How does Apache Spark run on a cluster?

towardsdatascience.com/how-does-apache-spark-run-on-a-cluster-974ec2731f20

Whether you are a Data Engineer or a Data Scientist, getting up and running with Apache Spark is a relatively easy process from a development perspective. It does require a slight change in paradigm thinking and understanding how Spark executes code and how it functions on our clusters is an important part of being efficient when using Spark.

What actually happens when we execute code through Spark? How does Apache Spark run on our cluster? This aim of this blog post is to cover that and go into depth into how Spark code runs.

This post will cover:

- Components and overall architecture of a Spark Application.
- The life cycle of a Spark Application (Outside and Inside Spark)
- Low-level execution properties of Spark.

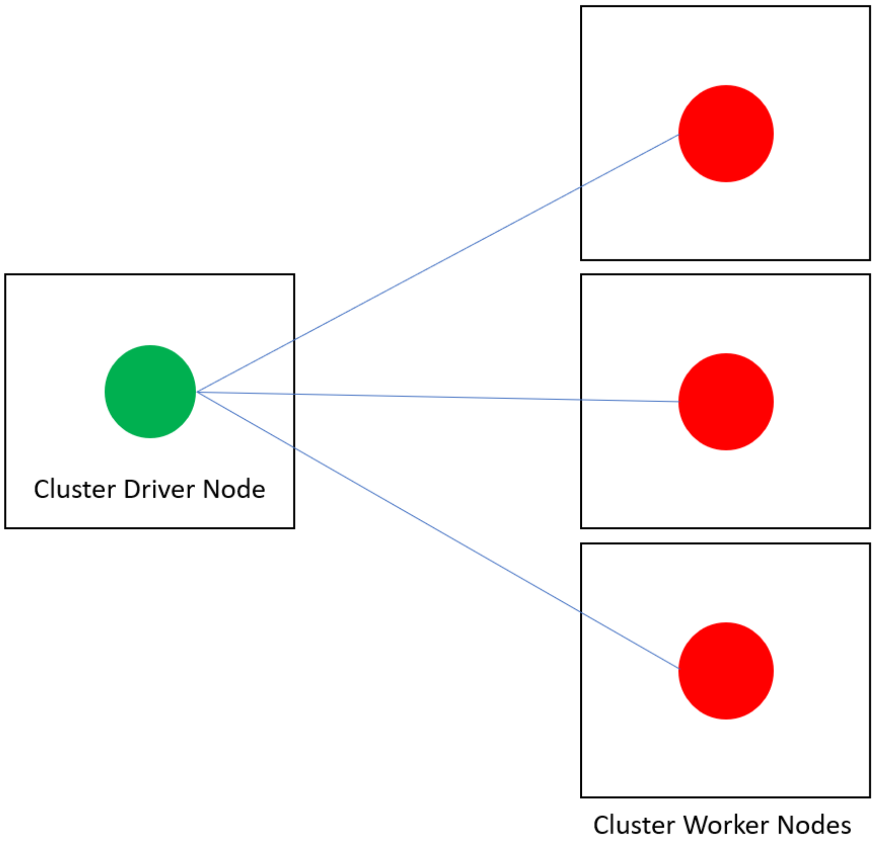
The Architecture of a Spark Application

The high level components of a Spark application include the Spark driver, the Spark executors and the Cluster Manager.

The Spark driver is really the process that is in the driver seat of your Spark Application. It controls the execution of a Spark application and maintains all of the state of the Spark cluster, which includes the state and tasks of the executors. The driver must interface with the cluster manager in order to get physical resources and launch executors. To put this in simple terms, this process is just a process on a physical machine that is responsible for maintaining the state of the application running on the cluster.

The Spark executors are the processes that perform the tasks assigned by the Spark driver. They have one responsibility, take the tasks assigned to them by the driver, run them and report back their state and results.Each Spark Application will have its own separate executor processes.

Finally, we have the Cluster Manager. This is responsible for maintaining a cluster of machines that will run your Spark Application. Cluster managers have their own 'driver' and 'worker' abstractions, but the difference is that these are tied to physical machines rather than processes. The following diagram will help explain this:



Cluster Driver and work

The circles in the above diagram represent daemon processes running and managing each of the worker nodes.

When it comes time to actually run a Spark Application, we request resources from the cluster manager to run it. This will depend on how our application's configuration is set up. Over the course of our Spark application's execution, the cluster manager will be responsible for managing the underlying machines that our application is running on.

At the time of writing, Spark supports three cluster managers:

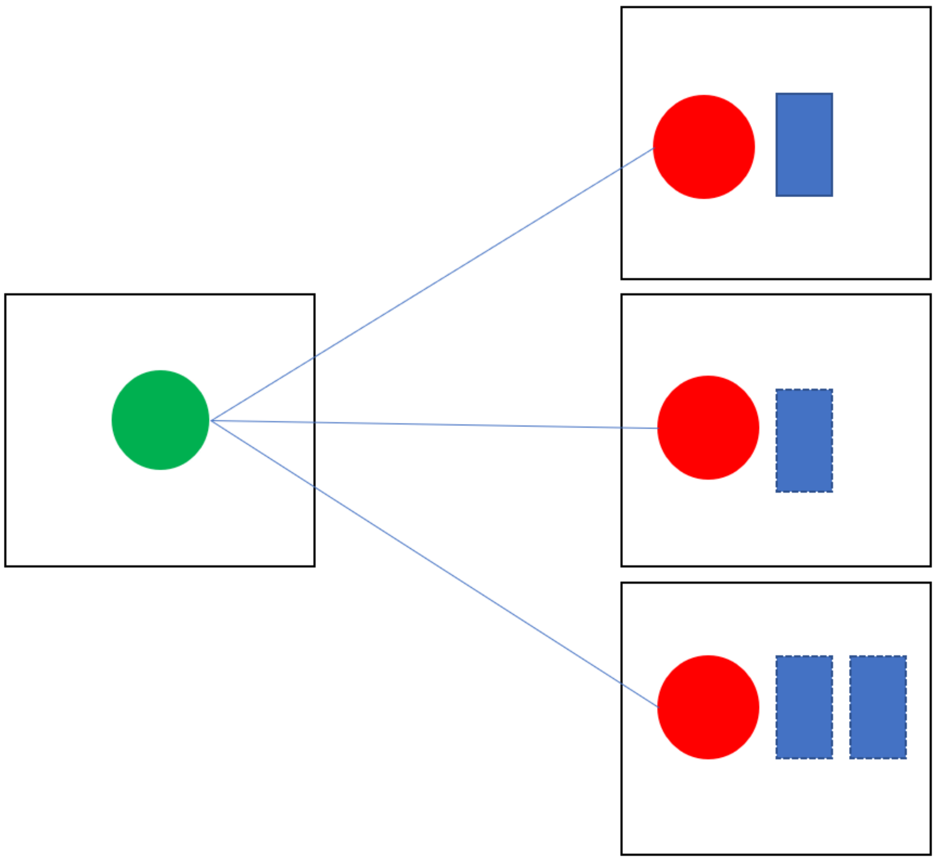
1. Built-in standalone cluster managers.
2. Apache Mesos.

3. Hadoop YARN.

Execution Modes

Execution modes give you the power to determine where the resources are physically location when you run your application. There are 3 modes to choose from: Cluster Mode, Client Mode and Local Mode.

Cluster mode is probably the most common way of running Spark Applications. In cluster mode, A user submits a pre-complied JAR, Python Script or R script to the cluster manager. The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes. The cluster manager is responsible for maintaining all Spark Application related processes. The below diagram represents the cluster manager and how it places our driver on a worker node and the executors on other worker nodes.



Spark’s cluster mode

Client mode is almost the same as cluster mode, however the Spark driver remains on the client machine that submitted the application. This means that the client machine is responsible for maintaining the Spark driver process and the cluster manager maintains the executor processes.

Local mode is very different from both cluster and client modes. Local mode runs the entire Spark Application on a single machine and it achieves parallelism through threads on that single machine. We use this mode as a way to learn Spark, test applications or just experiment with Spark. Don't use this mode for running applications that you deploy to production.

The Life cycle of a Spark Application outside of Spark

Let's discuss the life cycle of a Spark application outside the code.

The first step is for you to submit an actual application in the form of a pre-complied JAR or library. You're likely executing code on your local machine and you're going to make a request to the cluster manager driver node. We ask for resources for the Spark driver process only and the cluster manager places the driver onto a node in the cluster. The client then exists the process and the application starts running on the cluster.

The command will look something like this:

```
./bin/spark-spark submit --class <main-class> --master <master-url> --deploy-mode
cluster --conf <key>=<value> <application-jar>
```

Once the driver process has been placed on the cluster, it begins running our code. This should include a SparkSession that initializes a Spark cluster. SparkSession will communicate with the cluster manager, asking it to launch Spark executor processes across our cluster. The number of executors and relevant configurations are set by the user via command-line arguments in the original spark-submit call.

We now have a Spark Cluster that will execute our code. The driver and the workers will communicate with each other to execute our code and move the data around. Each worker will be scheduled with tasks by the driver and will respond with the status of those tasks whether they failed or not.

After a Spark Application finishes, the driver process exits with either the task succeeding or failing. The Cluster Manager will then shut down the executors in that Spark cluster for the driver and we can see whether the Spark Application failed or succeeded by asking the cluster manager.

The Life cycle of a Spark Application (Inside Spark)

Now let’s take a look at the life cycle of a Spark Application from inside our user-code. Each application that we create will consist of one or more Spark jobs and Spark jobs are executed serially, unless we use threading to launch multiple actions in parallel.

Creating a SparkSession is the first step in any Spark Application. You may see some legacy code use

```
new SparkContext
```

This should be avoided in favor of the builder method on the SparkSession, which more robustly instantiates the Spark and SQL Contexts and ensures that there is no context conflict, given there might be multiple libraries trying to create a session in the same Spark application.

To create a new SparkSession in Scala, you can do it like this:

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder().appName("Spark App")
                               .config("spark.sql.warehouse.dir",
"/user/hive/file").getOrCreate()
```

In PySpark, you would do it like this:

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.master("local").appName("Spark App")
                        .config("spark.some.config", "some-value").getOrCreate()
```

Once we've created a SparkSession, we can run our Spark code. From the SparkSession, we can access all of low-level and legacy contexts and configurations that we may need.

A SparkContext object within the SparkSession represents the connection to the Spark cluster. This is how we can use some of Spark's lower-level APIs, such as RDDs, accumulators etc. You can create RDD's, acculumaltors, and broadcast variables and you can run code on the cluster via a SparkContext.

We shouldn't have to initialize a SparkContext, we can access it through our SparkSession. If we do need to initialize it, we can do it like this:

```
import org.apache.spark.SparkContext

val sc = SparkContext.getOrCreate()
```

Now that we have a SparkSession, we can execute some code. All Spark code compiles down to RDD's, so let's take a job and break it down step by step to see what happens.

Logical Instructions to Physical Execution

Spark code is made up of transformations and actions. We can take DataFrames and convert them into physical execution plans. This is super important in understanding how Spark runs on a cluster.

Let's use a Python example to show this process. I've taken this example from Spark: The Definitive Guide book and I'll try and simplify their explanation:

```
df1 = spark.range(2, 1000000, 2)
df2 = spark.range(2, 1000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")

step4.collect()
```

When we run the above code, we can see that your action triggers one complete Spark job. We can access the explain plan via the Spark UI to see how this job has been executed (I'll do a post on the Spark UI at a later date).

When we call an action, we get the execution of a Spark job that consists of stages and tasks. Let's cover those now.

Spark Jobs

In general, there should be one Spark job for one action and actions always return results. Each job that we run will break down into a series of stages, the number of which depends on how many shuffle operations need to take place.

Stages

Stages represent groups of tasks that can be executed together to compute the same operation on multiple machines (a cluster!).

In general, Spark will try to pack as much work as possible (i.e as many transformations as possible inside your job) into the same stage, but the engine starts new stages after operations called shuffles.

A shuffle essentially represents a physical re-partitioning of the data. This could be sorting a DataFrame, or grouping data that was loaded from a file by key. Re-partitioning requires coordinating across executors to move data around. Spark starts a new stage after each shuffle, and keeps track of what order the stages must run in to compute the final result. A good rule to go by is that the number of partitions should be larger than the number of executors on your cluster, potentially by multiple factors depending on your workload.

Tasks

Now we can look at tasks. Stages are made up of tasks. Each task corresponds to a combination of blocks of data and a set of transformations that will run on a single executor. So for example, if we have a single large partition in our data set, we will only have one task. If there are 1,000 little partitions, we will have 1,000 tasks.

Tasks are just units of computation applied to a unit of data. By partitioning our data into a greater number of partitions, this means we can execute more tasks in parallel. This is a simple way to start optimizing our code.

Execution Details

Let's finish off this post by looking into the execution details. Spark tasks and stages in Spark have some important properties that we should take a look at. Spark will automatically pipeline stages and tasks that can be done together and for all shuffle operations, Spark writes the data to stable storage and can reuse it across multiple jobs.

When we look at the Spark UI to see how our jobs, we will need to know how Pipelining and shuffle persistence works.

Pipelining

What makes Spark such a great in-memory computation tool is that it performs as many steps at one point in time before it writes data to memory or disk. Pipelining is a key optimization and this occurs at and below the RDD level. Pipeline allows any sequence of operations that feed data directly into each other, without having to move it across nodes in our cluster. This is then combined into a single stage of tasks that perform all the required operations together.

Pipelining is transparent to us as we write Spark code. Spark's runtime automatically does this for us. Should we ever have to inspect our application via the Spark UI, then we'll see it there, or our logs.

Shuffle Persistence

The second property you'll see is shuffle persistence. When Spark needs to run an operation that has to move data across nodes, such as a reduce-by-key operation (where input data for each key needs to first be brought together from many nodes) the engine won’t be able to pipeline anymore and will instead revert to performing a cross-network shuffle.

Spark always shuffles by having tasks sending data write shuffles to local disks during the execution stage. The stage then groups and reduced launches and runs tasks that fetch their records from each shuffle file that performs our computation.

By saving the shuffle file to disk, this lets Spark run the stage later in time than the source stage and lets the engine re-launch failed reduce tasks without rerunning all the required input tasks.

Running a new job over data that’s been shuffled already does not rerun the source side of the shuffle. Spark already knows that it can use them to run the later stages of the job.

When we observe the job in the Spark UI, we’ll see these pre-shuffle stages marked as skipped. This will save us time in a workload that runs multiple jobs over data.

However, we can further improve this by caching the DataFrame, which gives us more control over where the data is saved.

Conclusion

I hope after reading this post, you have a better idea of how Spark Applications run on clusters and hopefully you have a better idea about how clusters actually run our Spark code.

This should give us a better understanding as to how we debug Spark Applications.