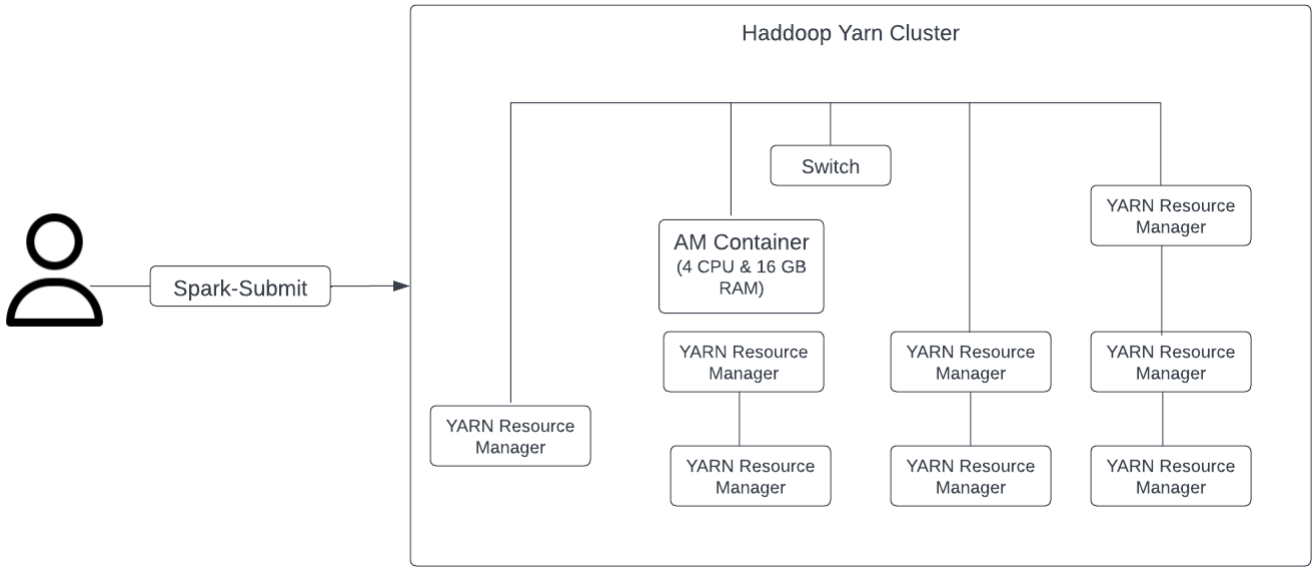
Let’s revise some concepts of Spark

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.



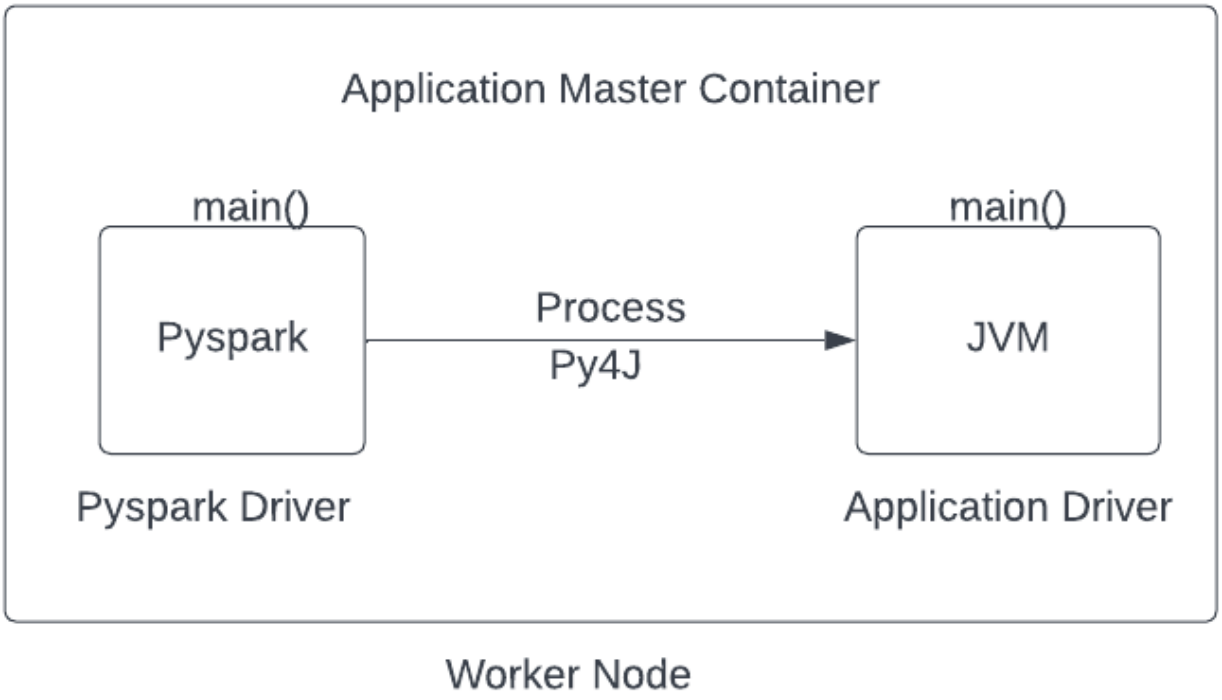
Hadoop Yarn Cluster

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

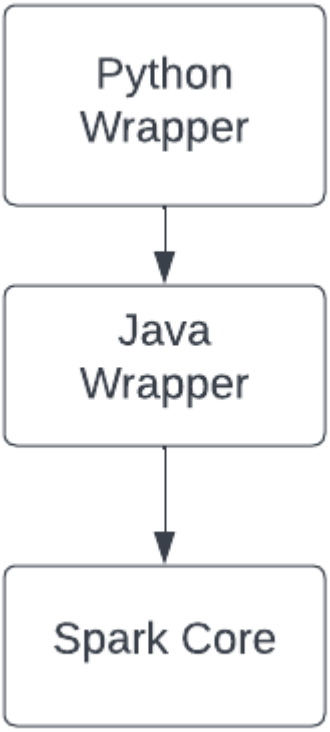
A container is an isolated virtual runtime environment. It comes with some CPU and memory allocation. For example, let's assume YARN RM gave 4 CPU Cores and 16 GB memory to this container and started it on a worker node.

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.

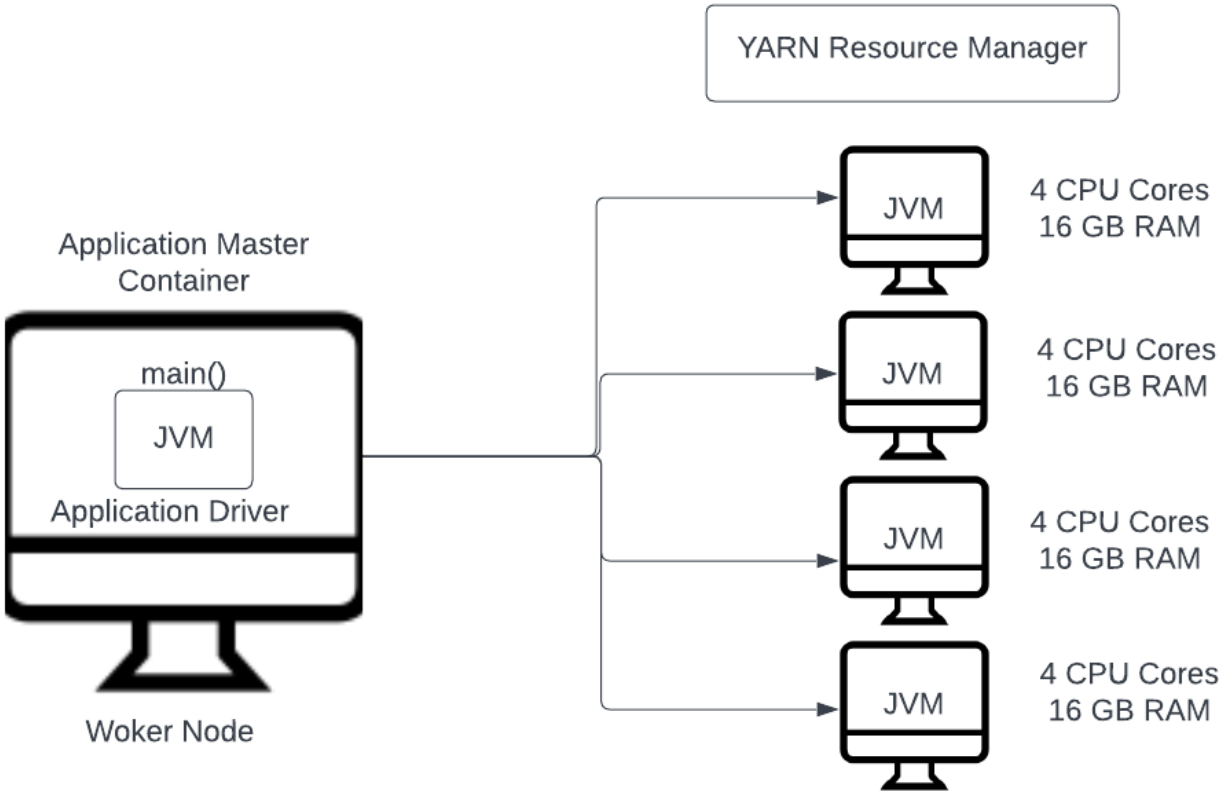
If you submitted the PySpark code, you would have a PySpark driver, and you will also have a JVM driver. These two will communicate using Py4J.

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]
```

--class	Not applicable for PySpark
--master	yarn, local[3]
--deploy-mode	client or cluster
--conf	spark.executor.memoryOverhead = 0.20
--driver-cores	2
--driver-memory	8G
--num-executors	4
--executor-cores	4
--executor-memory	16G

Arguments used in a Spark Submit Command

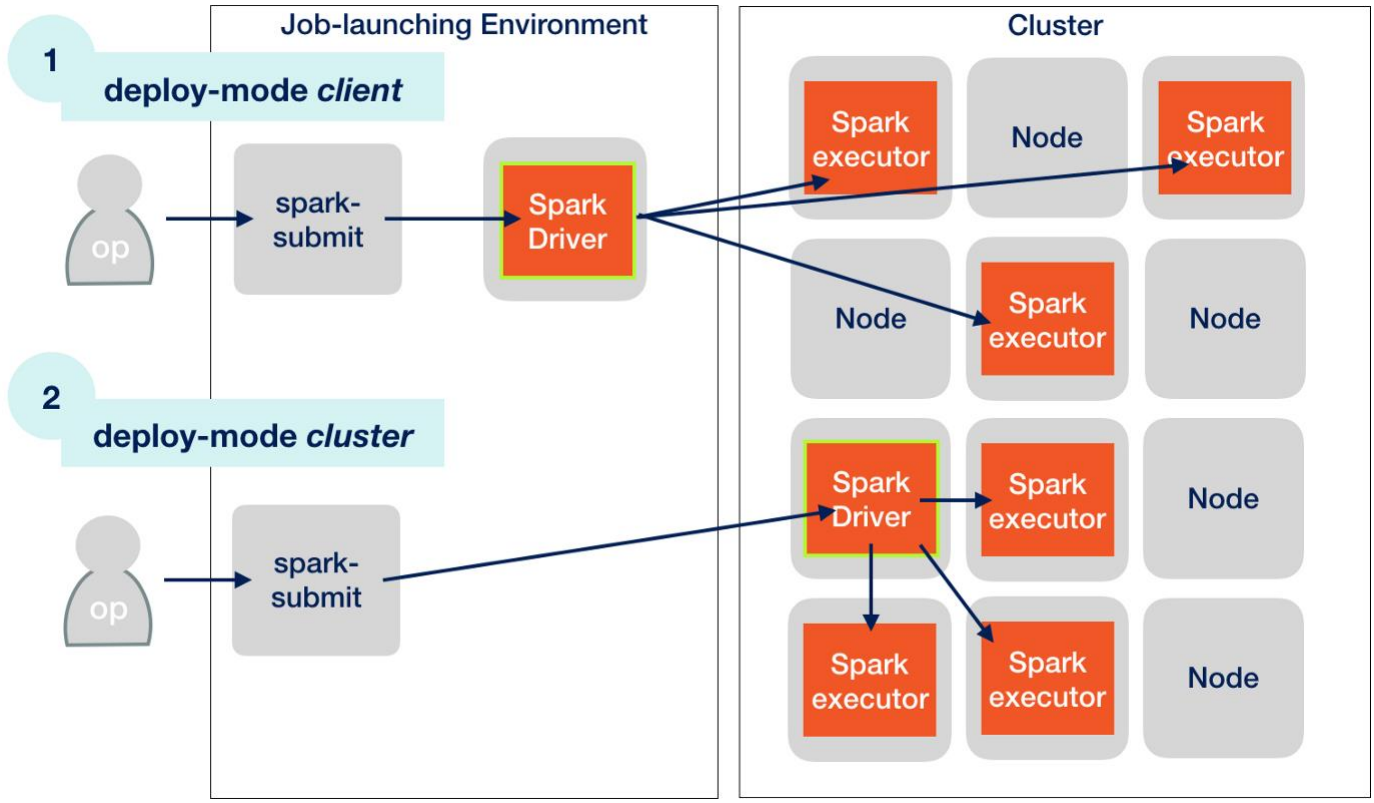
What are the types of Deploy Modes?

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.

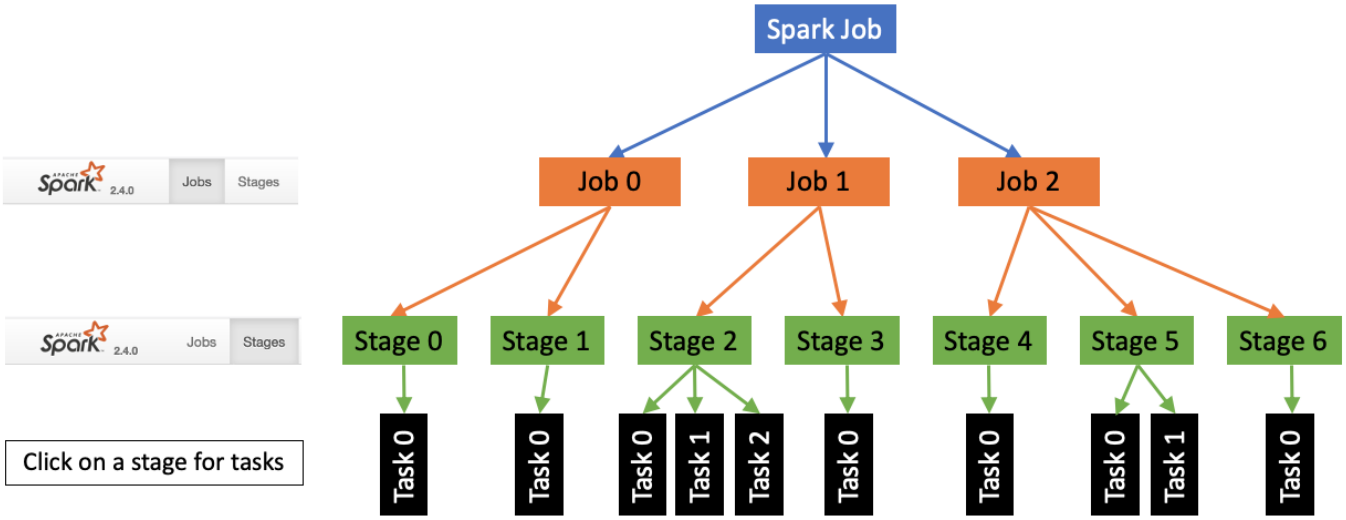
How do we choose the deploy mode?

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

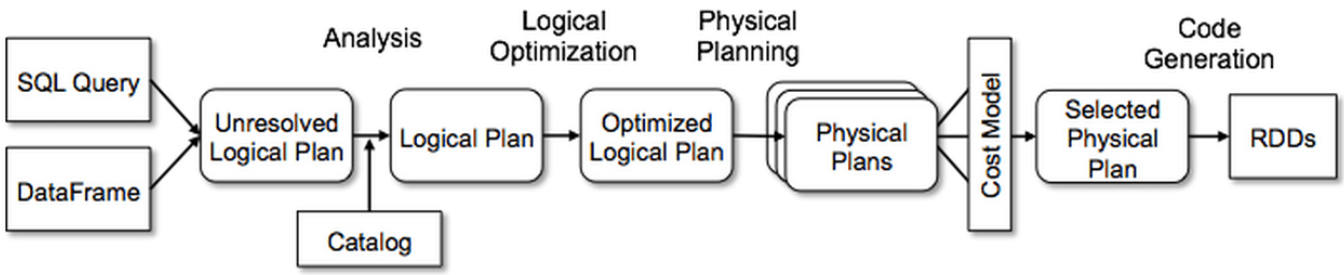
Apache Spark gives you two prominent interfaces to work with data.

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.



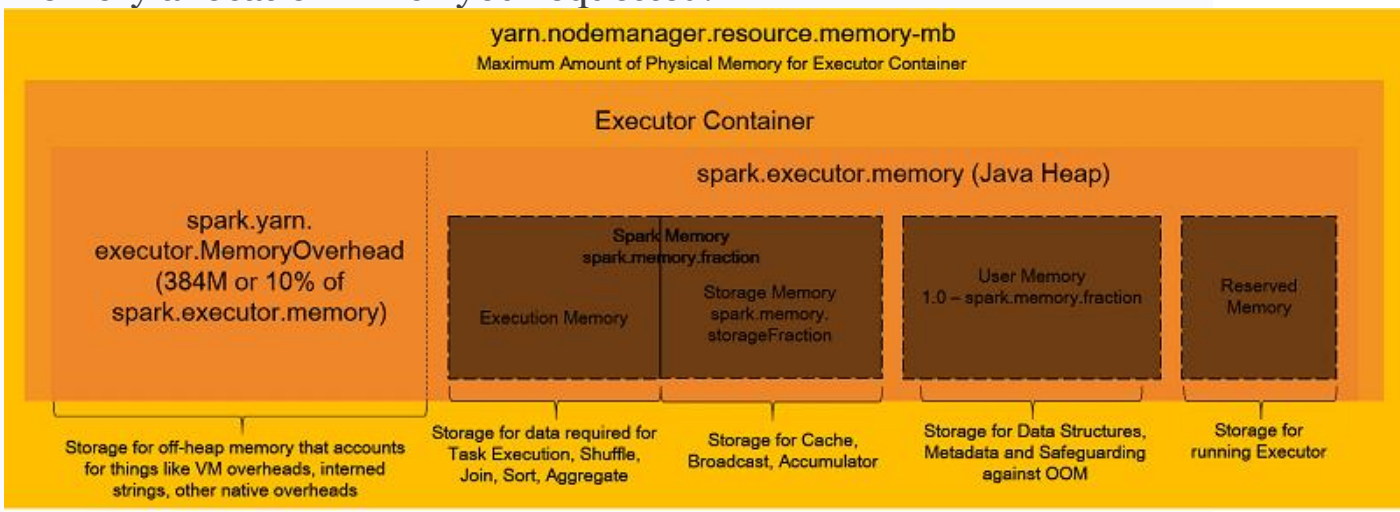
Catalyst Optimization

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.

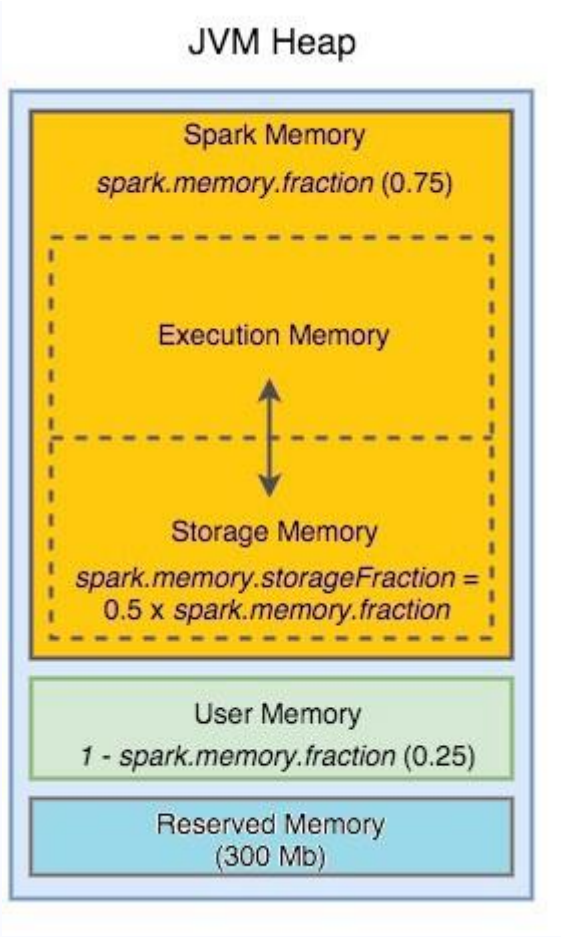
Both the memory portions are critical for your Spark application. And more often, lack of enough overhead memory

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.

Common Performance Problems

Performance issues in Spark are mostly related to the following topics:

- **Skew**: what occurs in case of imbalance in the size of data partitions.
- **Spill**: the writing of temp files to disk due to lack of memory.
- **Shuffle**: moving data between executors due to a wide transformation.
- **Storage**: the way data is stored on disk actually matters.
- **Serialization**: the distribution of code across the cluster (UDFs are evil).

Although finding the root of a problem can be quite hard since one problem can actually cause another one: Skew can induce spill, storage issues can induce excess of shuffle, a wrong way of addressing shuffle can increase skew… And sometimes, many of this causes can be present at the same time!

Skew

In Spark, data is typically read in partitions that are evenly distributed across the cluster executors. As long as we apply transformations to the data, it’s possible that some partitions will end up with much more records than others. This imbalance in the size of data partitions is what we call Skew.

Small amount of skew is not a problem and can be ignored. Although, large skews in the data can result in partitions that big that can't fit in the RAM of the workers. This would result in spill, and sometimes, even OOM errors hard to diagnose.

Detecting skew

We have to deep dive into the Spark UI and look at the join job and pay attention to the following:

- **Check Event Timeline:** we consider it unhealthy when we see very unbalance tasks, what means that some tasks are computing way more or less data than others, in other words, partitions are unbalaced. They should all have more or less the same duration.
- **Check Summary Metrics:** pay attention to the Shuffle Read values (min/25th p./median/75th p./max), if the 75th p (meaning most of the tasks) have a high number and the others very low means that most of the tasks are shuffling a lot of data, because the data required is in other partitions. If these Shuffle Read values are are similar to each other, it would indicate that all the tasks are shuffling about the same amount of data, and therefor the partitions are decently balanced.
- **Check Aggregated Metrics by Executor:** If Spill value (memory/disk) is very high, this is caused because of shuffle caused by the skew. Too big partitions can require to store data in temp files in disk because of lack of memory.
- **Inspecting the data:** Once you know you might be facing skew issues, you can get out of doubts by inspecting the data. By performing a count of records per category of "group by" or "join key" and checking if there are way many more records in some

categories than in others. If thre is a big unbalance, the biggest category (partition) will take much longer, causing lots of shuffling and spill.

What can we do to mitigate skew?

In case of OOM issues, you can feel tempted of increasing the RAM of your cluster's workers. This might fix the issue and make your job run, but that won't fix the root of the problem and it might push the problem to later time.. If skew is detected, the first thing to solve is the uneven distribution of records across all partitions. For this purpose we can:

- If in Spark 3, Enable AQE (Adaptative Query Execution).
- If in Databricks, specific skew hint (Skew Join Optimization).
- Otherwise, apply Key Salting. *Salt the skewed column with a random number creating better distribution across each partition at the cost of extra processing*.

Keep in mind that when applying these solutions the job duration can even increase, but remember that even more important than duration, mitigating skew removes potential OOM errors.

How to implement Key Salting for skewed joins?

The general idea of the Key Salting consists on reducing the number of partitions, by increasing the number of join keys. For that we can create a new column wich values are based on the join key, plus a random number within a range. This needs to be applied to all the involved dataframes that are joined by the affected key. After this,

we can perfom the join operation using the new key. As a result, we will end up with more and smaller partitions, and tasks.

The random number range must be chosen after experimentation. In case we choose a large number, we can end up with too many small partitions, and if it's too less, we can keep having the skew problem.

Implementing Key Salting can require lot of effort, consider invest evergy to salt only skewed keys.

Spill

In Spark, this is defined as the act of moving a data from memory to disk and vice-versa during a job. This is a defensive action of Spark in order to free up worker's memory and avoid OOM errors when a partition is too large to fit into memory. In this way, the Spark job can come to an end by paying the high penalty of the compute time caused by the read-write overhead of spilling data.

There are several ways we can get into this problem:

- Having set a `spark.sql.filesMaxPartitionBytes` too high (the default is 128MB) thus ingesting large partitions that might not fit in memory.
- The `explode()` operation of even a small array. Each partition will end up with as much rows as items in the array, therefor the resulting partition size might not fit in memory.
- The `join()` or `crossJoin()` of two tables.

- Aggregating results by a skewed key. As we saw before, having unbalanced datasets can lead to bigger partitions than others, and in some cases this bigger partitions might not fit in memory.

Detecting Spill

In the Spark UI this is represented by:

- **Spill (Memory)**: size in memory of the spilled partitions
- **Spill (Disk)**: size in disk of the spilled partitions (allways smaller than memory because of compression).

This values are only represented in the details page for a single stage (summary metrics, aggregated metrics by executor, task table) or in SQL query details. This makes it hard to recognize because you have to manually search for it. Be aware that in case there is no spill, you won't find these values.

An alternative to manual search for spill, is implementing a SpillListener to track automagically when a stage spills. Unfortunately, this is only available in Scala.

What can we do to mitigate spill?

Once we find our stages are spilling we have a few options to mitigate it:

- Check if spill is caused by data skew, in that case, mitigate that problem first.

- If possible, increase the memory of the cluster's workers. In this way, larger partitions will fit in memory and Spark won't need to write that much into disk.
- Decrease the size of each partition by increasing the number partition. We can do this by tuning the `spark.sql.shuffle.partitions` and `spark.sql.maxPartitionBytes`, or explicitly repartitioning.

Mitigaing spill is not always worth it, so check first if it makes sense or not.

Shuffle

Shuffle is a natural operation of Spark. It's just a side effect of wide transformations like joining, grouping, or sorting. In these cases, the data needs to be shuffled in order to group records with the same keys together under the same partitions for later on being able to execute the aggregations by those keys. When a wide transformation happens, partitions are written to disk, so the executors of the next stage can read the data and continue the job. Because the data needs to be moved across workers, this behaviour can result in lots of network IO.

As I have mentioned before, shuffling it's inevitable. Just put the focus only on the most expensive operations. At same time, be aware that targeting other problems like skew, spill, or tiny files problems is often more effective.

What can we do to mitigate the impact of shuffles?

The biggest pain point of shuffles is the amount of data that needs to be moved across the cluster. In order to reduce this problem, we can apply the following strategies:

- **Use fewer and larger workers to reduce network traffic**. Having same number of executors (CPUs) in less workers, would reduce the amount of data that needs to be transferred throught the network to other workers.
- **Reduce the amount of data being shuffled**. Sometimes these wide transformations are performed over data that is not needed for the final result, increasing the cost unnecesarily. Therefore, we should filter out those columns and rows that are not required before the execution of a wide transformation.
- **Denormalize datasets**. In case a query that causes expensive shuffling is executed very often by data users, the output of this query can be persisted in a data lake and can be queried directly.
- **Broadcast the smaller table**. When one of the tables involved in a join is way smaller than the others, we can broadcast it to all the executors, so it will be fully present there and Spark will be able to join it to the other table partitions without shuffling it. This is called BroadcastHashHoin and it can be applied by using `.broadcast(df)`. However, the default table size threshold is 10MB. We can increase this number by tuning the `spark.sql.autoBroadcastJoinThreshold`, but not too much since it puts the driver under pressure and it can result in OOM errors. Moreover, this approach increases the IO between driver and executors, it doesn't work well when many empty partitions, and requires enough memory in driver and executors.

- **Bucketed datasets**. For joins, data can be pre-shuffled and store it by buckets, and optionally sorted per bucket. This is worth it when working with terabyte size, tables are joined quite often, and no filters are applied. This requires all tables involved to be bucketed by key with same number of buckets (normally one per core). However, the cost to produce and maintain this approach is very high, and must be justifiable.

Storage

The way initial data is ingested in the data lake can lead to problems which are normally related to: tiny files, directory scanning, or schemas.

Tiny Files

We consider tiny files those that are considerably smaller than the default block size of the underlaying file system (128MB).

Having a dataset partitioned in too many tiny files implies longer total time for openening and closing files, and it leads to very bad perfromance. It often relates to a high overhead with ingesting data, or as a result of a Spark job.

We can use the Spark UI to see how many files are read under the SQL tab and checking the read operation.

This problem can be mitigated by:

- **Compacting the existing small files to larger files** equivalent to block size or the efficient partition size used.

- **Make ingesting tools to write bigger files**.

When produced as a result of a **Spark job**, Spark is partitioning the data way more than required for its size, and it's reflected when writing. We can mitigate this by:

- **Changing default partition number**.
  Tuning `spark.sql.shuffle.partitions` (200 by default).
- **Explicitly repartitioning the data before writing**.
  Applying `repartition()` or `coalesce()` functions to decrease the number of partitions, or in case of Spark 3.0+ with AQE enabled set the `spark.sql.adaptive.coalescePartitions.enabled` to true.

Directory Scanning

Having too many directories for some dataset (because of data partioitning) lead to performance issues at scanning time. Too partitioned datasets with no much data also ends up into tiny files problem

We can detect this by paying attention to the scanning time under the SQL tab read operation.

We can mitigate this problem by:

- **Partitioning stored data in a smarter way**.
- **Registering datasets as a tables**. When doing so, metadata like where to find the files belonging to that dataset are stored in the Hive Metastore, thus it's not needed to scan the directory

anymore. However, first time we register the table, it will need some time to retrieve the metadata by scanning direcotries first.

Schemas

Inferring schemas require a full read of a file to determine the data type of each column. This involves time for opening up and scan the files. Reading parquet files requires a one-time read of the schema, because schema is included on the files itself. On the other hand, supporting schema evolution is potentially expensive if you have hundred or thousands of part-files, each schema has to be read in and then merged collectively, and that might be really expesive. Schema merging can be enabled via `spark.sql.parquet.mergeSchema`.

We can mitigate the schema problem by:

- **Providing schema every time**.
- **Registering datasets as tables**. In this way the schema will also be stored in the Hive Metastore.
- **Using Delta format**. Merges schemas automatically for supporting schema evolution.

Serialization

It happens when we need to apply a non-native API transformation, known as UDFs. This implies to serialize the data into a JVM object to be modified outside Spark. The impact of this is way worse in Python, since JVM objects are not native for Python, so they need to be transformed first, execute the code on top of it, and seriallize the

result back to JVM object, causing a big overhead. On the other hand, this part is not needed in Scala since it's JVM native language.

In any case, UDFs are a barrier for the Catalyst Optimizer, since it can't connect the code before and after applying the UDF, since it's impossible for it to know what the UDFs are doing, and how to optimize the overal job execution.

We can mitigate serialization issues by:

- **Avoid using UDFs, Pandas UDFs or Typed Transformations** whenever possible and use native Spark high order functions instead.
- **If there is no other option**:
  – **Python**: use **Pandas UDF** over "standard" Python code. Pandas UDF uses PyArrow to serialise batches of records (treated as a Pandas Series or Data Frame) to later apply the UDF to every record in Python. On the other hand, regular Python UDFs serialises every single record individually, and executes the UDF on it.
  – **Scala**: use **Typed Transformations** over "standard" Scala code.

If your data transformations require the application of many UDFs, consider Scala as a programming language.

In Apache Spark if the data does not fits into the memory then Spark simply persists that data to disk.

The persist method in Apache Spark provides 6 persist storage levels to persist the data. That are as follows:

1. **MEMORY_ONLY**

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_ONLY**)*

useDisk	useMemory	useOffHeap	deserialized	replication
⊗	✓	⊗	✓	1

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

2. **MEMORY_AND_DISK**

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_AND_DISK**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✓	✓	⊗	✓	1

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

3. **MEMORY_ONLY_SER**

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_ONLY_SER**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✗	✓	✗	✗	1

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

4. **MEMORY_AND_DISK_SER**

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_AND_DISK_SER**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✓	✓	✗	✗	1

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

5. **DISK_ONLY**

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**DISK_ONLY**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✓	⊗	⊗	⊗	1

Store the RDD partitions only on disk.

6. **OFF_HEAP**

Persistence (caching) - StorageLevel

OFF_HEAP (experimental)

useDisk	useMemory	useOffHeap	deserialized	replication
⊗	⊗	✓	⊗	1

Store RDD in serialized format in Tachyon.

Which Storage Level to Choose?

- RDD fits comfortably in memory, use MEMORY_ONLY
- If not, try MEMORY_ONLY_SER
- Don’t persist to disk unless, the computation is really expensive.