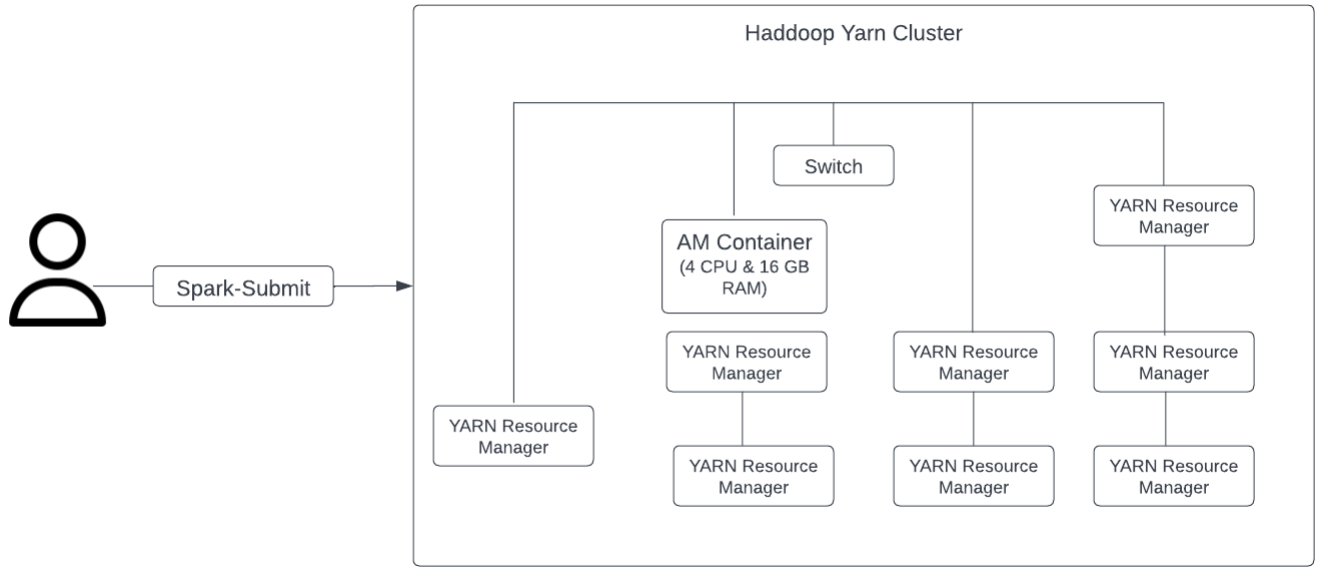
Let’s revise some concepts of Spark

1. Spark is a distributed computing platform
2. Spark Application is a distributed application
3. Spark Application needs a cluster, e.g., Hadoop YARN and Kubernetes

What is a cluster?

A pool of computers working together but viewed as a single system, e.g., I have ten worker nodes, each with 16 CPU cores and 64 GB RAM, So, my total CPU capacity is 160 CPU cores, and the RAM capacity is 640 GB.



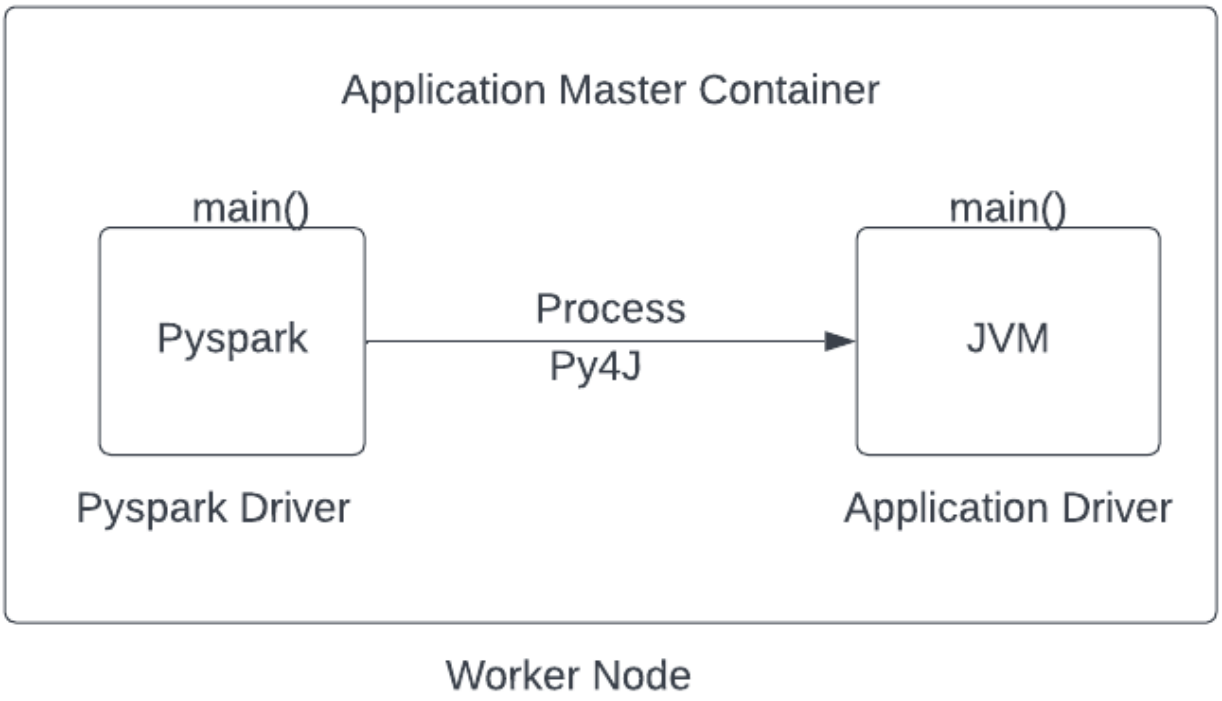
Hadoop Yarn Cluster

How is a Spark Application run on the cluster?

I will use the Spark Submit command and submit my Spark Application to the cluster.

My request will go to the YARN resource manager. The YARN RM will create one Application Master Container on a worker node and start my application's main() method in the container.

What is a container?



..
Application Master Container

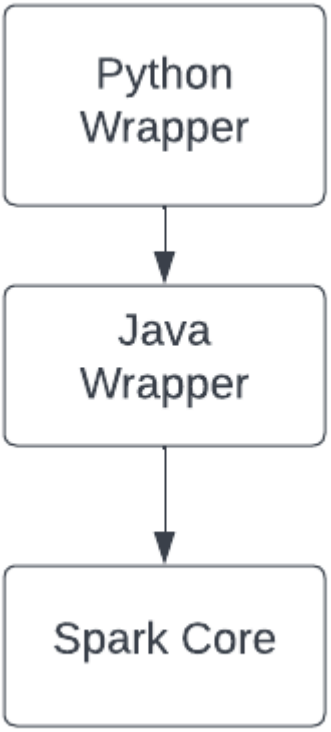
A container is an isolated virtual runtime environment. It comes with some CPU and memory allocation. For example, let's assume YARN RM gave 4 CPU Cores and 16 GB memory to this container and started it on a worker node.

Now my application's main() method will run in the container, and it can use 4 CPU cores and 16 GB of memory.

The container is running main() method on my application, and we have two possibilities here.

1. PySpark Application
2. Scala Application

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



PySpark Application

Spark developers wanted to bring this to Python developers, so they created a Java wrapper on top of Scala code, then created Python wrapper on top of the Java wrappers, and the Python wrapper is known as PySpark.

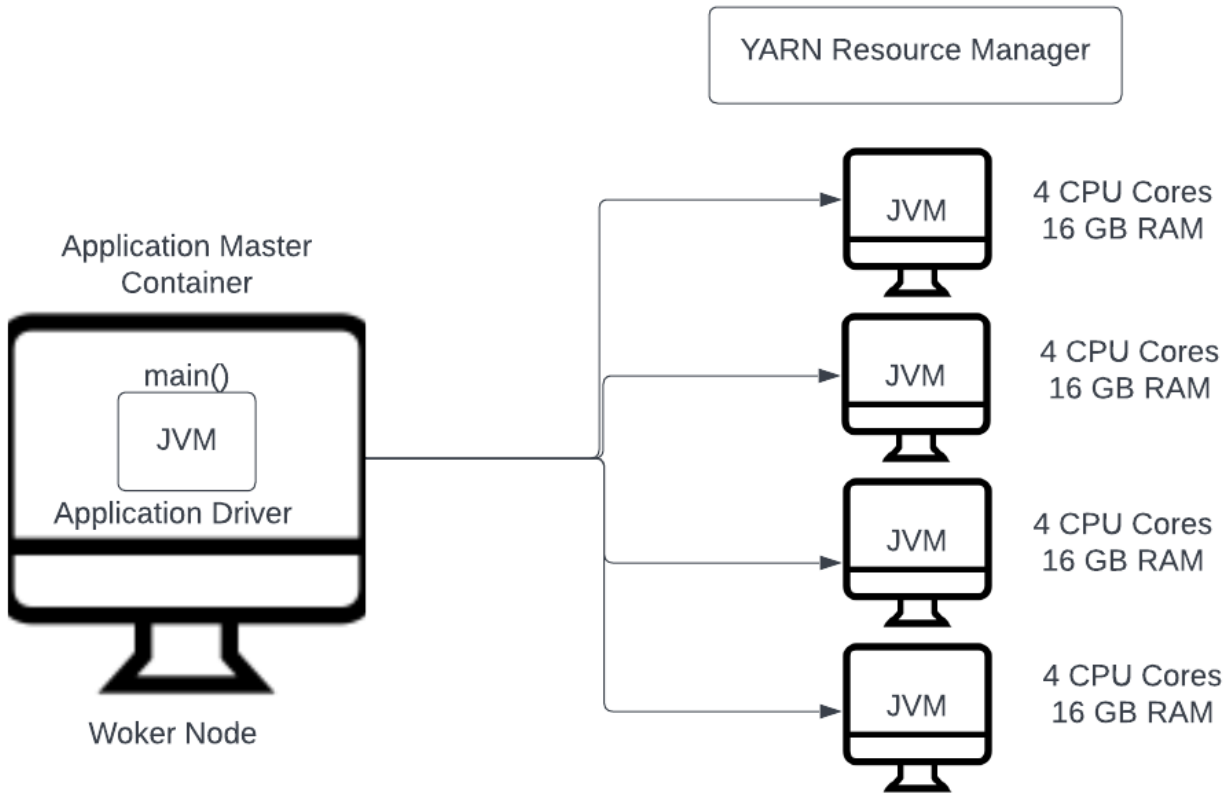
If you submitted the PySpark code, you would have a PySpark driver, and you will also have a JVM driver. These two will communicate using Py4J.

The PySpark main method is the PySpark Driver, and the JVM application is Application Driver.

The Application driver distributes the work to others. So, the driver does not perform any data processing work. Instead, it will create some executors and get the work-done from them.

But how does Application Driver work?

After starting, the driver will go-back to the YARN RM and ask for some more containers. The RM will create some more containers on worker nodes and give them to the driver.



Application Driver Execution

Now the driver will start spark executor 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 also a JVM application. These executors are responsible for doing all the data processing work. The driver will assign work to the executors, monitor them, and manage the overall application, but the executors do all the data processing.

PySpark code translated into Java code, and 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.

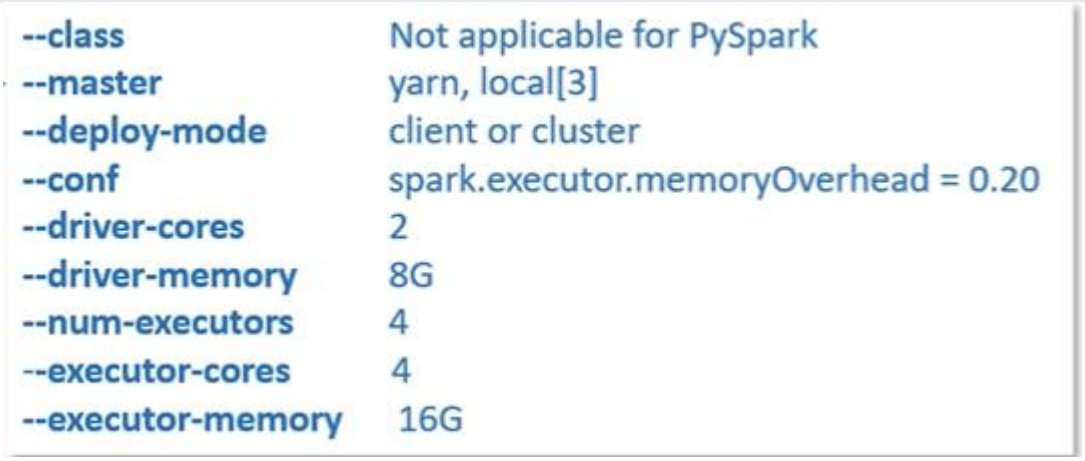
So, the executors will create a Python runtime environment so they can execute your Python code.

Spark Submit and its Options

What is Spark Submit?

The Spark Submit is a command-line tool that allows you to submit the Spark application to the cluster.

```
spark-submit --class<main-class> --master<master-url> --deploy-mode<deploy-mode> <application-jar>[application-args]
```



Arguments used in a Spark Submit Command

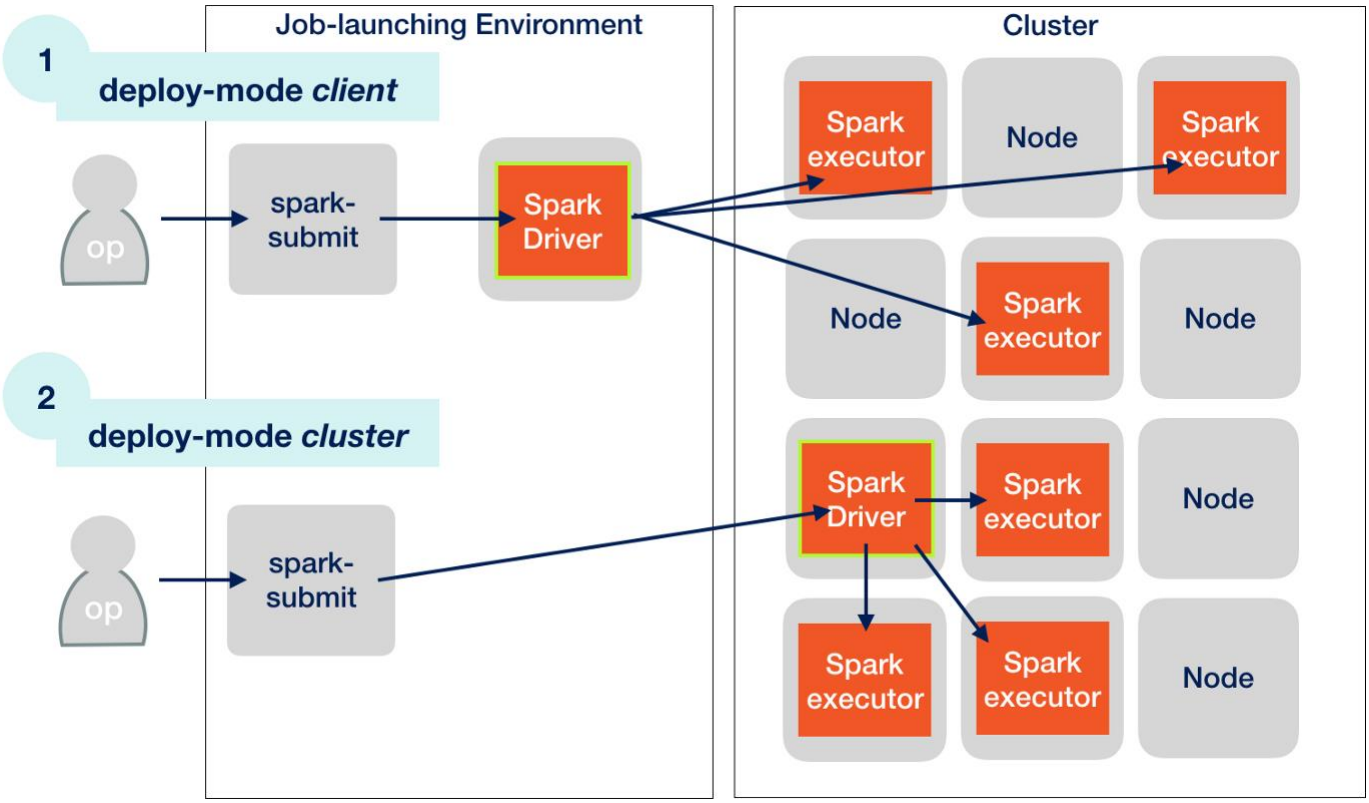
What are the types of Deploy Modes?

The Spark Submit allows you to submit the spark application to the cluster, and you can apply to run in one of the two modes.

1. Cluster Mode
2. Client Mode

In the cluster mode, Spark Submit will reach the YARN RM, requesting it to start the driver in an AM container. YARN will start your driver in the AM container on a worker node in the cluster.

Then the driver will again request YARN to begin executor containers. So, the YARN will start executor containers and hand them over to the driver. So, in cluster mode, your driver is running in the AM container on a worker node in cluster. Your executors are also running in the executor containers on some worker nodes in cluster.



Execution of Client Deploy Mode and Cluster Deploy Mode

In the Client Mode, Spark Submit doesn't go to the YARN resource manager for starting an AM container. Instead, the spark-submit command will start the driver JVM directly on the client machine. So, in this case, the spark driver is a JVM application running on

your client machine. Now the driver will reach out to the YARN resource manager requesting executor containers. The YARN RM will start executor containers and hand them over to the driver. The driver will start executors in those containers to do the job.

Client Machines in the Spark Cluster are also known as Gateway Nodes.

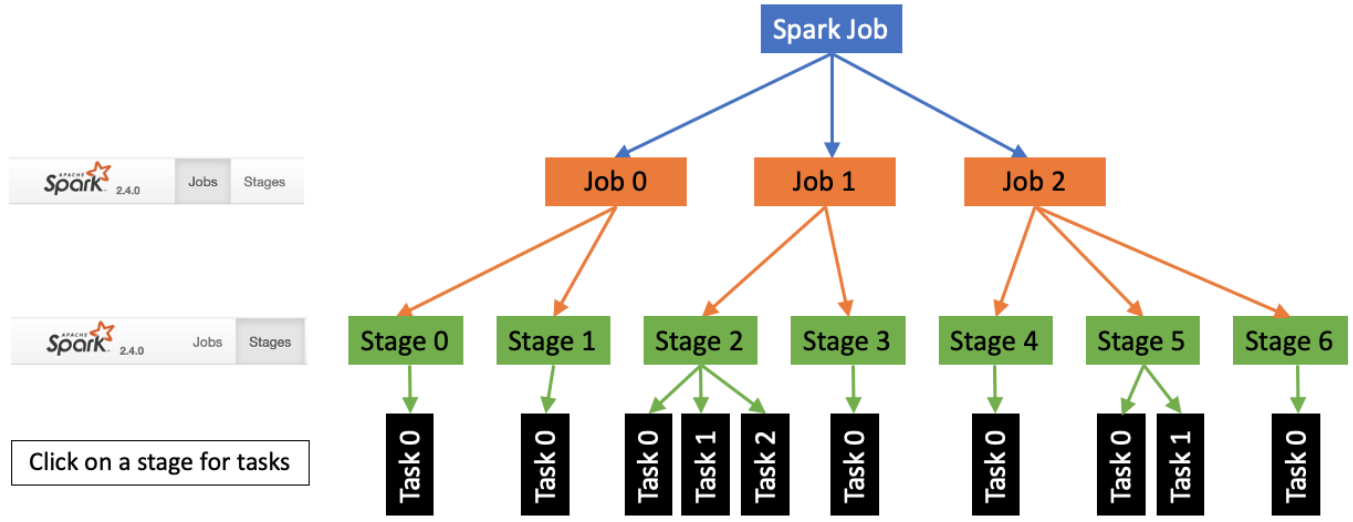
How do we choose the deploy mode?

You will almost always submit your application in cluster mode. It is unlikely that you submit your spark application in client mode. We have two clear advantages of running your application in cluster mode.

1. The cluster mode allows you to submit the application and log off from the client machine, as the driver and executors run on the cluster. They have nothing active on your client's machine. So, even if you log off from your client machine, the driver and executor will continue to run in the cluster.
2. Your application runs faster in cluster mode because your driver is closer to the executors. The driver and executor communicate heavily, and you don't get impacted by network latency, if they are close.

The designed client mode is for interactive workloads. For example, Spark Shell runs your code in client mode. Similarly, Spark notebooks also use the client mode.

Spark Jobs: Stage, Shuffle, Tasks, Slots



Spark Jobs Hierarchy

1. Spark creates one job for each action.
2. This job may contain a series of multiple transformations.
3. The Spark engine will optimize those transformations and creates a logical plan for the job.
4. Then spark will break the logical plan at the end of every wide dependency and create two or more stages.
5. If you do not have a wide dependency, your plan will be a single-stage plan.
6. But if you have N wide-dependencies, your plan should have N+1 stages.
7. Data from one stage to another stage is shared using the shuffle/sort operation.
8. Now each stage may be executed as one or more parallel tasks.

9. The number of tasks in the stage is equal to the number of input partitions.

The task is the most critical concept for a Spark job and is the smallest unit of work in a Spark job. The Spark driver assigns these tasks to the executors and asks them to do the work.

The executor needs the following things to perform the task.

1. The task Code
2. Data Partition

So, the driver is responsible for assigning a task to the executor. The executor will ask for the code or API to be executed for the task. It will also ask for the data frame partition on which to execute the given code. The application driver facilitates both these things for the executor, and the executor performs the task.

Now, let's assume I have a driver and four executors. Each executor will have one JVM process. But I assigned 4 CPU cores to each executor. So, my Executor JVM can create four parallel threads and that's the slot capacity of my executor.

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 and it is going to assign tasks to fit in the executor slots.

The last stage will send the result back to the driver over the network. The driver will collect data from all the tasks and present it to you.

Spark SQL Engine and Query Planning

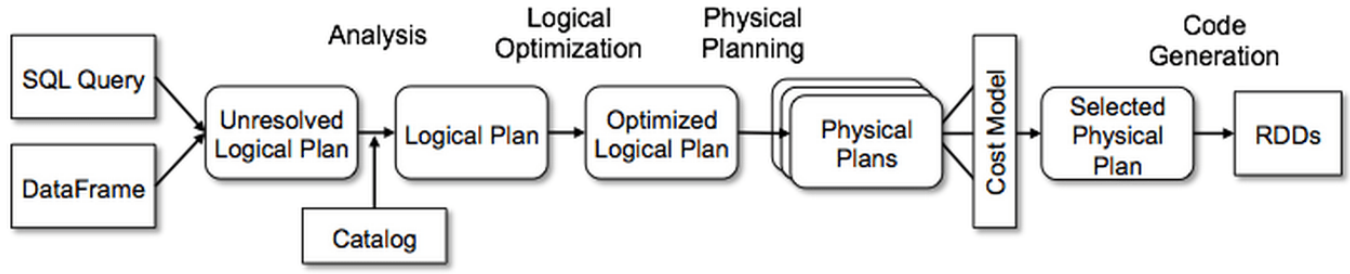
Apache Spark gives you two prominent interfaces to work with data.

1. Spark SQL
2. Dataframe API

You may have Dataframe APIs, or you may have SQL both will go to the Spark SQL engine.

For Spark, they are nothing but a Spark Job represented as a logical plan.

The Spark SQL Engine will process your logical plan in four stages.



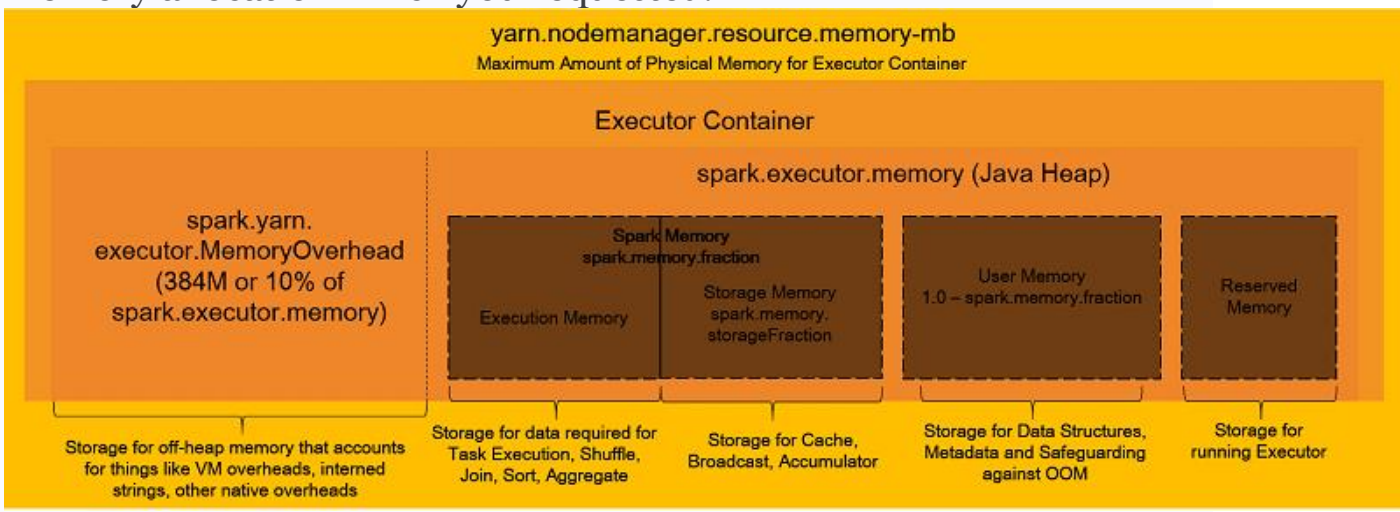
Catalyst Optimization

1. The Analysis stage will parse your code for errors and incorrect names. The Analysis phase will parse your code and create a fully resolved logical plan. Your code is valid if it passes the Analysis phase.

2. The Logical Optimization phase applies standard rule-based optimizations to the logic plan.
3. Spark SQL takes a logical plan and generates one or more in the Physical Planning phase. Physical planning phase considers cost based optimization. So the engine will create multiple plans to calculate each plan's cost and select the one with the low cost. At this stage the engine use different join algorithms to generate more than one physical plan.
4. The last stage is Code Generation. So, your best physical plan goes into code generation, the engine will generate Java byte code for the RDD operations, and that's why Spark is also said to act as a compiler as it uses state of the art compiler technology for code generation to accelerate execution.

SPARK Memory Allocation

Assume you submitted a spark application in a YARN cluster. The YARN RM will allocate an application master (AM) container and start the driver JVM in the container. The driver will start with some memory allocation which you requested.



Spark Memory Architecture

You can ask for the driver's memory using two configurations.

1. spark.driver.memory
2. spark.driver.memoryOverhead

So, let's assume you asked for the spark.driver.memory as 1GB and the default value of spark.driver.memoryOverhead as 0.10

The YARN RM will allocate 1 GB memory for the driver JVM, and 10% of requested memory or 384 MB, whatever is higher for container overhead.

The overhead memory is used by the container process or any other non JVM process within the container. Your Spark driver uses all the JVM heap but nothing from the overhead.

So, the driver will again request the executor containers from the YARN. The YARN RM will allocate a bunch of executor containers.

The total memory allocated to the executor container is the sum of the following:

1. Overhead Memory
2. Heap Memory
3. Off Heap Memory
4. PySpark Memory

So, a Spark driver will ask for executor container memory using four configurations.

What are the configurations used for executor container memory?

1. Overhead memory is the spark.executor.memoryOverhead
2. JVM Heap is the spark.executor.memory.
3. Off Heap memory comes from spark.memory.offHeap.size.
4. The PySpark memory comes from the spark.executor.pyspark.memory.

So, the driver will look at all these configurations to calculate your memory requirement and sum it up.

The container should run on a worker node in the YARN cluster. What if the worker node is a 6 GB machine? YARN cannot allocate an 8 GB container on a 6 GB machine due to lack of physical memory. Before you ask for the driver or executor memory, check with your cluster admin for the maximum allowed value.

While using YARN RM, you should look for the following configurations.

1. yarn.scheduler.maximum-allocation-mb
2. yarn.nodemanager.resource.memory-mb

You do not need to worry about PySpark memory if you write your Spark application in Java or Scala. But if you are using PySpark, this question becomes critical.

PySpark is not a JVM process but overhead memory. Some of which is constantly consumed by the container and other internal processes. If your PySpark occupies more than accommodated in the overhead, you will see an OOM error.

Let's have a quick recap

1. You have a container, and the container has got some memory.

2. This total memory is broken into two parts: Heap memory(driver/executor memory) and Overhead memory (OS Memory)

3. The heap memory goes to your JVM.

4. We call it driver memory when running a driver in this container. Similarly, we call it executor memory when the container runs an executor.

The overhead memory uses for a bunch of things. The overhead uses for network buffers. So, you will be using overhead memory as your shuffle exchange or reading partition data from remote storage, etc.

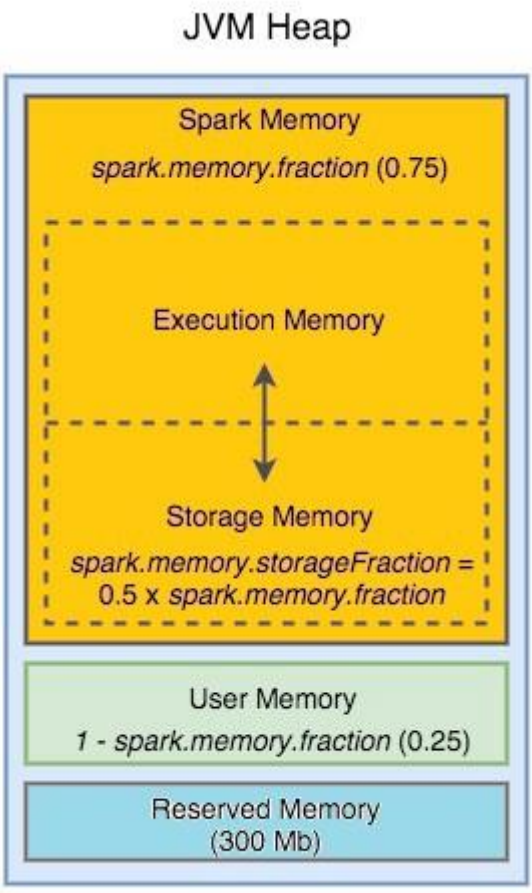
Both the memory portions are critical for your Spark application. And more often, lack of enough overhead memory

will cost you an OOM exception. As the overhead memory is overlooked, it is used as shuffle exchange or network read buffer.

Spark Memory Management

Let's focus on the JVM memory in this part. The heap memory is further broken down into three parts.

1. Reserved Memory
2. Spark Memory Pool
3. User Memory



So, let's assume I got 8 GB for the JVM heap. This 8 GB is divided into three parts. Spark will reserve 300 MB for itself. That's fixed, and the Spark engine itself uses it.

The next part is the Spark executor memory pool, controlled by the spark.memory.fraction configuration, and the default value is 60%. So, for example, the spark memory pool translates to 4620 MB.

How do Spark Memory Pools work?

We have got 8 GB or 8000 MB. Three hundred is gone for Reserved memory. Right? We are remained with 7700 MB. Now take 60% of this, and you will get 4620 MB for the Spark memory pool. What is left? We are left with 3080 MB, and gone for user memory.

Now let's try to understand the three memory pools.

1. The Reserved Pool is gone for the Spark engine itself. You cannot use it.

2. The Spark Memory Pool is your main executor memory pool which you will use for data frame operations and caching.

The Spark memory pool is where all your data frames and data frame operations live. You can increase it from 60% to 70% or even more if you are not using UDFs, custom data structures, and RDD operations. But you cannot make it zero or reduce it too much because you will need it for metadata and other internal things.

Spark Executor Memory Pool is further broken down into two sub pools.

- Storage Memory
- Executor Memory

The default break up for each sub pool is 50% each, but you can change it using the spark.memory.storageFraction configuration.

We use the Storage Pool for caching data frames and the Executor Pool is to perform data frame computations.

3. The User Memory Pool is used for non data frame operations.

Here are some examples for User Memory Pool:

- If you created user defined data structures such as hash maps, Spark would use the User Memory Pool.
- Similarly, Spark internal metadata and user-defined functions are stored in the user memory.
- All the RDD information and the RDD operations are performed in user memory.

But if you are using Data Frame operations, they do not use the user memory even if the data frame is internally translated and compiled into RDD. You will be using user memory only if you apply RDD operations directly in your code.