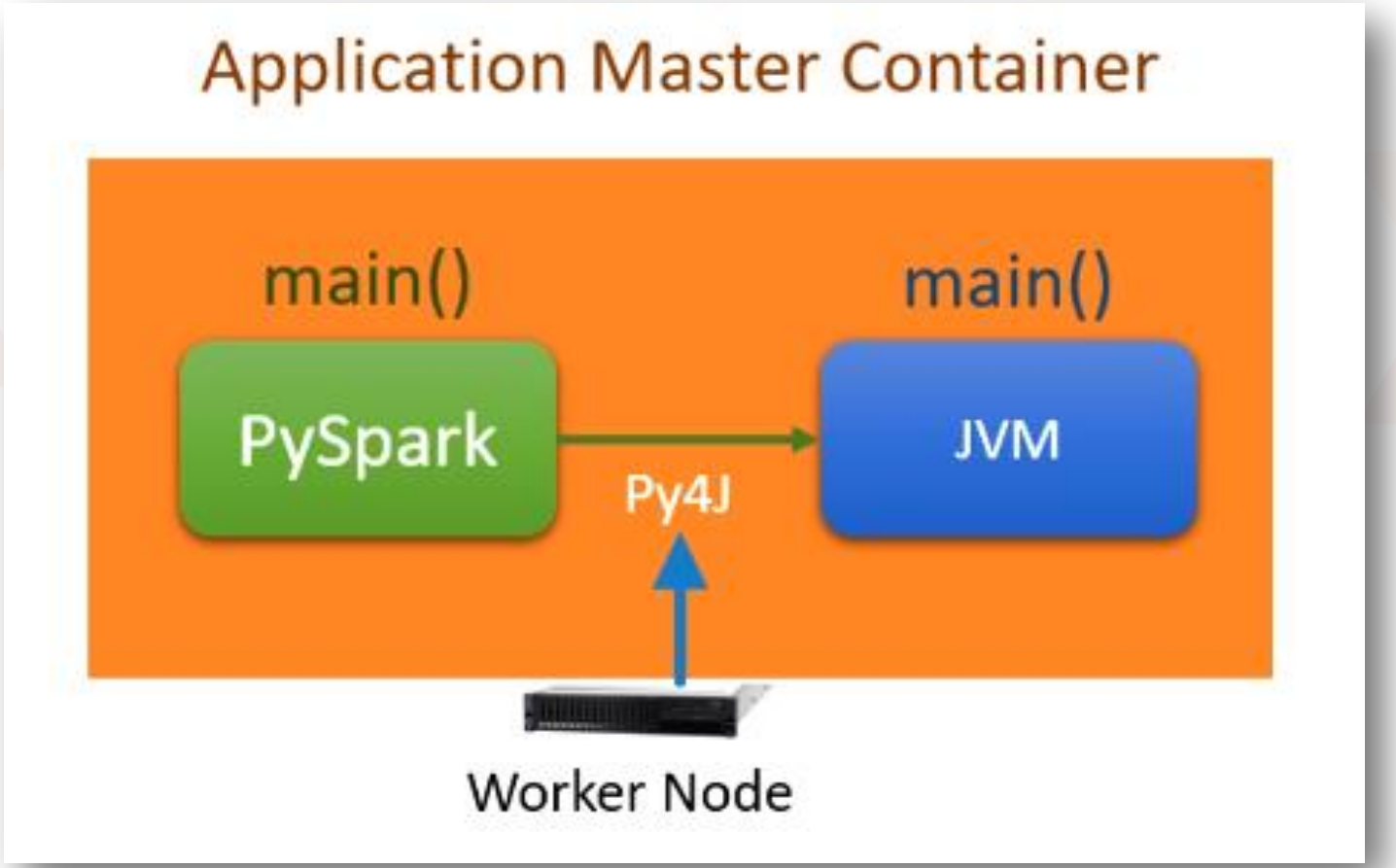
And then, they created a Python wrapper on top of the Java wrappers.
And this Python wrapper is known as PySpark.



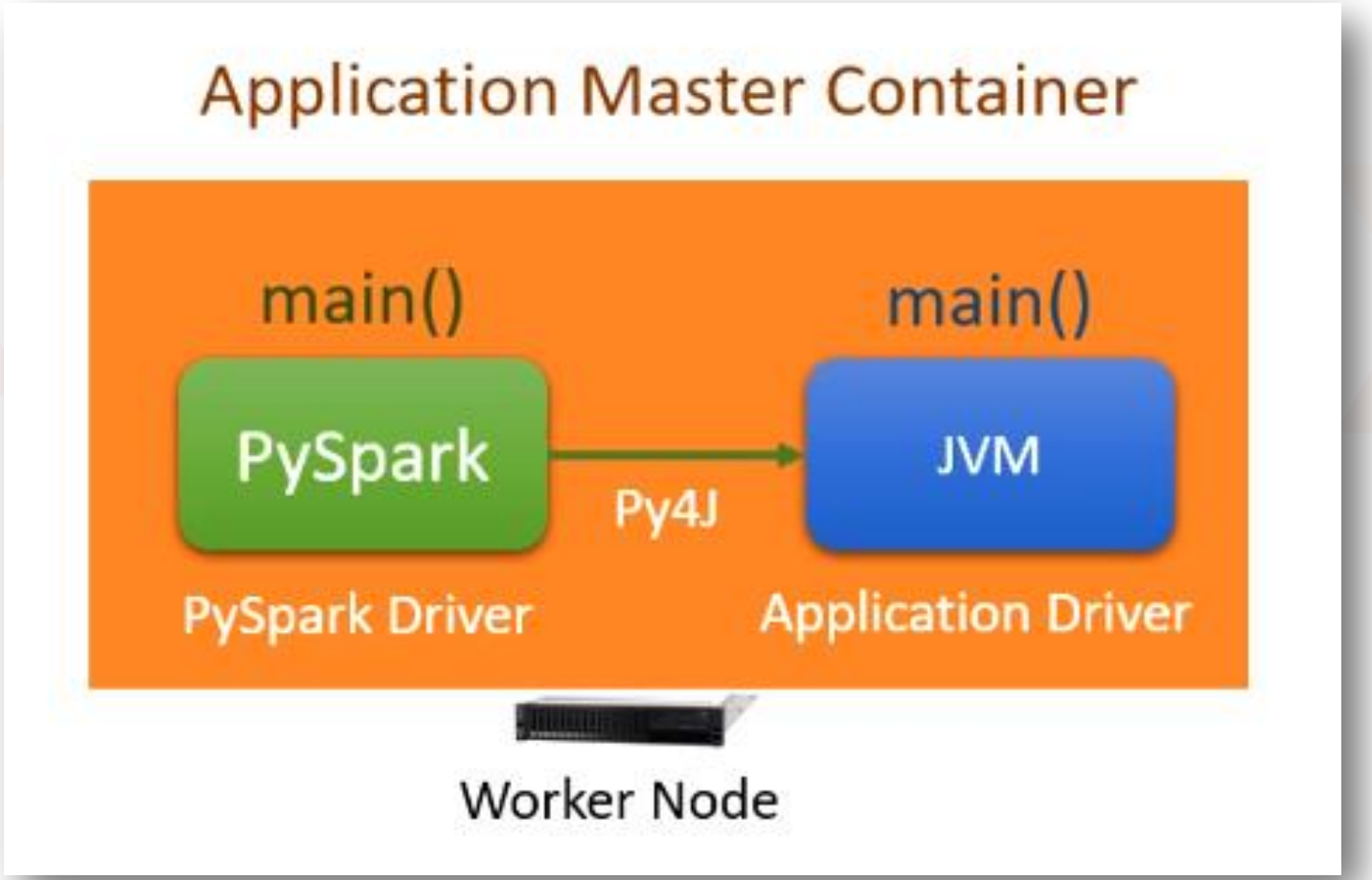
So I have Python code in my main() method.
This python code is designed to internally start a Java main() method.
So my PySpark application will start a JVM application.



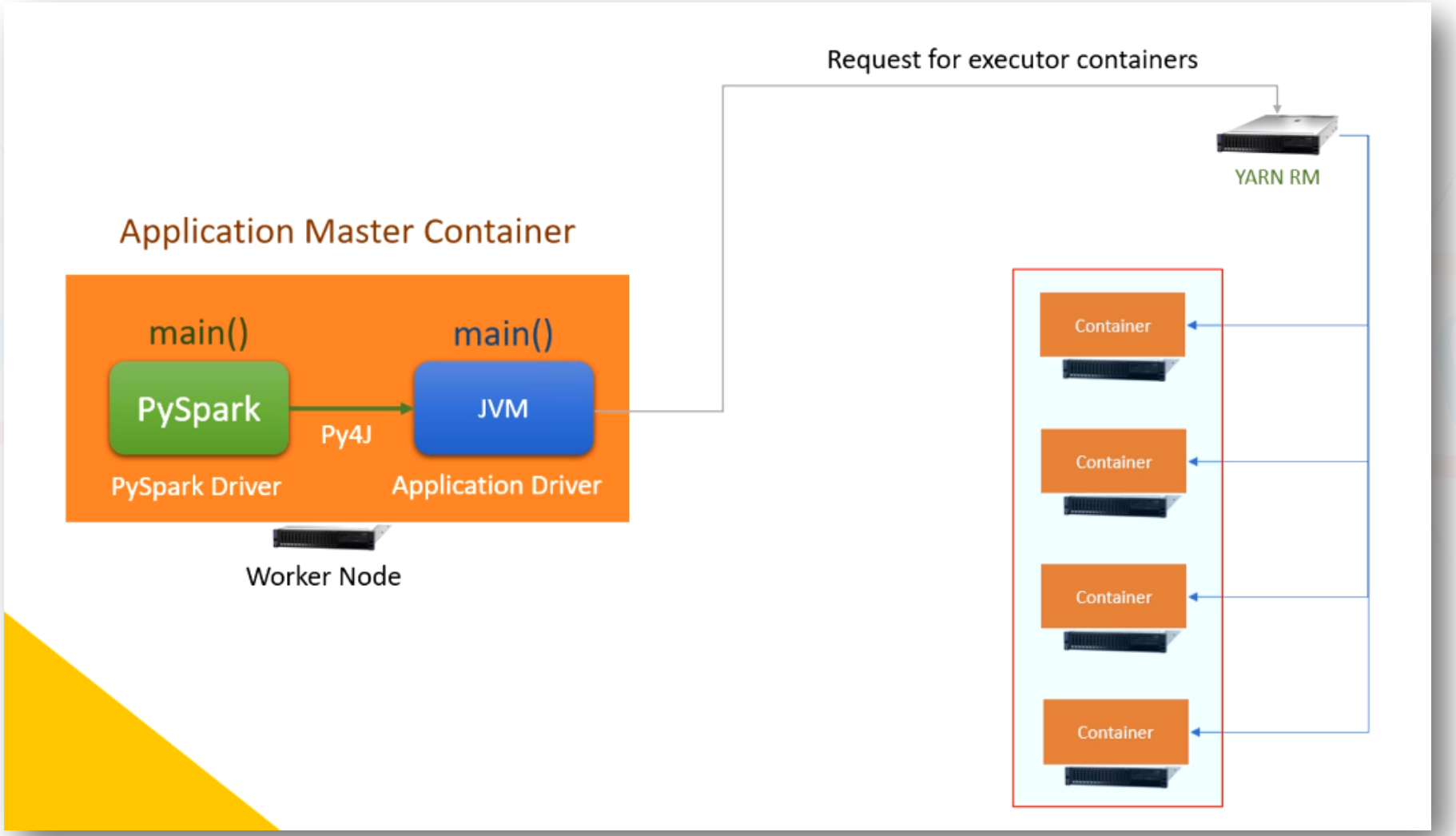
Once we have a JVM application, the PySpark wrapper will call the Java Wrapper using the Py4J connection. What is Py4J? Py4J allows a Python application to call a Java application. And that's how PySpark works. It will always start a JVM application and call Spark APIs in the JVM. The actual Spark application is always a Scala application running in the JVM. But PySpark calls Java Wrapper using Py4J, and the Java Wrapper runs Scala code in the JVM.



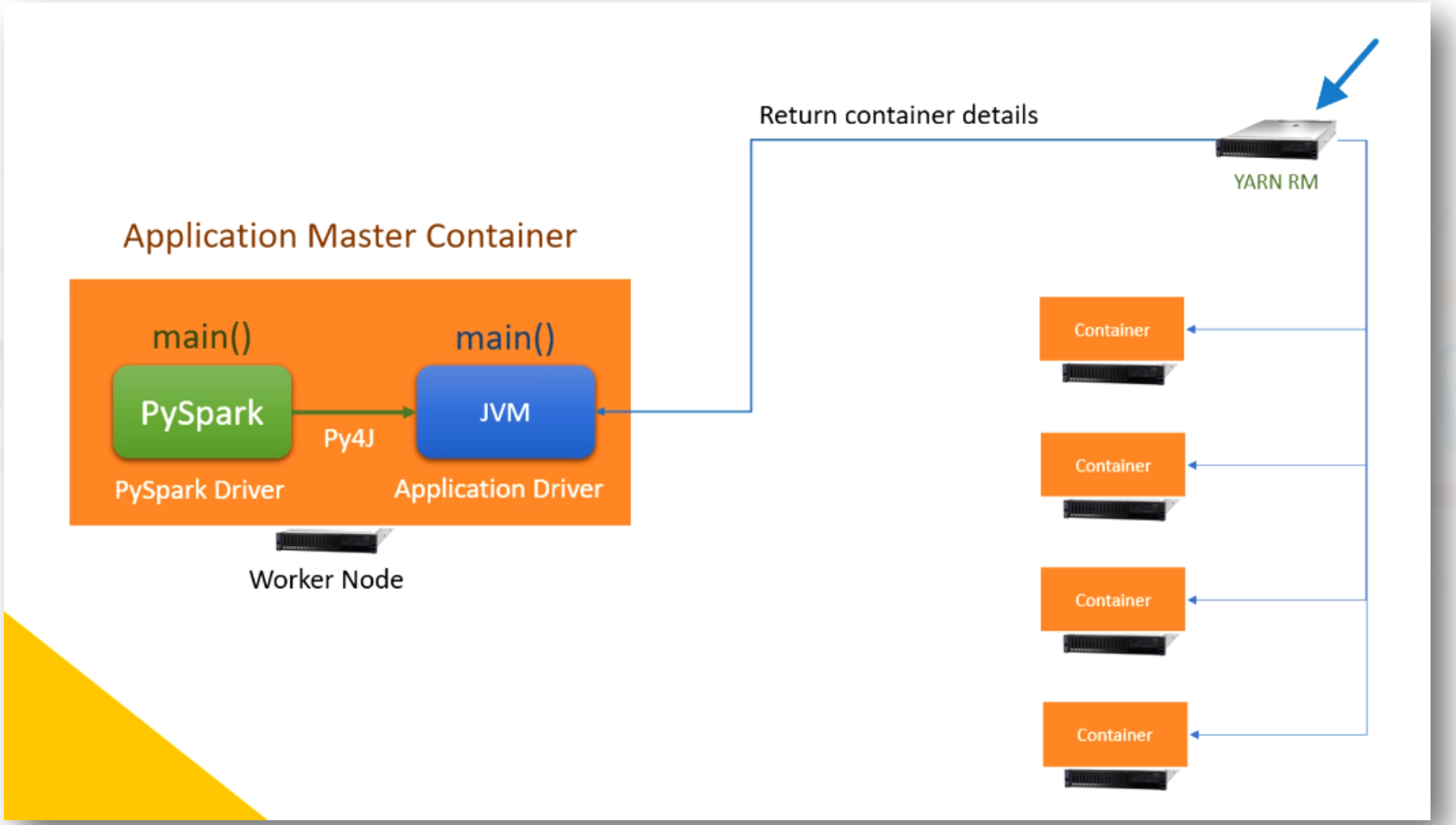
The PySpark main method is my PySpark Driver. And the JVM application here is my Application Driver. These two terms are critical to remember. So your Spark application driver is the main method of your application. If you wrote a PySpark application, you would have a PySpark driver and an application driver. But if you wrote it in Scala, you won't have a PySpark driver, but you will always have an application Driver.



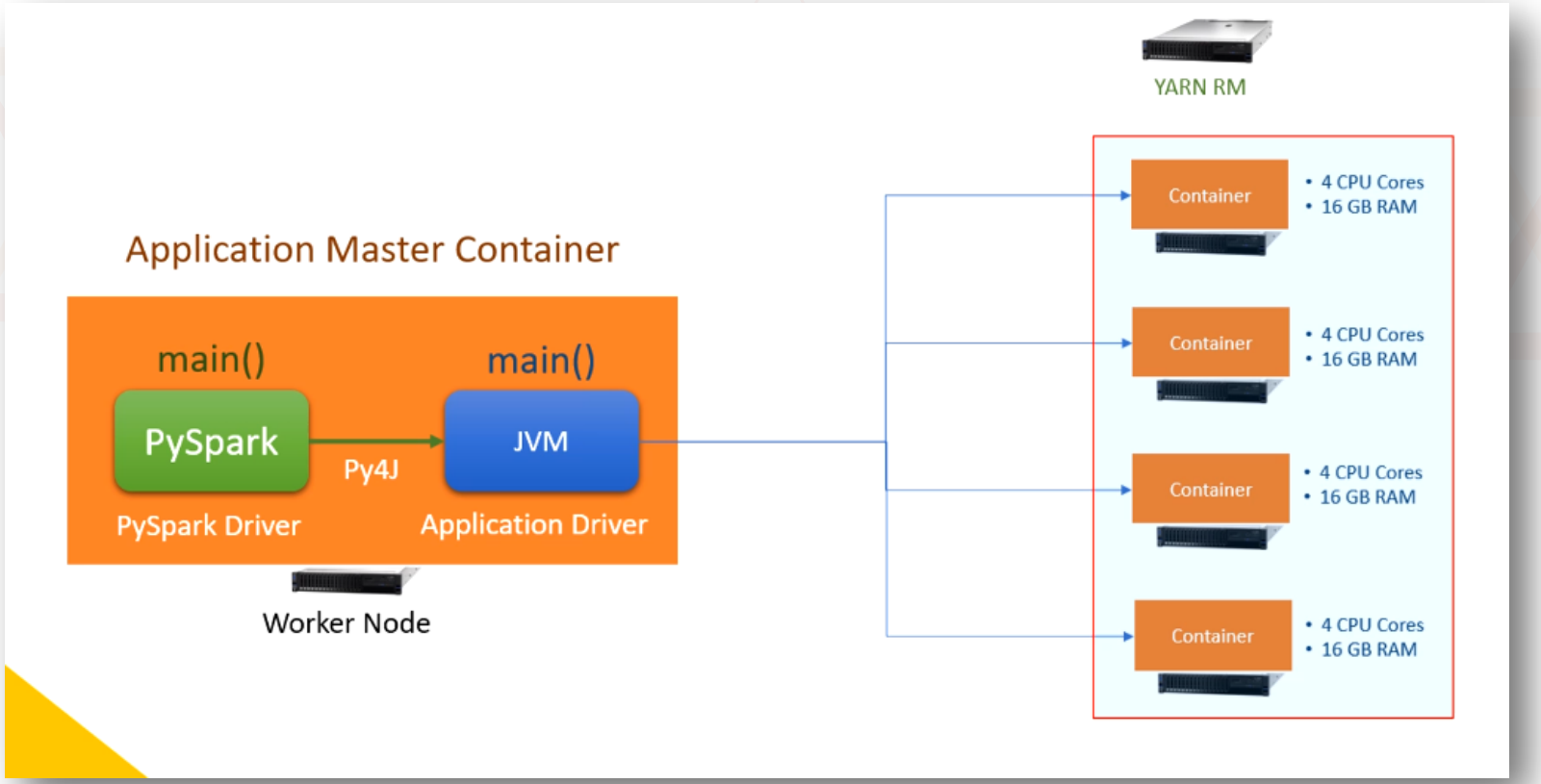
Spark application driver does not perform the data processing work. Instead, it will hire executors and distribute the work to them. Remember that. The driver does not perform any data processing work. Instead, it will create some executors and get the work done from them.



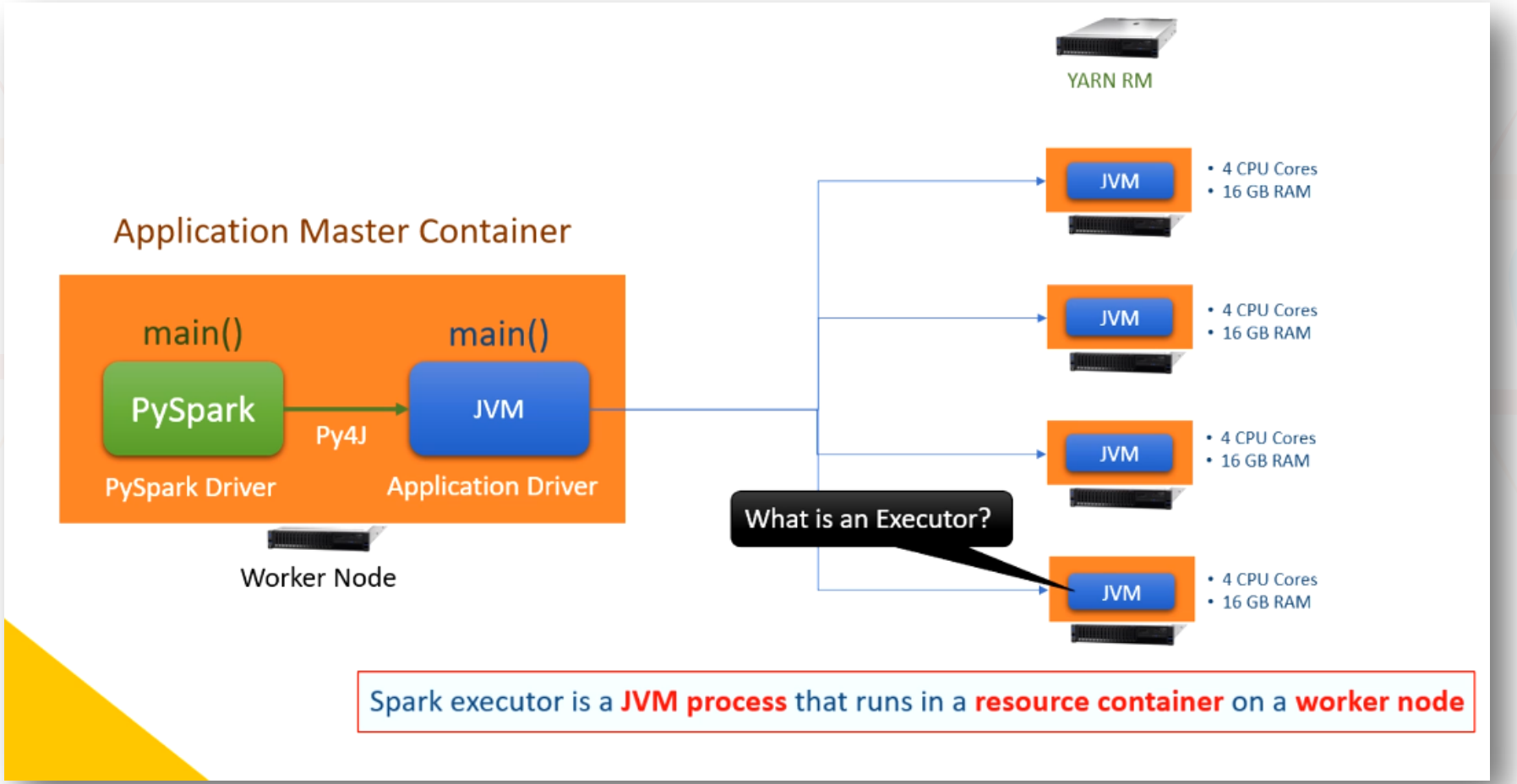
After starting, the driver will go back to the Cluster RM and ask for some more resource containers.



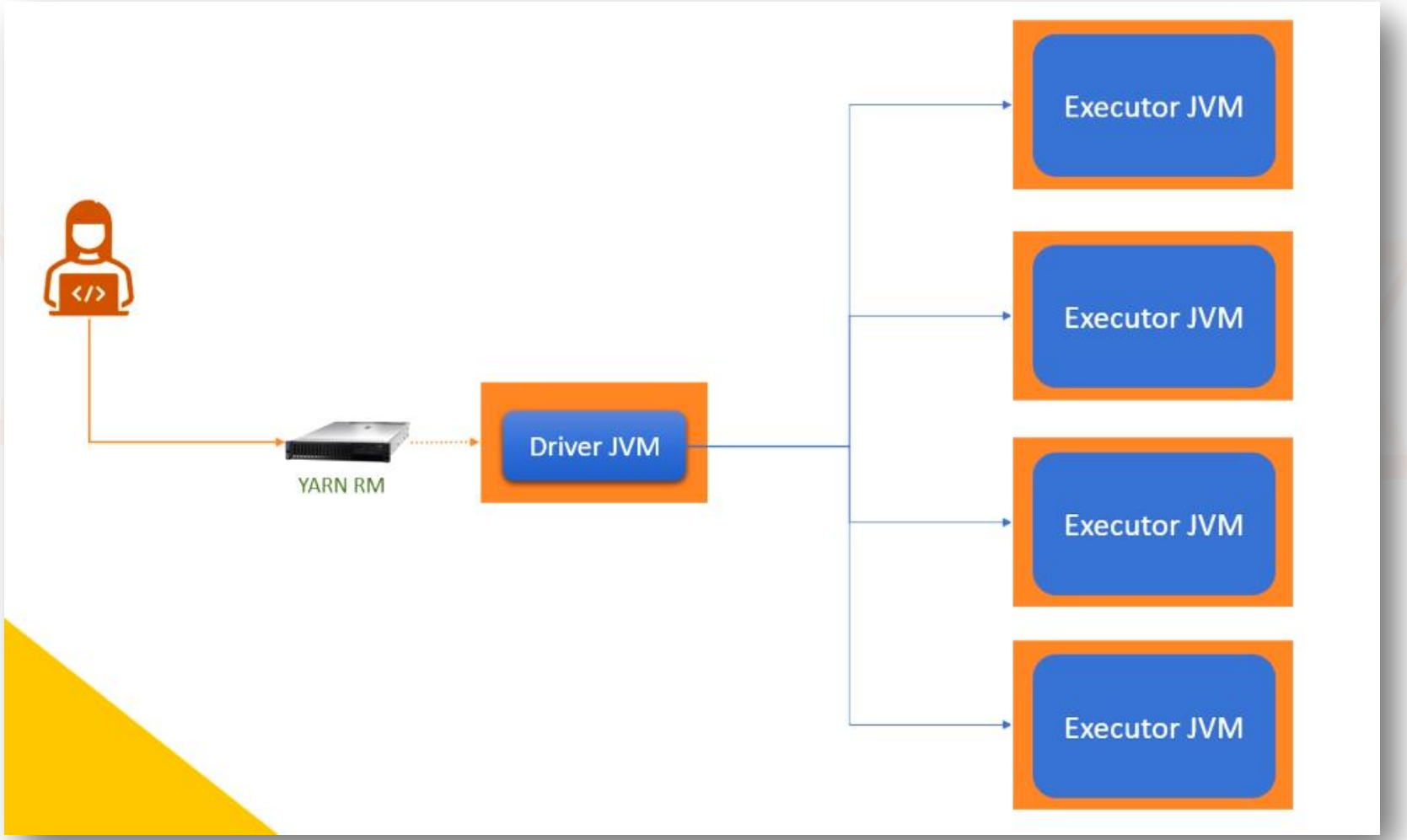
The RM will create some more containers on worker nodes and give them to the driver. So let's assume we got four new containers. And let's assume each container comes with 4 CPU Cores and 16 GB of memory. Now the driver will start the spark executor process in these containers. Each container will run one Spark executor, and the Spark executor is a JVM application. So your driver is a JVM application, and your executor is a JVM application. These executors are responsible for doing all the data processing work.



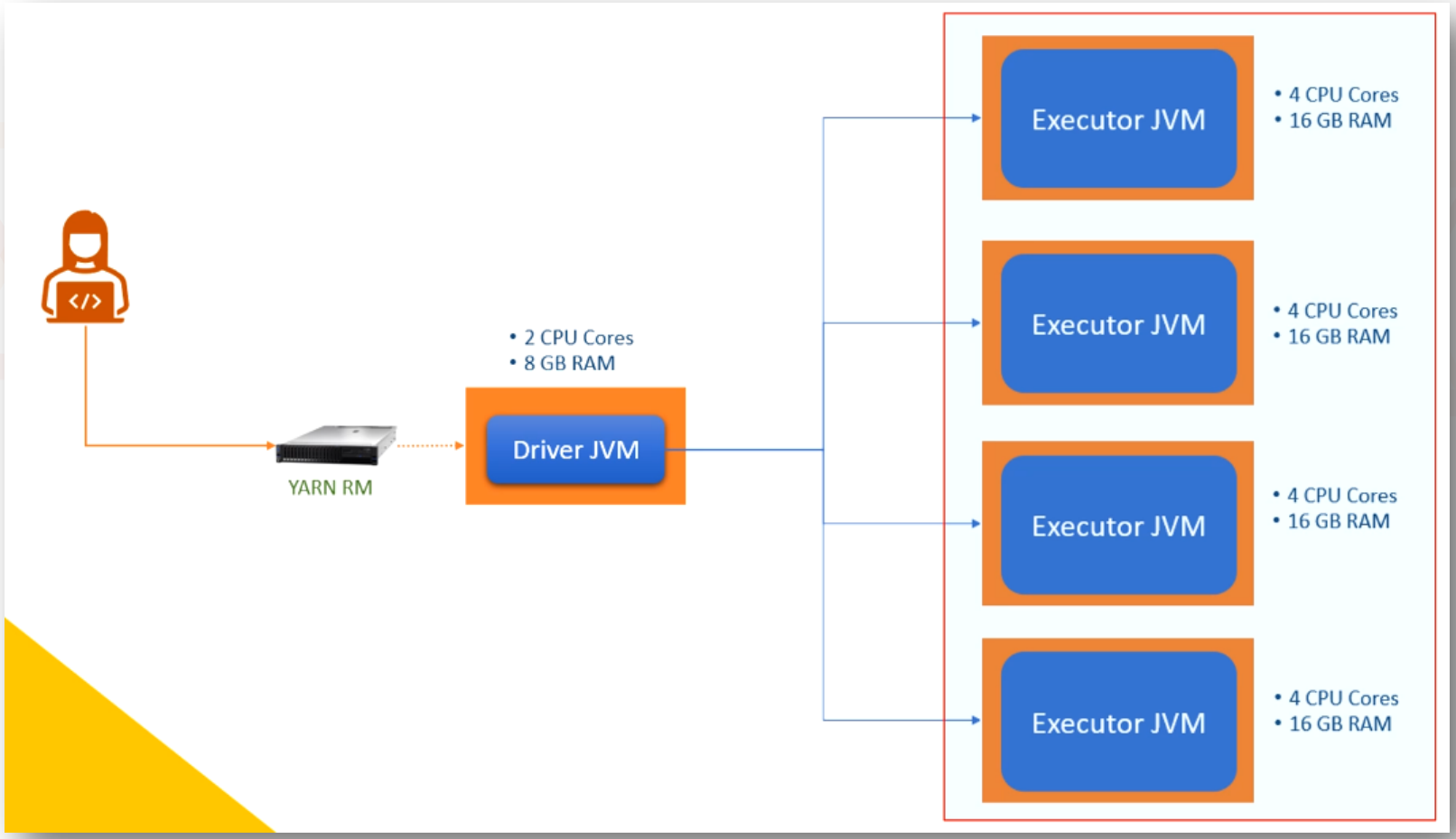
Spark executor is a JVM process that runs in a resource container on a worker node.
We have three different things here: Executor, Resource Container, and Worker.
The executor is a JVM process that will do the data processing work. The executor runs inside a resource container. And the resource container is assigned on a worker node.



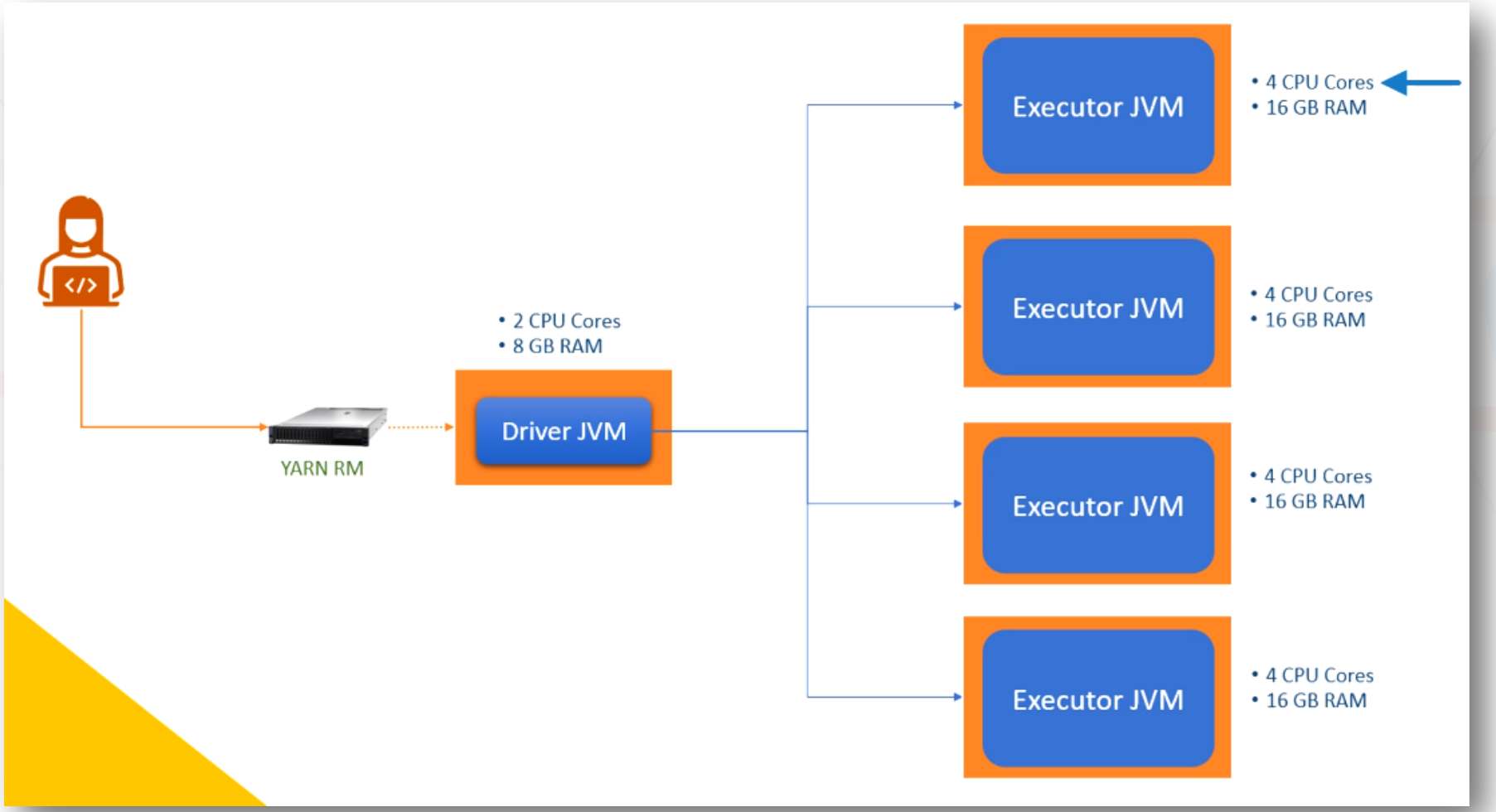
So, we have one driver and some executors. And that's a typical Spark application architecture. Every Spark application ends up creating one driver and one or more executors. And they will work together to do the data processing.



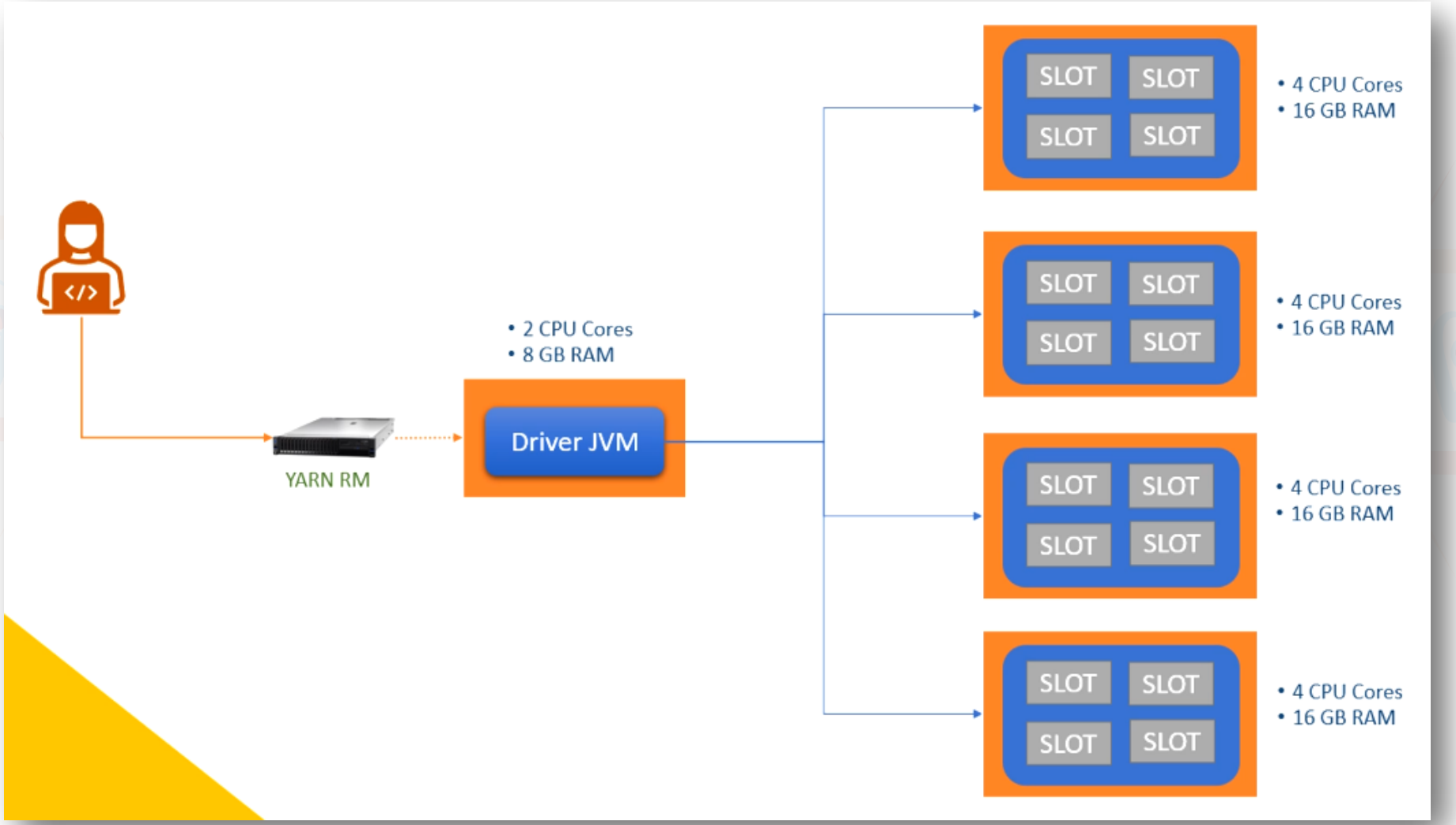
So we have one driver and a bunch of executors. Both of these processes run in a resource container. We gave 2 CPU cores and 8 GB RAM to the drive container. So the driver cannot use more than that. Similarly, each executor has 4 CPU Cores and 16 GB of RAM. And the executor cannot use more than that. The driver is a normal process, and it doesn't do a lot of work. Most of the work is done by the executor.



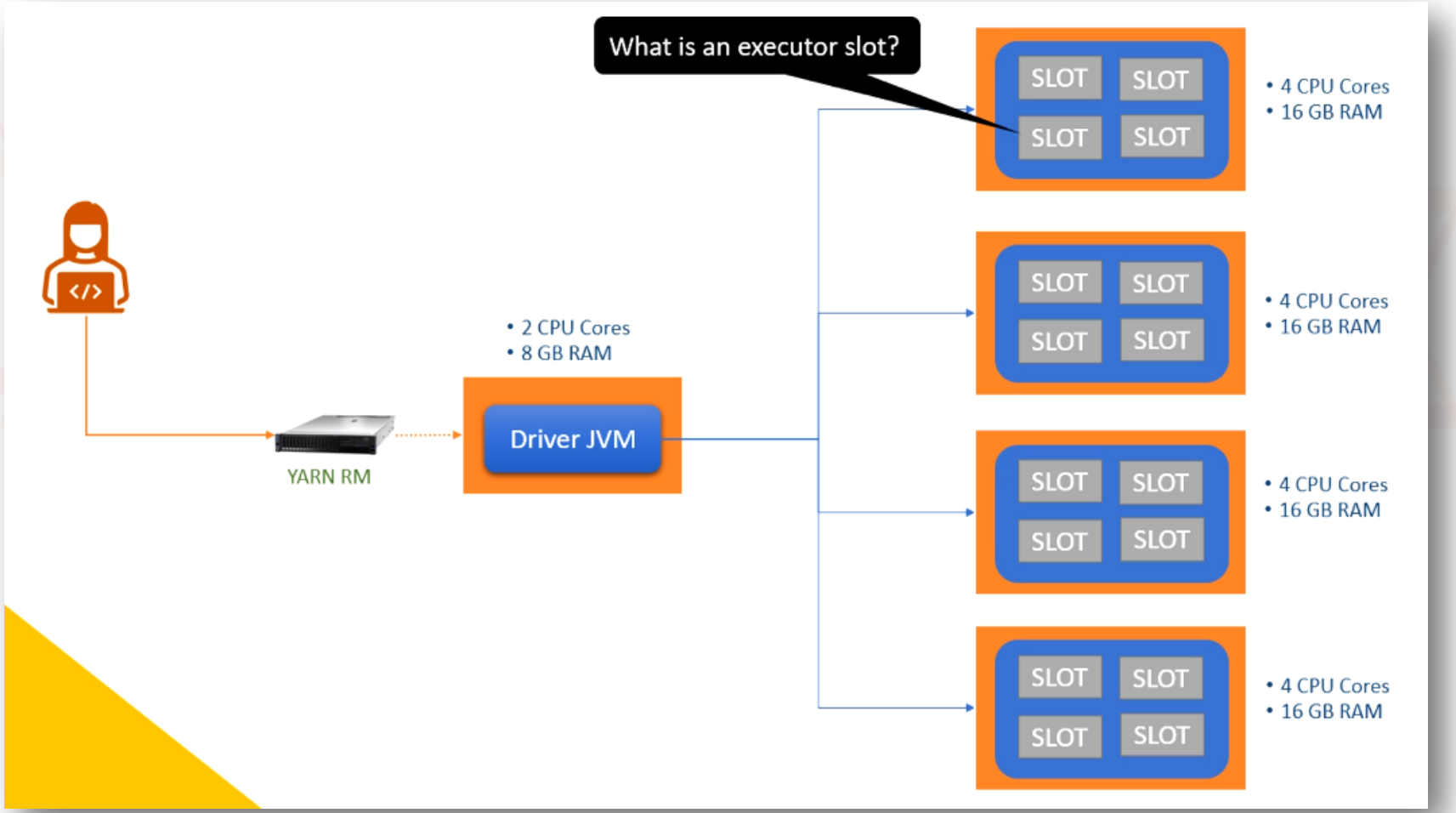
Now, let's drill the executor container to the next level. In this example, the executor container is given 4 CPU cores. So the executor JVM process will start with four parallel threads.
Why four? Because we gave 4 CPU cores to each executor. So it can create four parallel threads.
And that's the slot capacity of my executor.



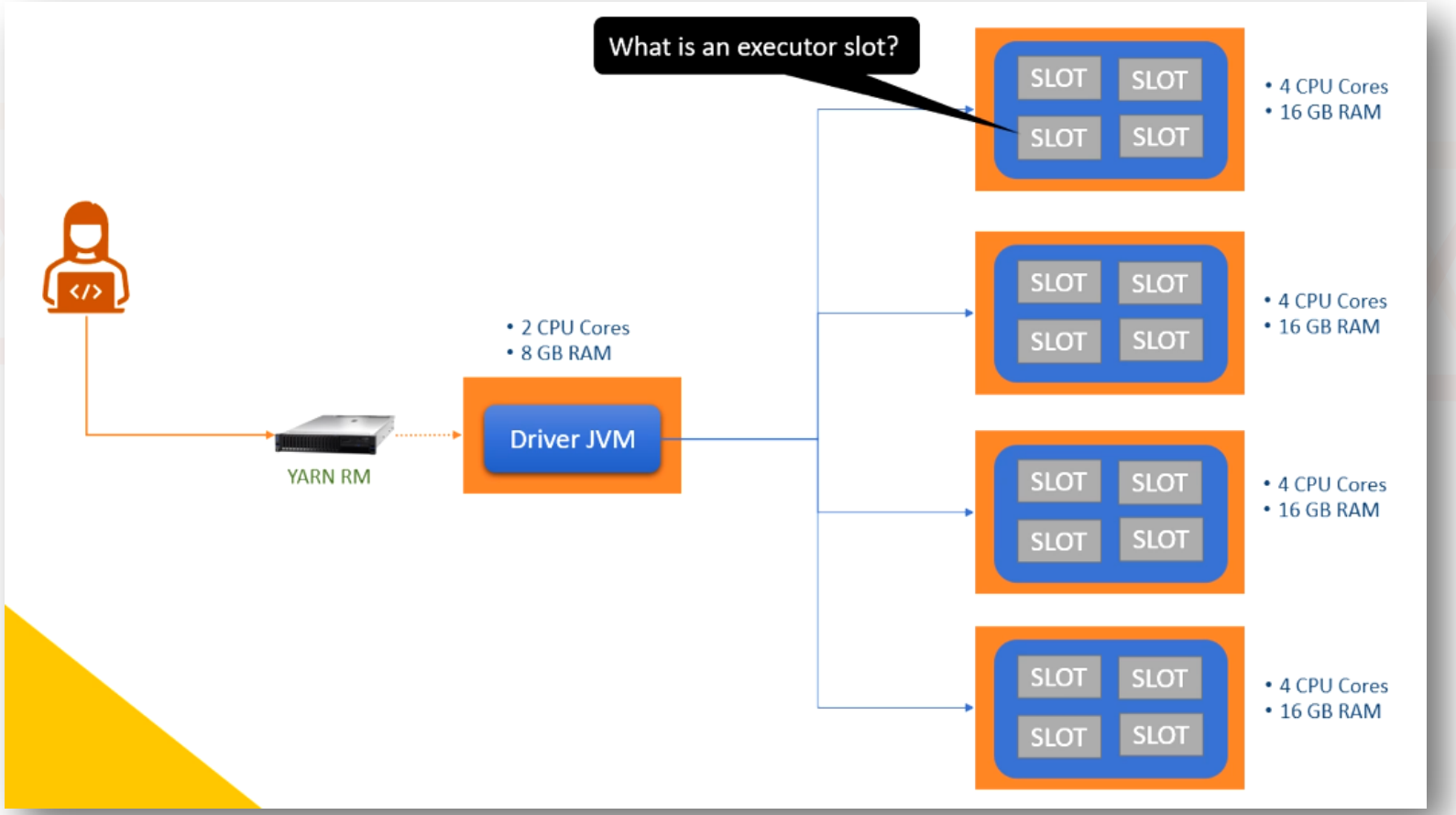
Here is how slots capacity looks like in executors.
So each executor can have four parallel threads, and we call them executor slots. The driver knows how many slots we have at each executor.



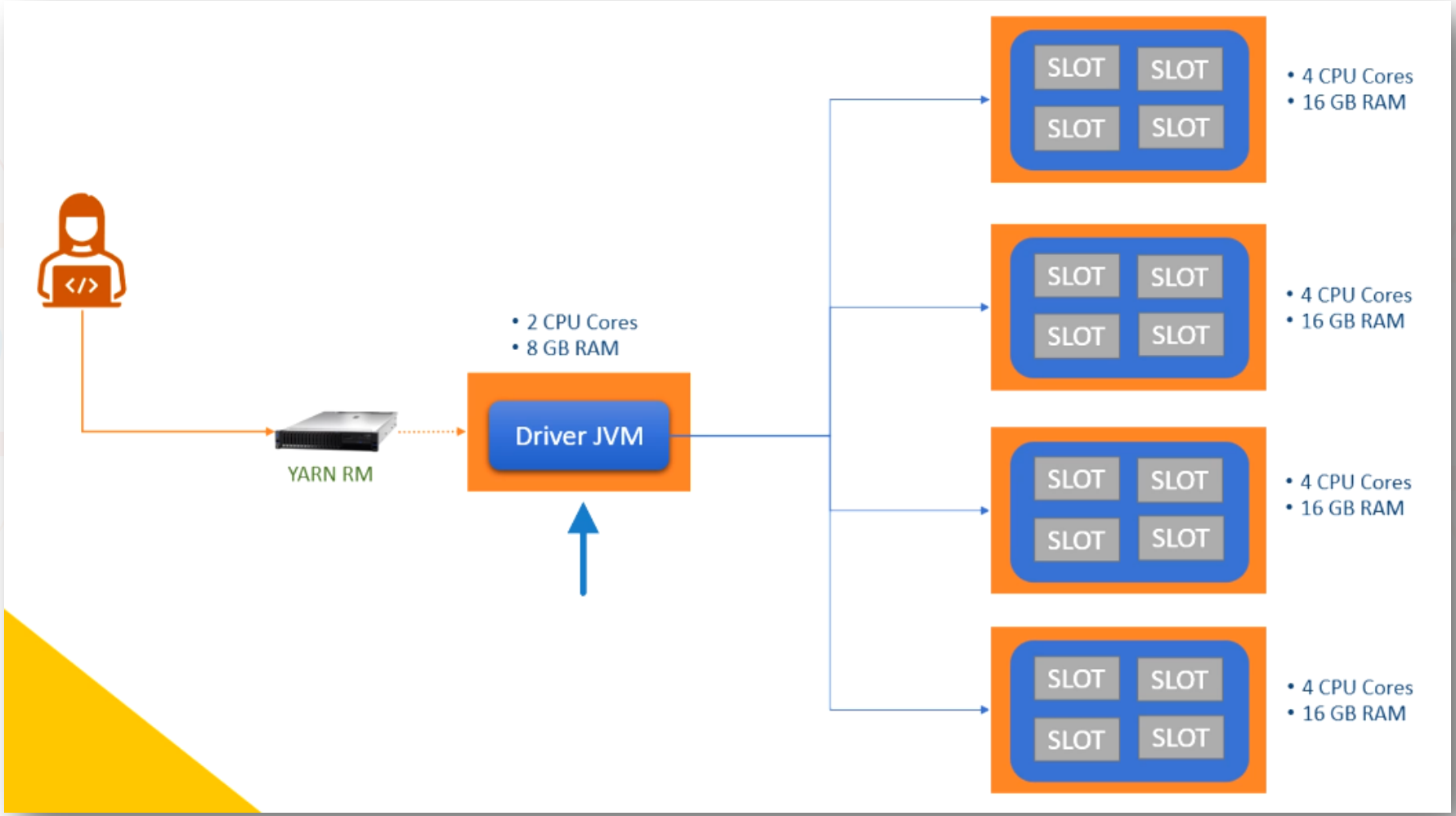
Executor Slot is a parallel thread inside the executor process. The driver knows all the information. I mean, It knows how many executors are there, and how many slots are there for each executor. The executors and the slots are the one side of the equation. Each slot can process one data partition at a time. And the slots are where all the data processing work happens.



But what comes into the slot? The driver knows that also.
I mean, the driver knows both sides of the equation. It knows how many slots are there and where they are. And the driver also knows what task I need to perform, so it will assign tasks in the slots. Now let's try understanding another side of the equation and quickly understand the task that fits in these slots.

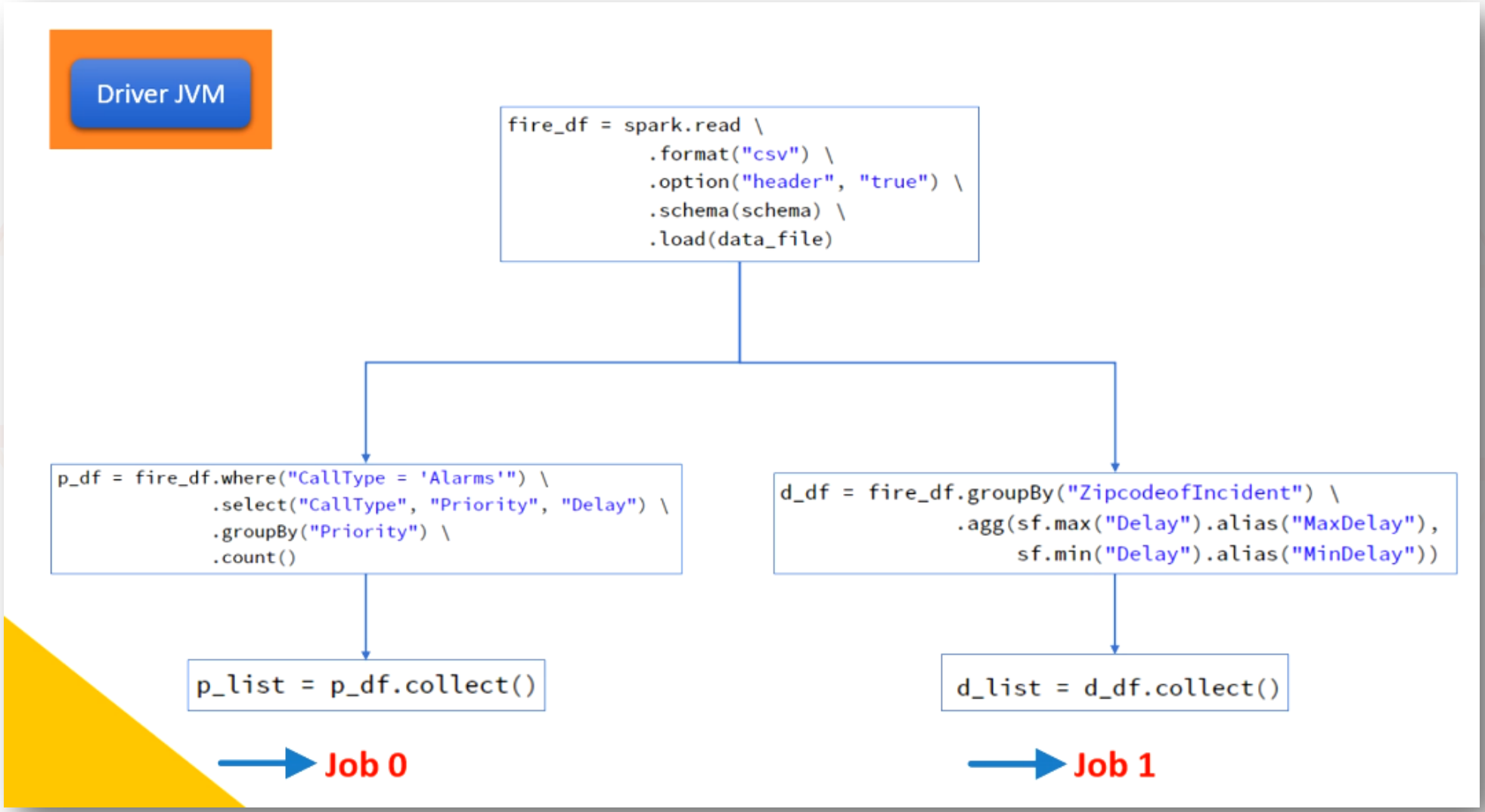


The driver is the main method.
So all your PySpark code is also there at the driver.

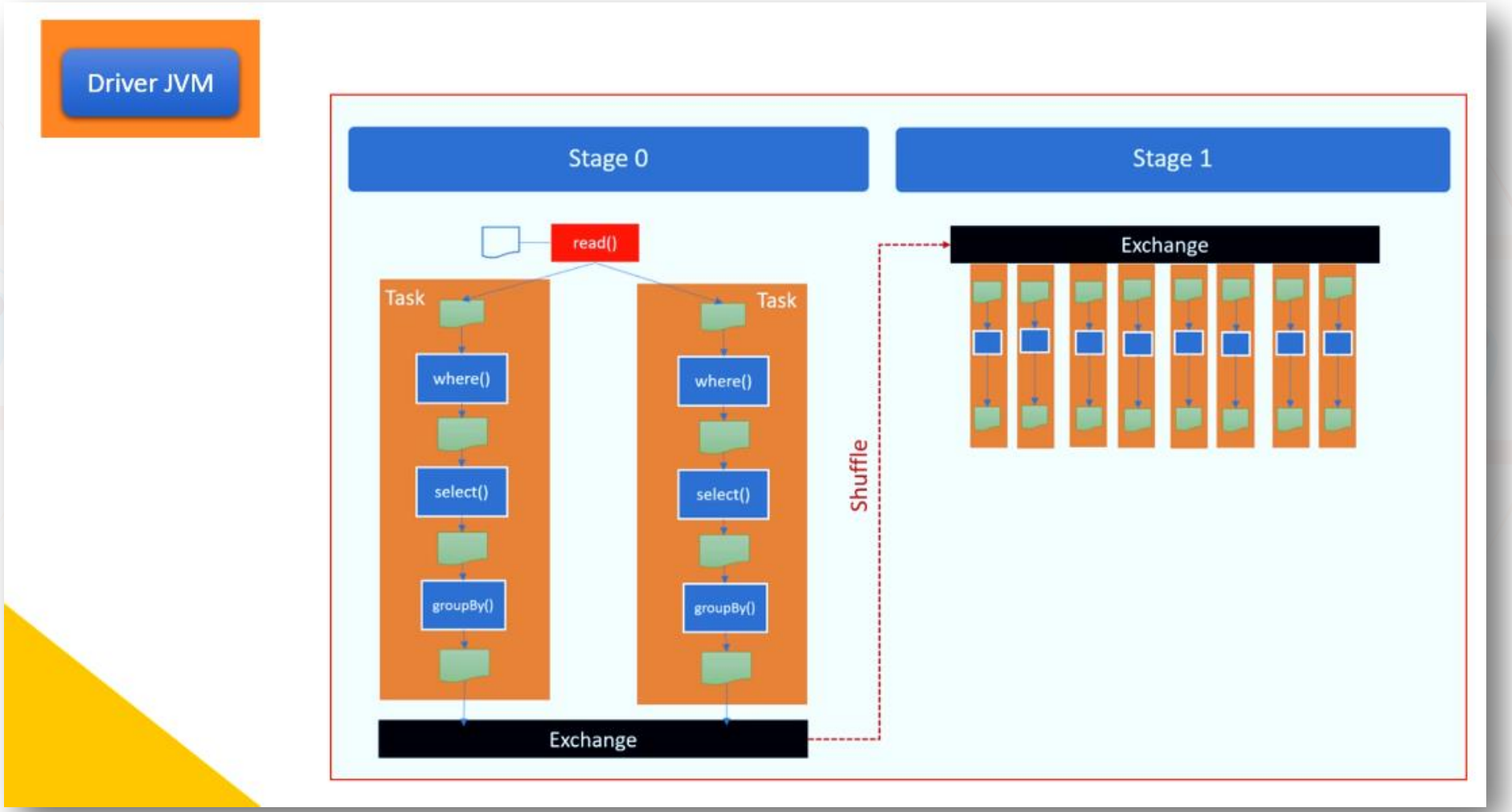


Let's assume you have the following PySpark code.
Now, the driver will analyze your PySpark code.

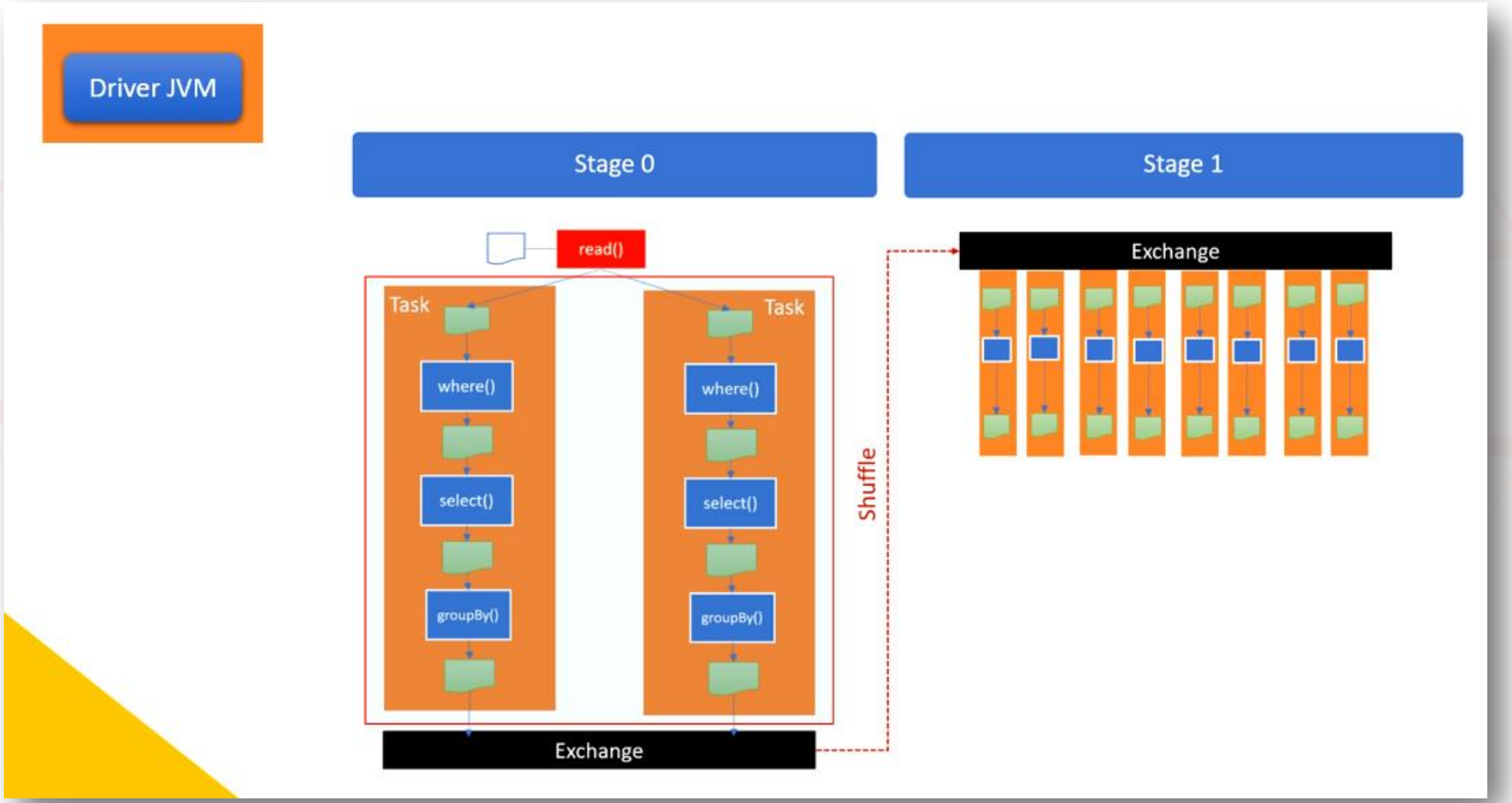
It will create an inverted tree of your code and figure out that we need to run two Spark jobs. Then it will compile the code for the first job and create an execution plan.



The plan looks like this. And this is known as Spark Job DAG. A Job DAG is nothing but an execution plan or an explanation plan. And it tells the driver about the stages and tasks. Looking at this DAG, the driver knows how I can finish this Job. I will run stage 0, and then I will run stage 1.

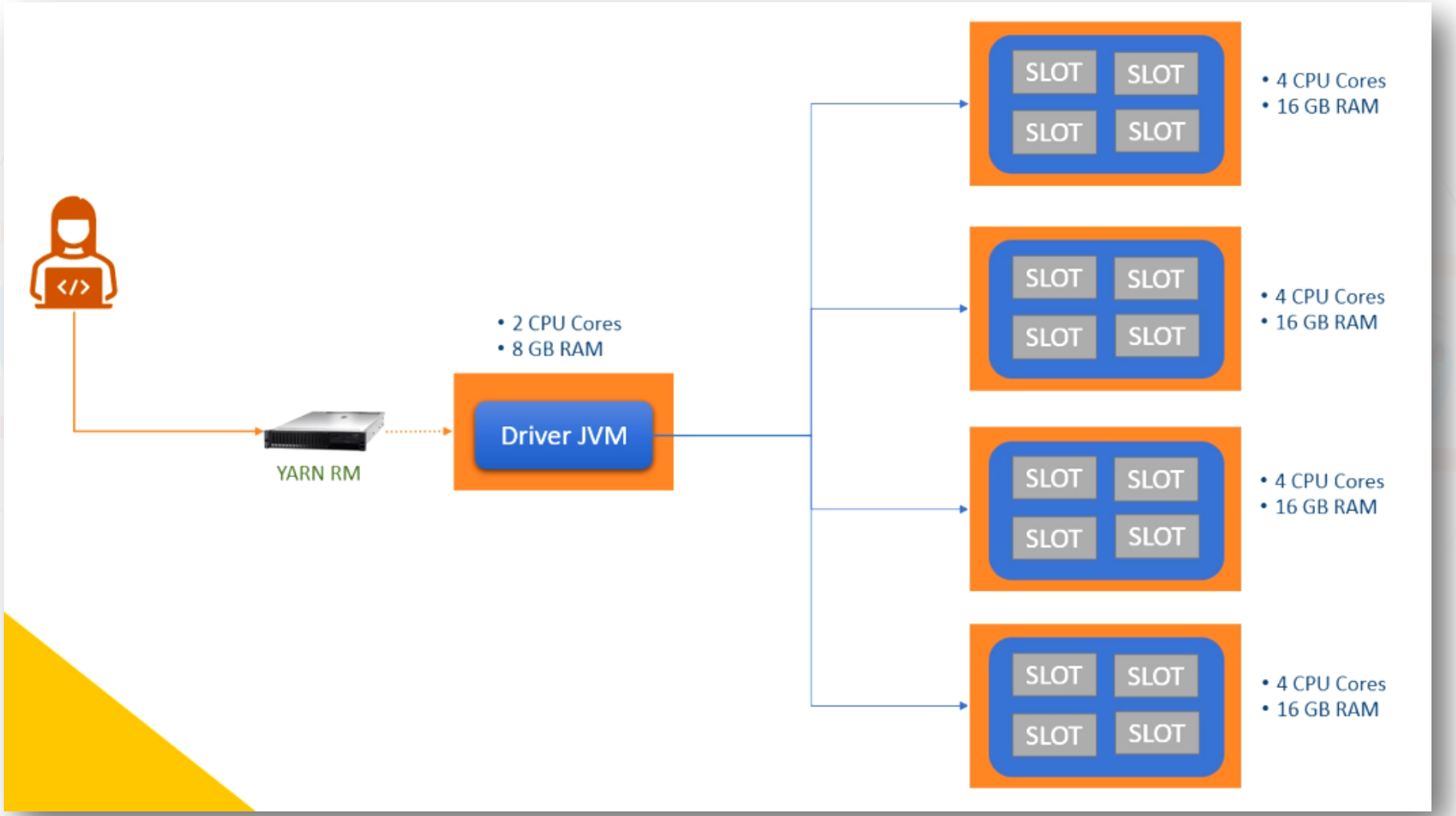


Stage zero can run two parallel tasks. Why two? Because the stage zero input data has only two partitions. So we can run only two parallel tasks.
The driver will assign these two tasks in some executor slots.

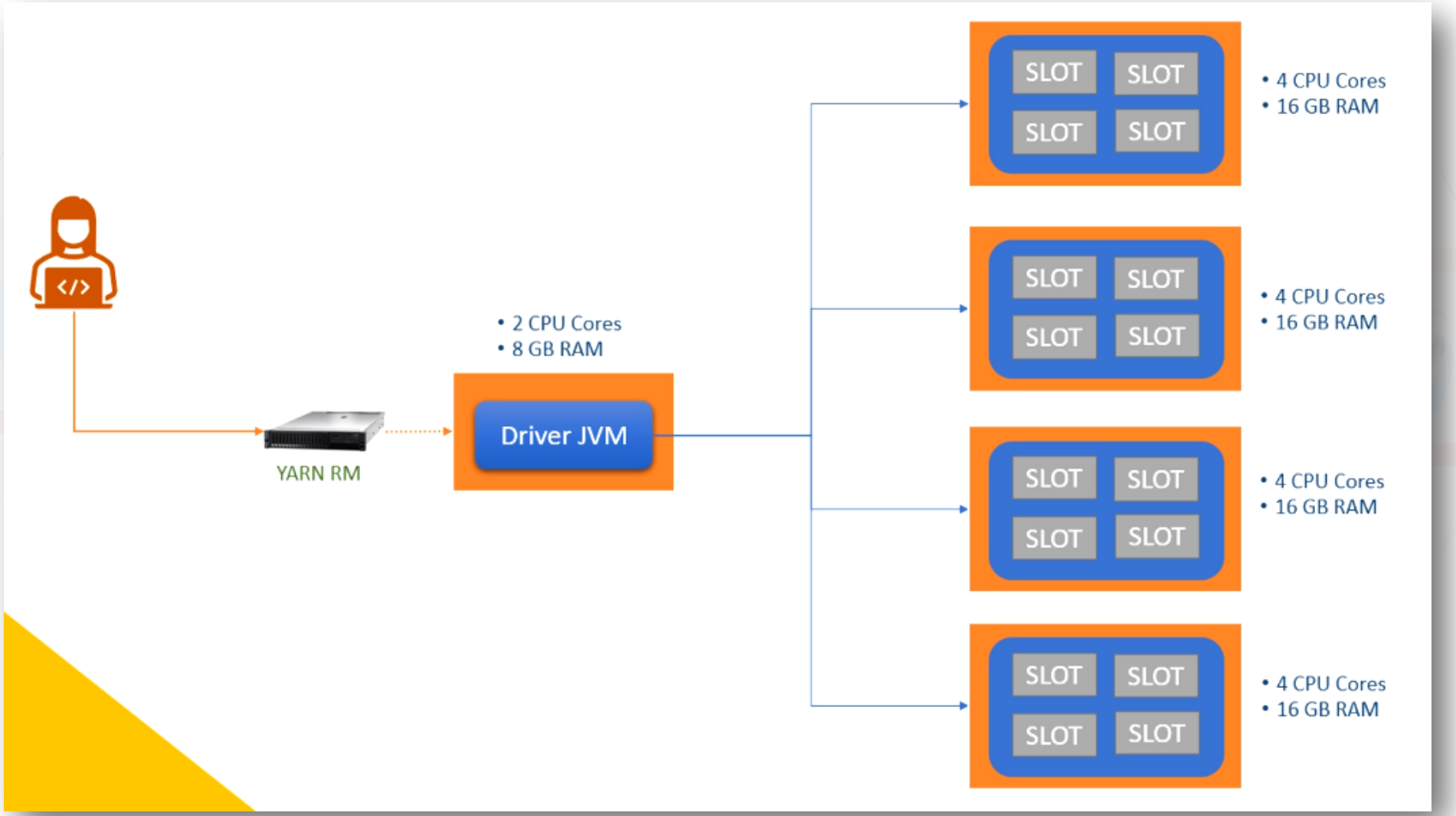


The task from the execution plan fits in the executor slots.
The executor slot is the smallest unit of execution.
And the task is the smallest unit of work.
So the Spark driver assigns the task in the executor slot.

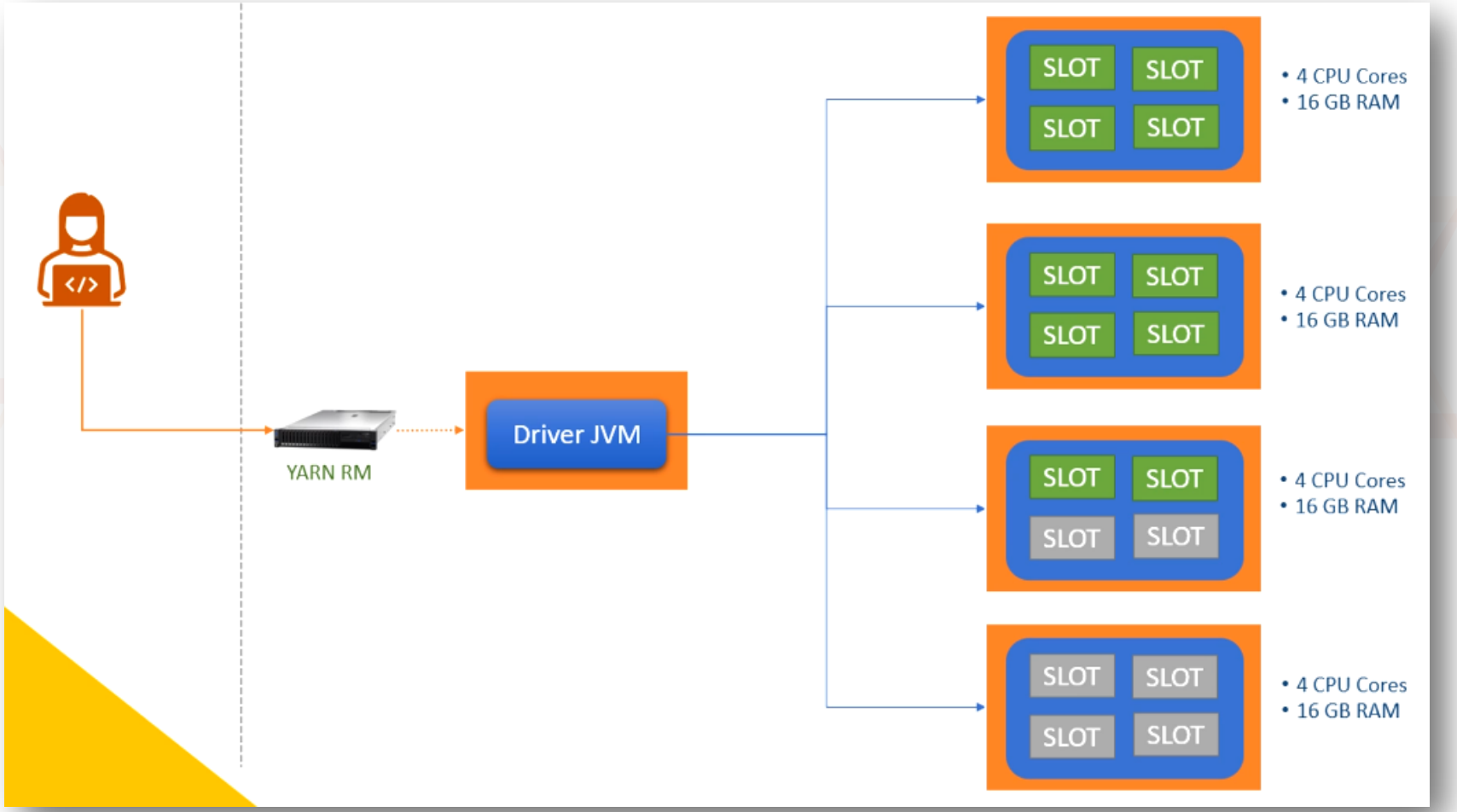
Let me quickly recap. Here is my spark cluster. I started a Spark application and allocated four executors. I have 4 CPU cores and 16 GB of memory for all my executors. So now I have 16 slots.



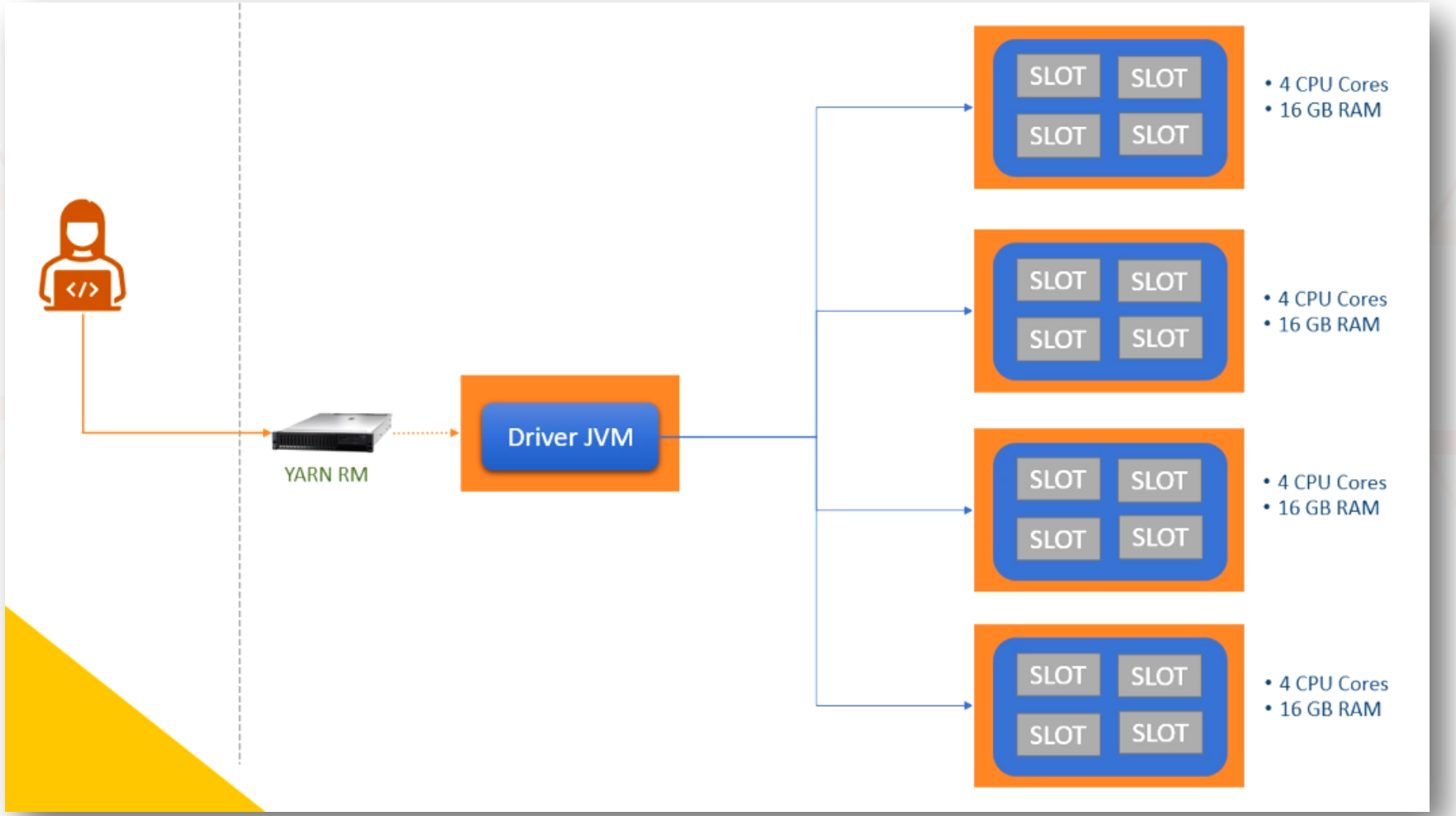
The drive knows how many slots we have at each executor. So for this configuration, let's assume the driver has a stage to finish. And you have ten input partitions for the stage. So you can have ten parallel tasks for the same.



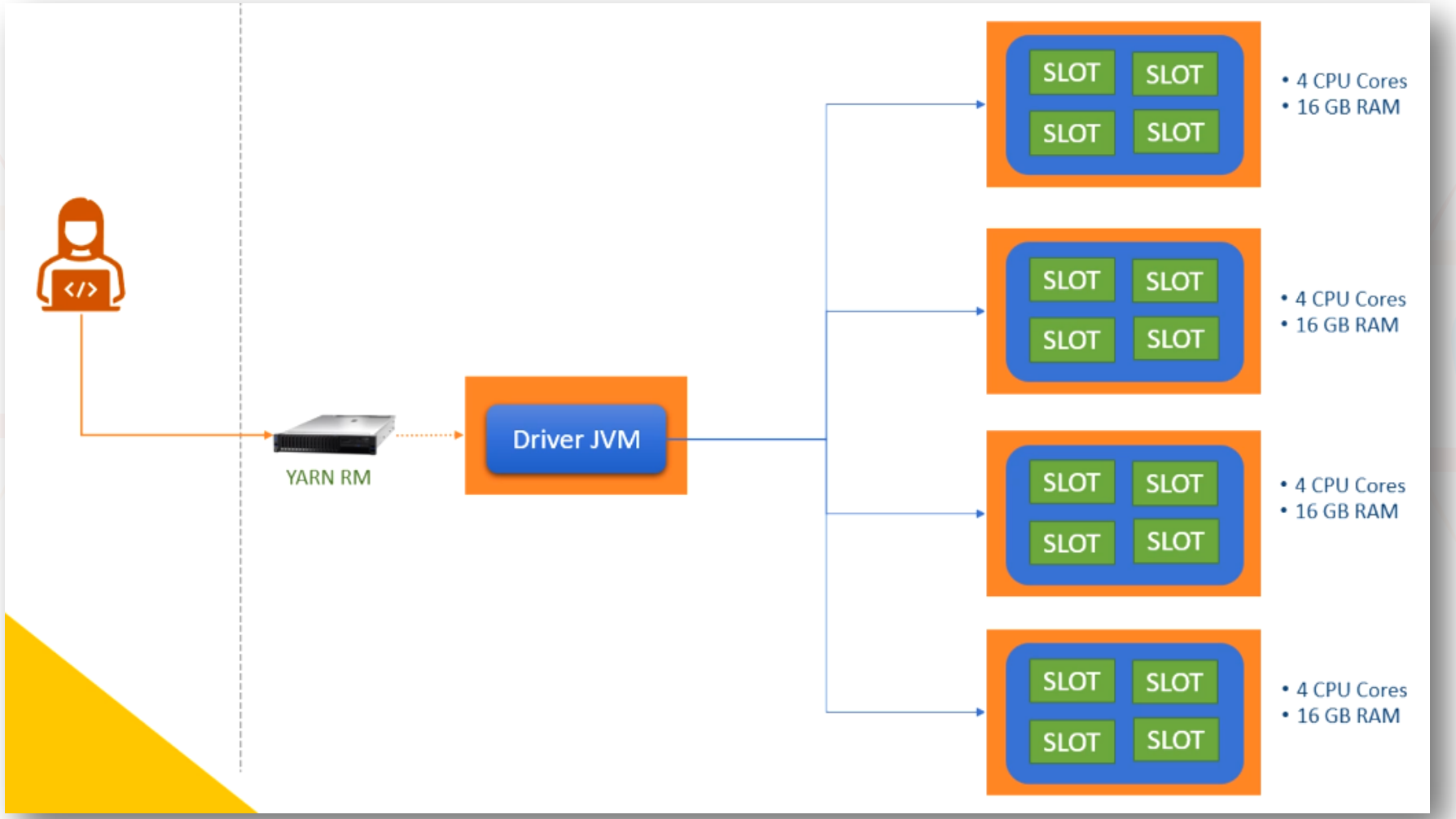
Now the driver will assign those ten tasks in these slots, and it might look like the one shown in image below. All the green slots are filled. This assignment may not be as uniform as I showed here. But let's assume it is as shown here. But we have some extra capacity that we are wasting because we do not have enough tasks for this stage.



Now let's assume this stage is complete. All slots are now free.
Now the driver should start the next stage. And we have 32 tasks for the next stage. But we have 16 slots only.



This won’t be a problem for the driver. The driver will schedule 16 tasks in the available slots. The remaining 16 will wait for slots to become available again. And that's how these tasks are assigned and run by the executor.



So, we started with the following two lists:

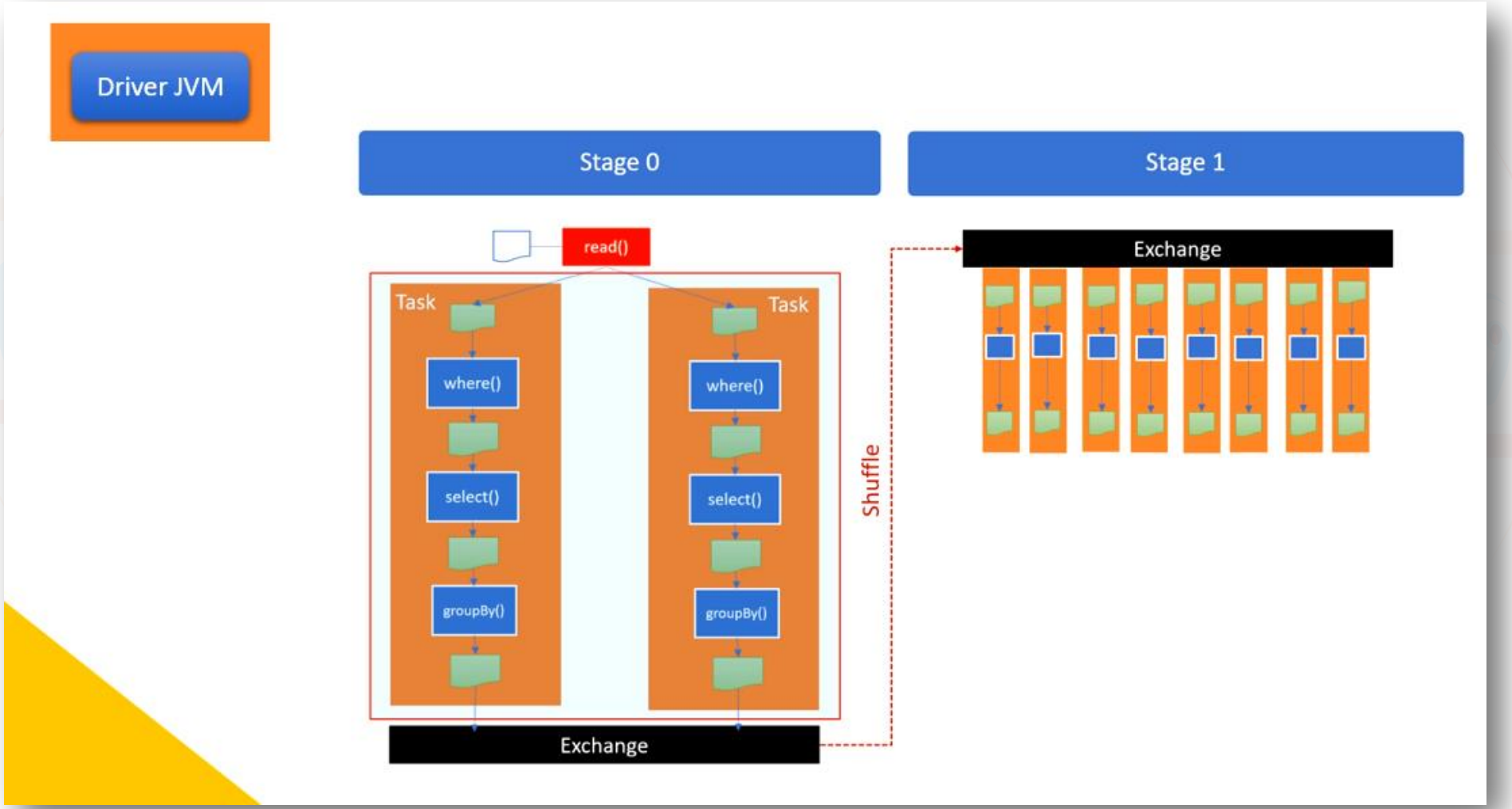
1. Spark Cluster, Master, Worker
2. Resource Container Application Master
3. Driver and Executor
4. Driver Cores and Memory Executor
5. Cores, Slots, and Memory

1. Spark Execution Plan and DAG
2. Job, Stages, and Tasks

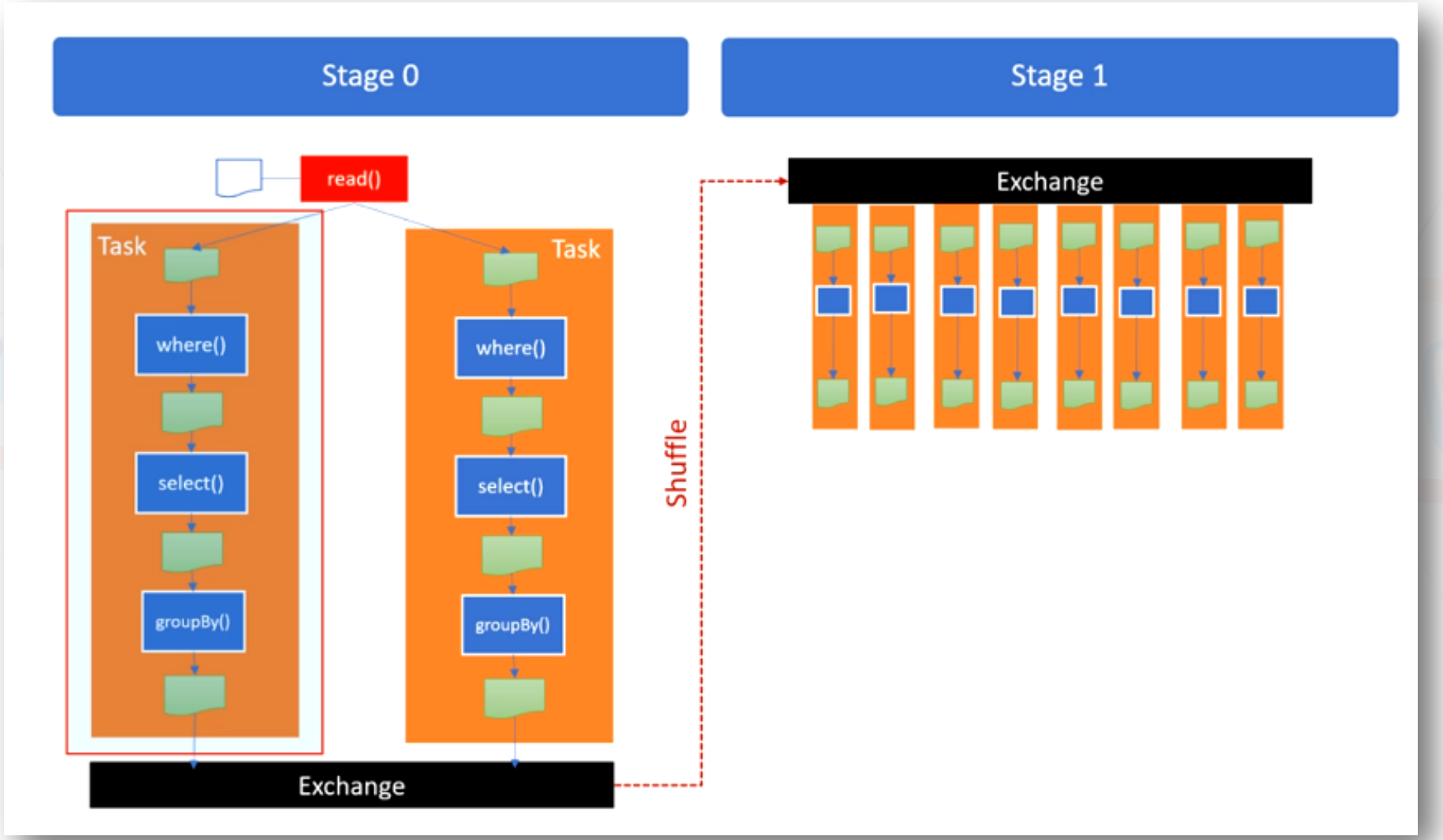
And I hope you now understand everything listed here. The main thing was to understand how the task fits into the slot. However, I am still left with few things:

1. How RDD Partitions fit into this structure.
2. How Spark actions behave
3. What happens when we write results

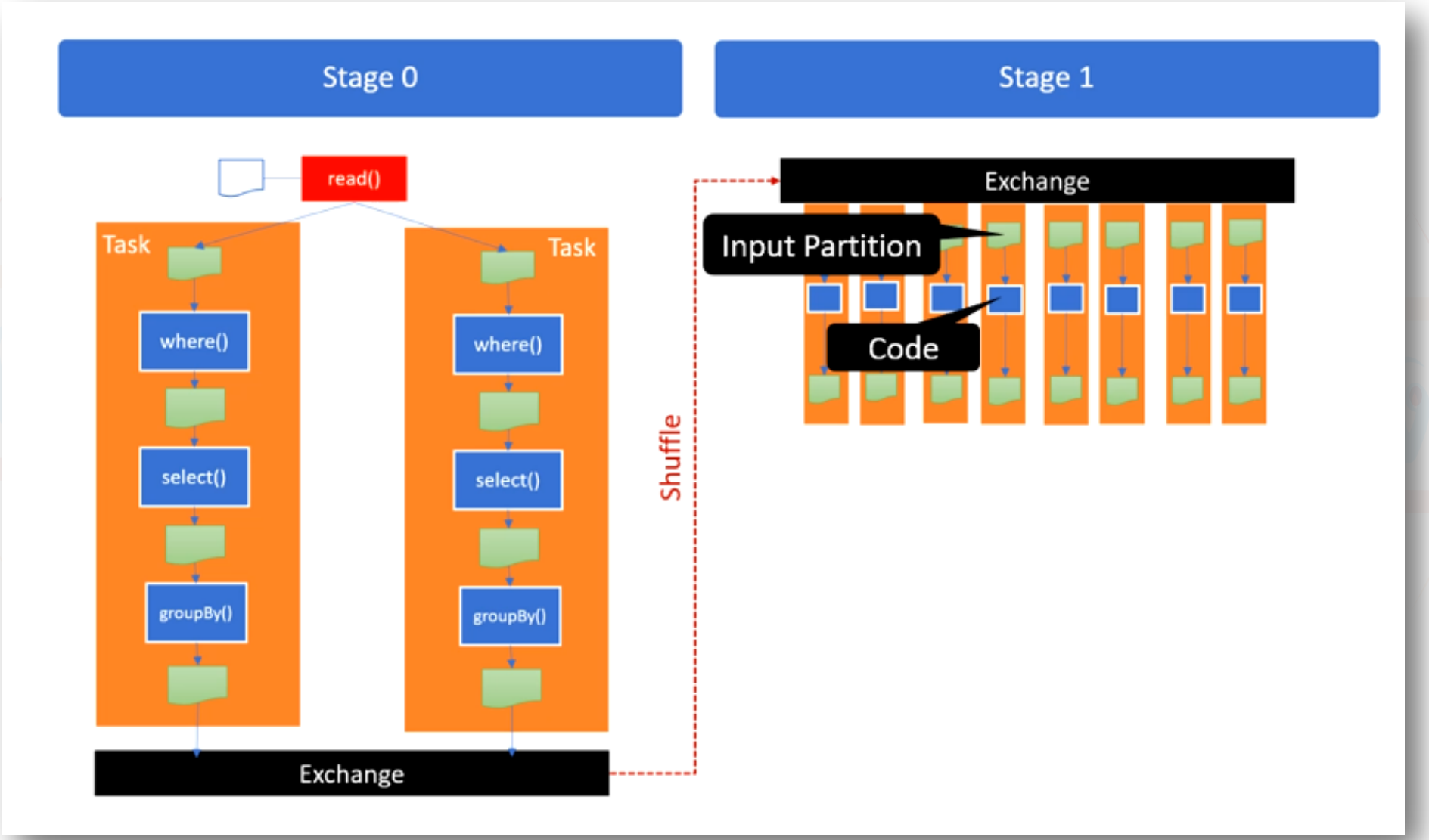
We can run a stage in parallel tasks. And the number of tasks in the stage will depend on the number of input partitions. If we have two input partitions, the stage will have two tasks. If we have eight input partitions, the stage will have eight tasks. And all these tasks can run in parallel.



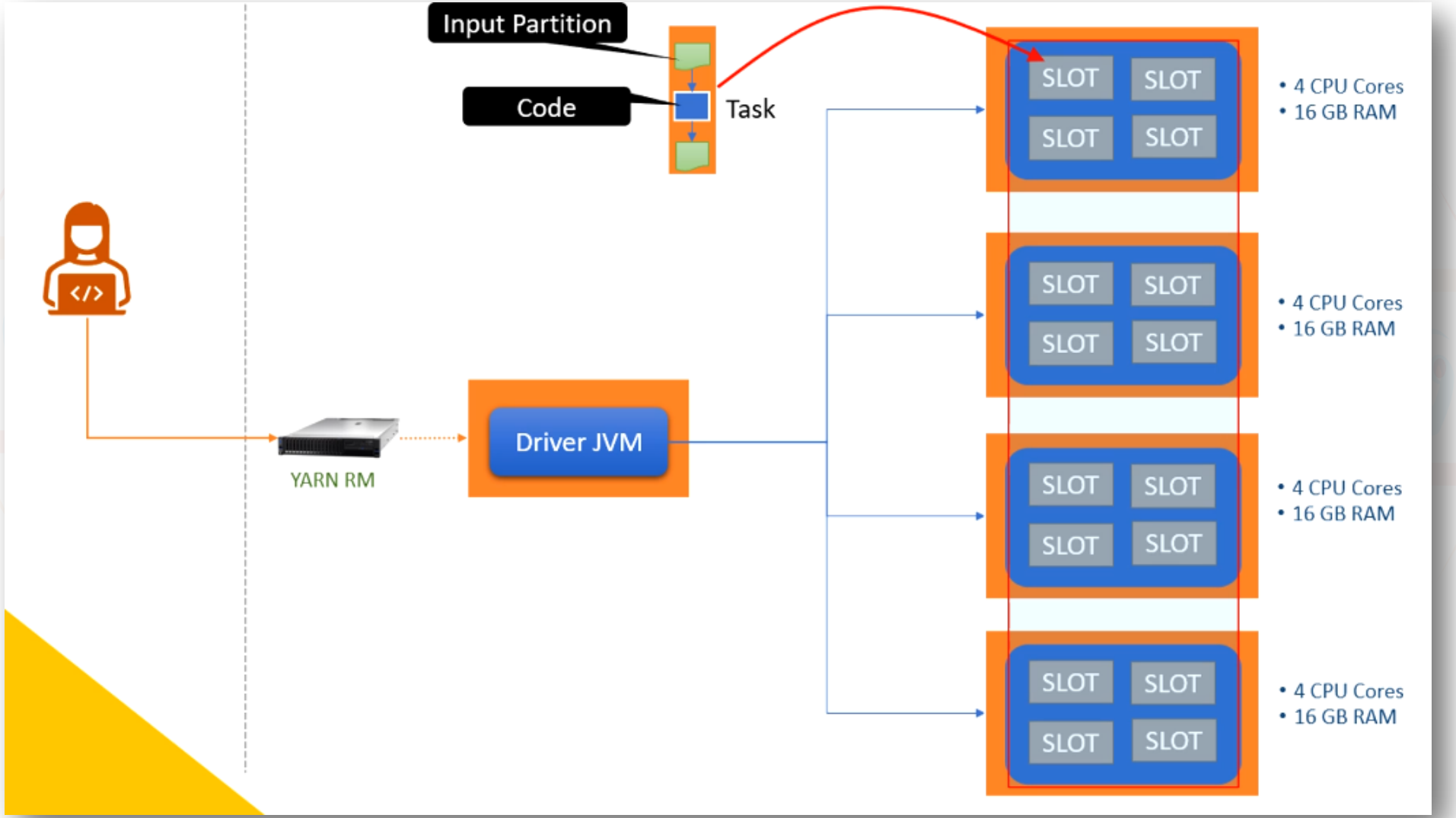
A task is a set of transformations and an input partition.
It is a data partition and some transformations that are applied to the partition.



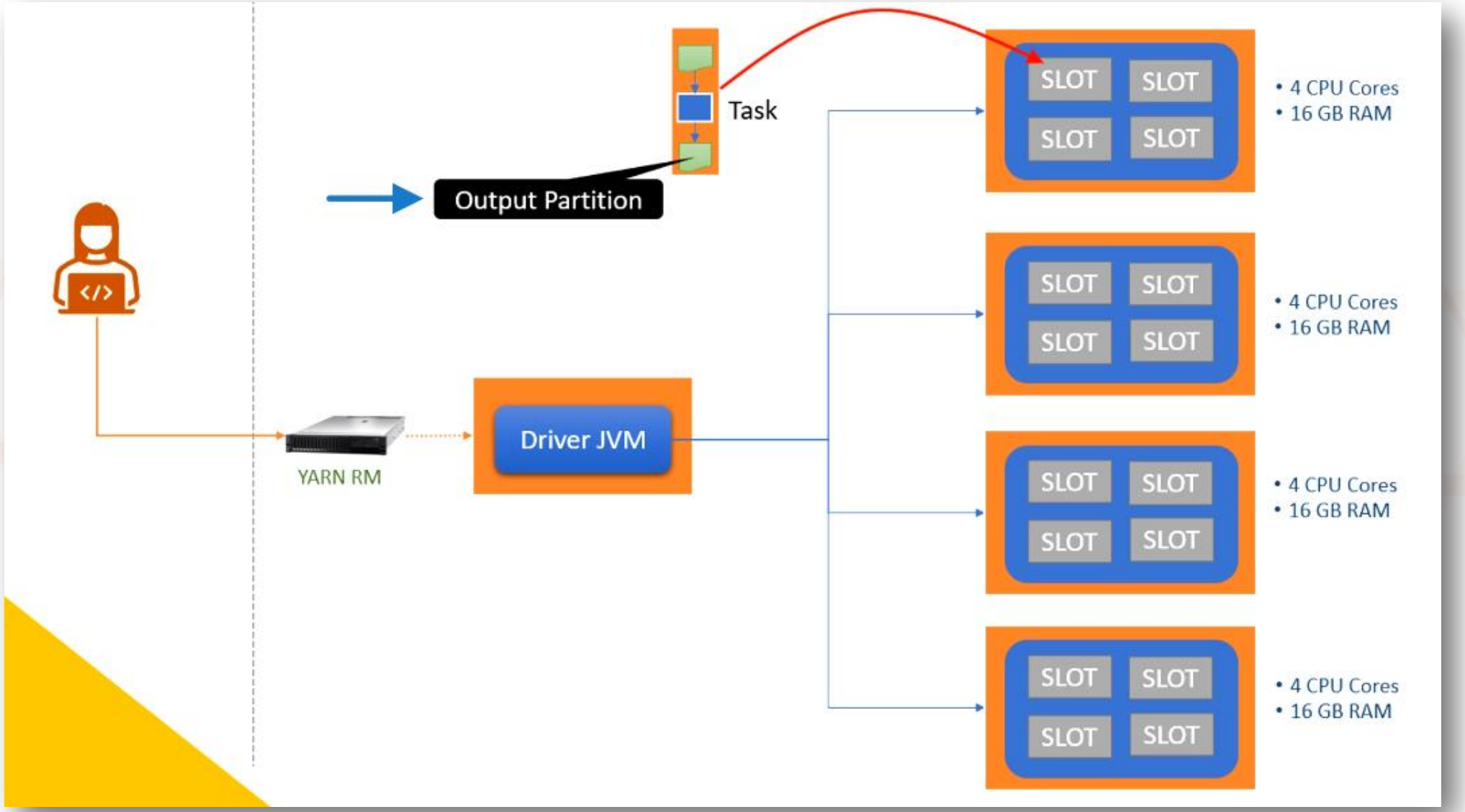
So when the driver assigns a task to a slot, it assigns both the code and the data partition.



So each slot in this diagram will get one task. That means it will get one input partition and some code.



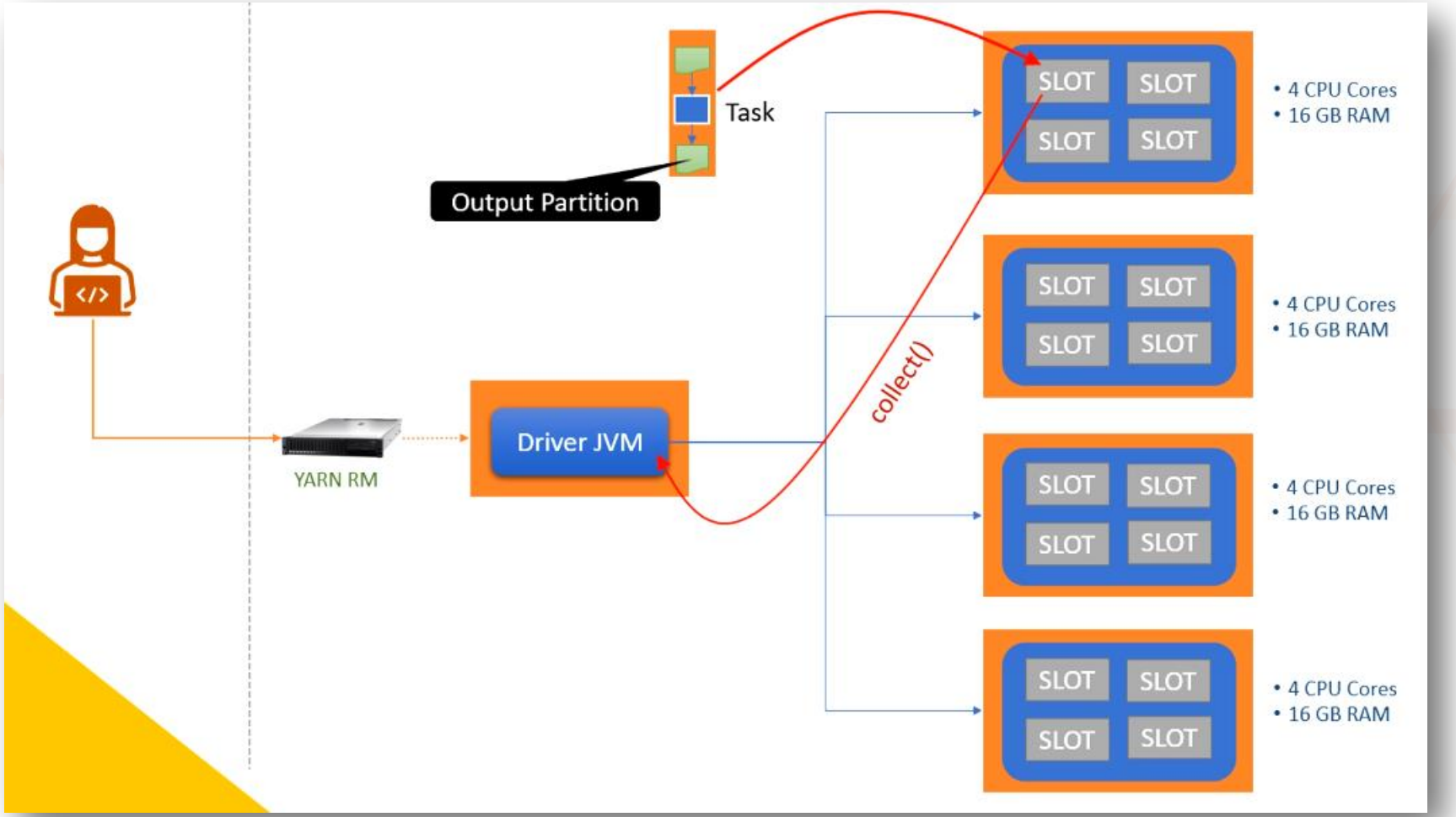
It will run the code on the given input partition to create a new transformed output partition.



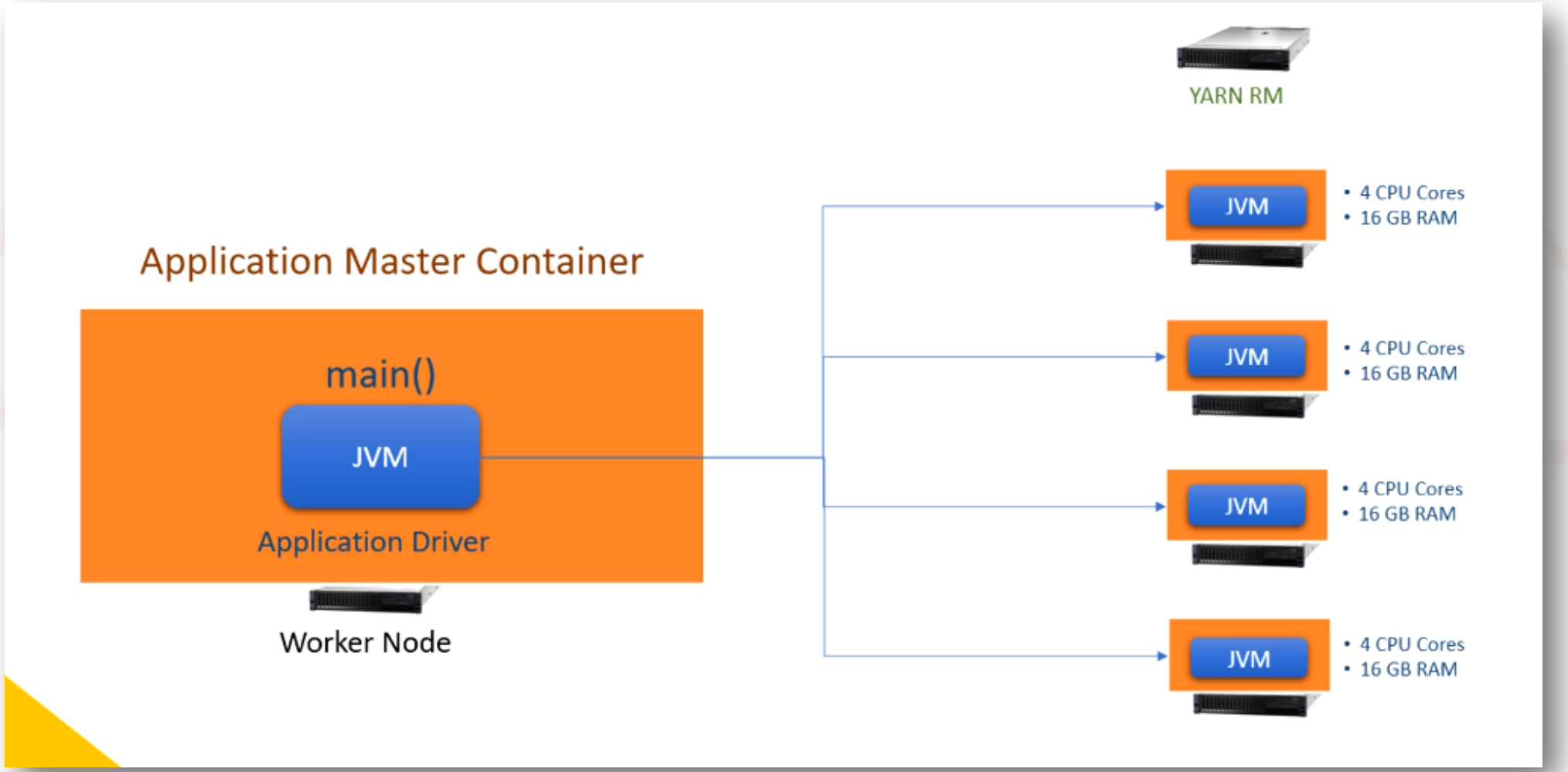
What happens to the output partition? Well, it depends on the stage, and we have three possibilities:

1. Send the output partition to the shuffle exchange - If your stage ends with a shuffle, the executor will send the output to the shuffle exchange.
2. Write the output partition to the storage directory - If the last command in the stage is to write the output in a directory or table, the executor will write the output partition as a part file.
3. Send the output partition to the driver - If your last command is an action to send data to Python, the executor will send it to the driver. For our example, if your last command is a collect() method, the executor will send the output partition to the driver.

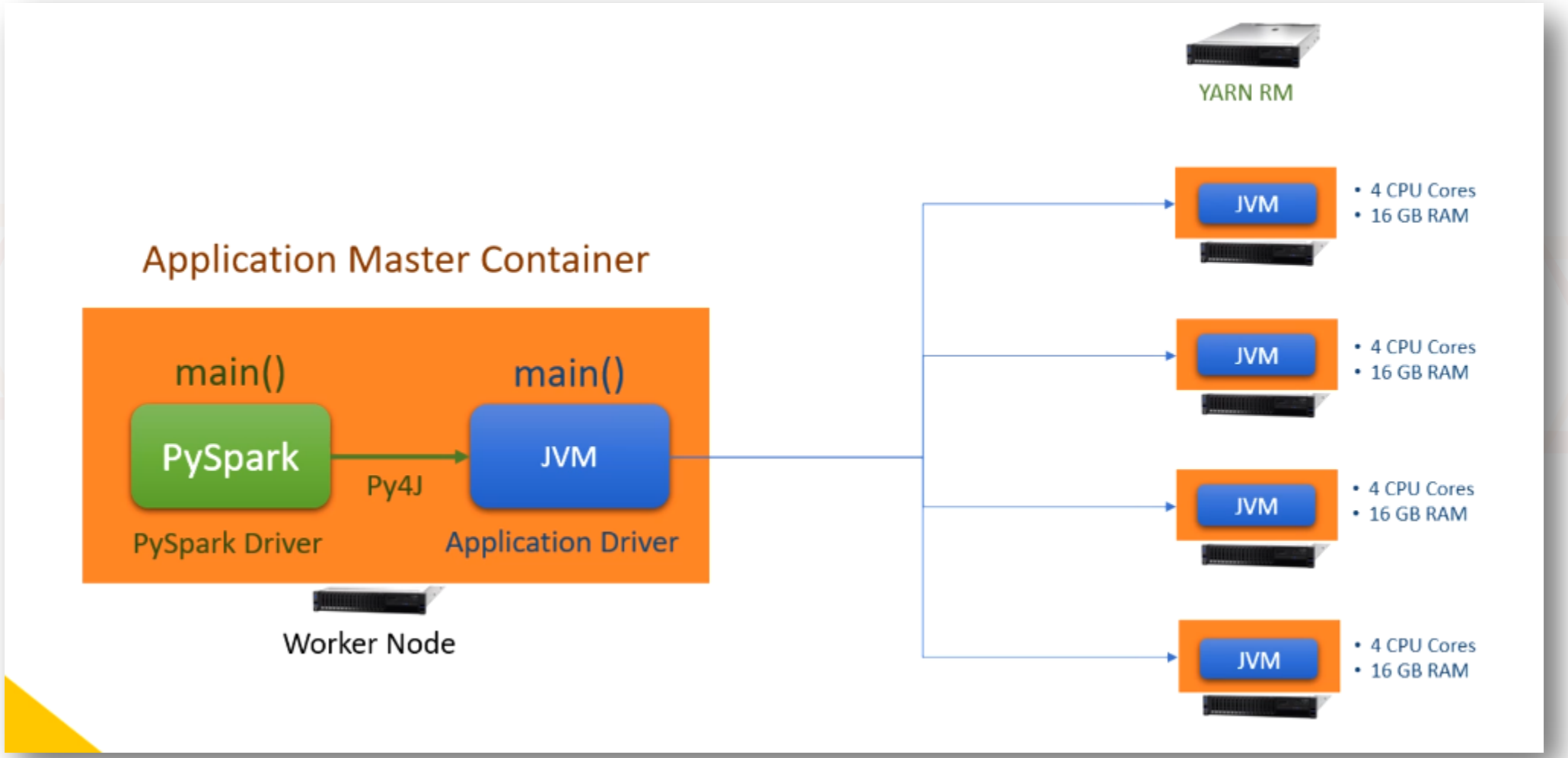
And this data travels over the network and goes to the driver's memory. In a typical case, we give a lot of memory to the executors but do not give large memory to the driver. So, you should be careful to collect() large data from the driver. Collecting large data from the driver that doesn't fit in the driver's memory will crash the driver with an OOM error.



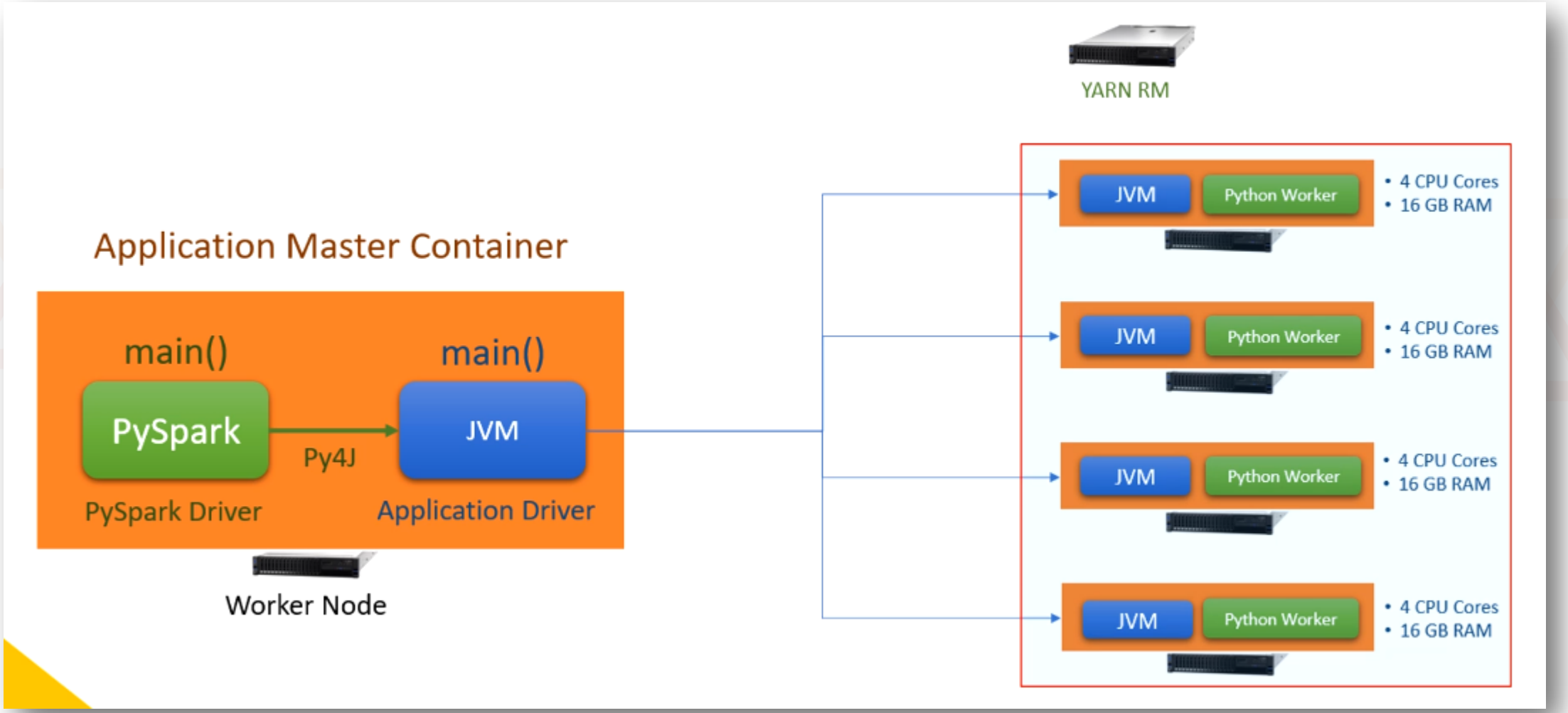
If you use Spark Dataframe API in Scala or Java, your runtime architecture looks like this. You will have one JVM driver and one or more JVM executors.



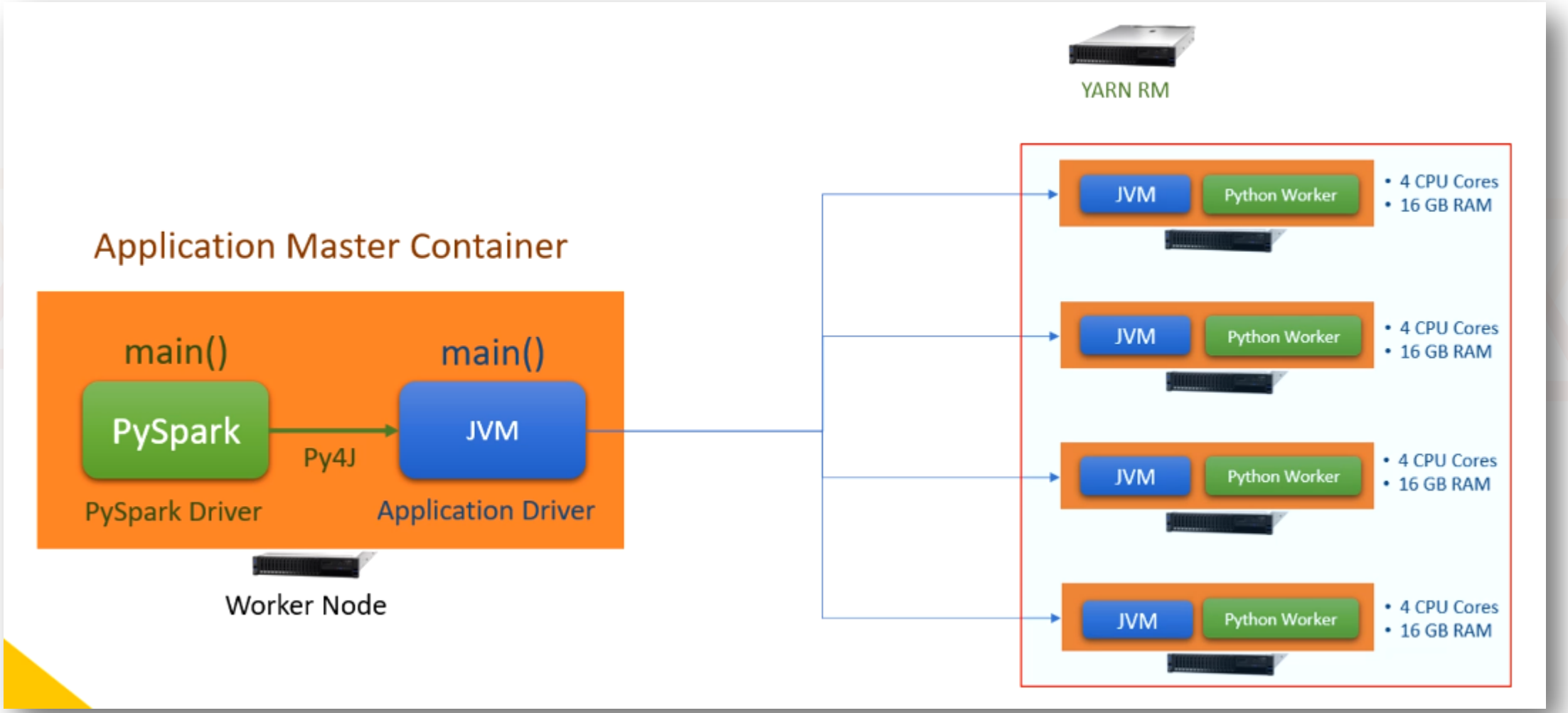
If you are using PySpark Dataframe APIs, your runtime architecture looks like this. You will have one PySpark Driver, one JVM driver, and one or more JVM executors.



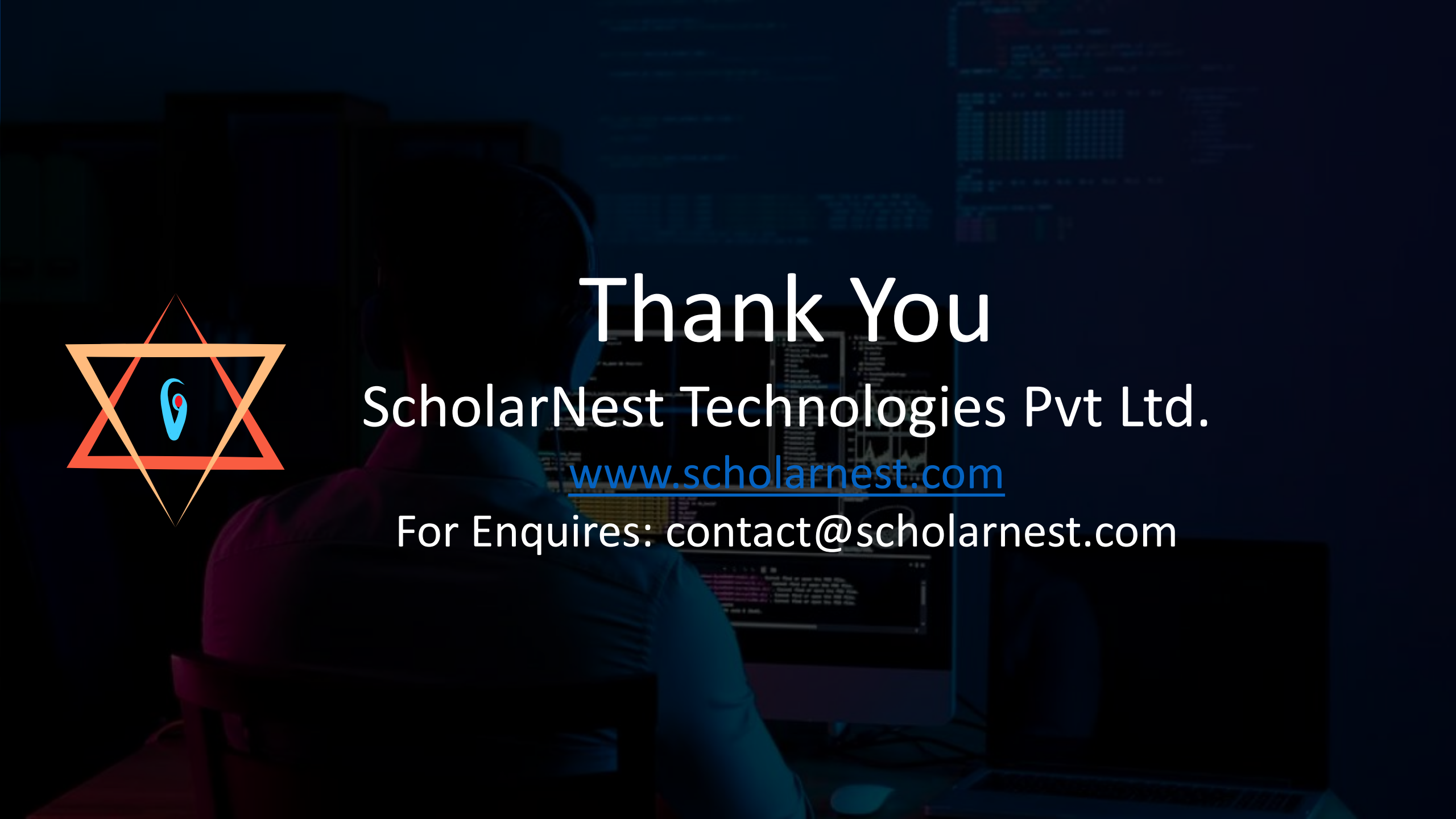
But if you are also using some additional Python libraries that are not part of the PySpark, then your runtime architecture looks like this.
Even if you create UDFs in Python, your runtime architecture will look like this.



So what is the difference? You have a Python worker at each executor. What is a Python Worker, and Why do we need them? Python worker is a Python runtime environment. And you need them only if you are using some Python-specific code or libraries.



PySpark is a wrapper on Java code.
So as long as you are using only PySpark, you do not need a Python runtime environment.
All the PySpark code is translated into Java code, and it runs in the JVM.
But if you are using some Python libraries which doesn't have a Java wrapper, you will need a Python runtime environment to run them.
So the executors will create a Python runtime environment so they can execute your Python code.
I will talk more about this in a later part of the course. But for now, let's remember that you may have Python workers inside the executor container for running custom python code outside the PySpark API.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com