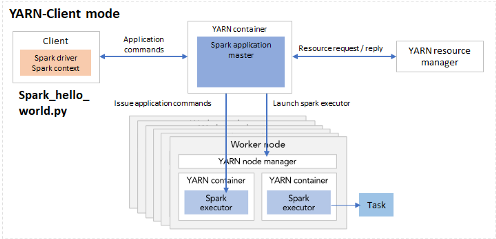
Learning Spark is not an easy thing for a person with less background knowledge on distributed systems. Even though I have been using Spark for quite some time, I find it time-consuming to get a comprehensive grasp of all the core concepts in Spark. The [official Spark documentation](https://spark.apache.org/docs/latest/?source=post_page---------------------------) provides a very detailed explanation, yet it focuses more on the practical programming side. Also, tons of online tutorials can be overwhelming to a starter. Therefore in this article I would like to note down those Spark **core concepts,**but in a more visualized way. Hope you will find it useful as well!

Note: probably you already have some knowledge about Hadoop, so I will skip explanations on trivial things such as nodes and clusters.

**Spark architecture and deploy modes**

To put it simple, Spark runs on a master-worker architecture, a typical type of parallel task computing model. When running Spark, there are a few modes we can choose from, i.e. local (master, executor, driver are all in the same single JVM machine), standalone, YARN and Mesos. Here we only talk about Spark on YARN and the difference between YARN client and YARN cluster since both are most commonly used, yet very confusing.

Below two pictures illustrate the setup for both modes. They look quite similar, don’t they? However, by looking at the orange highlighted part you will probably notice the minor difference, which is the location of Spark driver program. This is basically the only difference between the two modes.



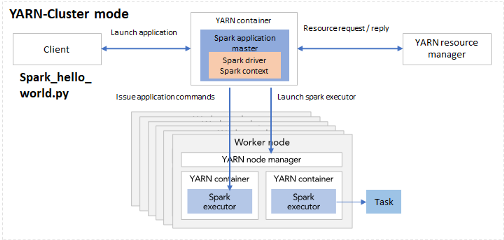


Fig 1. Spark deployment mode YARN-client (left) and YARN-cluster (right)

Suppose you’ve written a Spark application called spark\_hello\_world.py. In *client* mode, when executing the python file using spark-submit, the driver is launched directly within the spark-submit process, hence it will reside in the same machine as with spark\_hello\_world.py. When initializing the Spark context, the driver within the local machine will connect to the application master in the cluster. Starting from the master, Spark launch more executors.

In *cluster* mode, the spark\_hello\_world.py code lives in the client machine and the client machine is outside of the cluster. When executing the application python code, it launches a driver program in one of the nodes in the cluster. Together with Spark application master it can launch executors and issue application commands.

Given that the setup do not differ much, you must be wondering why we need two different modes. In practice, this relates to whether the client machine is physically co-located with the worker machines or not. If the client machine is “far” from the worker nodes, e.g. you write the spark\_hello\_world.py on your laptop but the workers are AWS EC2 instances, then it makes sense to use cluster mode, so as to minimize network latency between the drivers and the executors. In another scenario, if your python file is in a gateway machine quite “close” to the worker nodes, the client mode could be a good choice.

**Executors**

Now that we understand the Spark cluster setup, let’s zoom in to one of the most important elements in Spark - executor. Executors are the processes that run tasks and keep data in memory or disk storage across them.

When going through the Spark documentation you might be surprised at the number of configurable parameters related to executors. Instead of trying hard to figure out the relation between several parameters in one’s head again and again, let’s look at it visually.

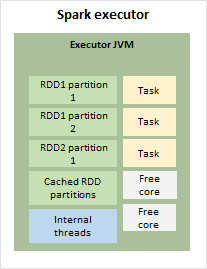


Fig 2. Spark executor internals

As shown in Figure 2, in each executor there is an executor JVM, storing the RDD partitions, cached RDD partition, running internal threads and tasks. If there are more cores than required by the tasks, there would also be free cores in the JVM. This green block of executor JVM will be our starting point to look at the memory management in executors.

**Executor memory management**

In the executor container, there are mainly two blocks of memory allocated: memory overhead and executor memory.

*Memory overhead* is reserved off-heap memory for things like VM overheads, interned strings, other native overheads, etc.. By caching data outside of main Java heap space, but still in RAM, the off-heap memory allows the cache to overcome lengthy JVM Garbage Collection pauses when working with large heap sizes.

*Executor memory* consists of three parts as follows.

* Reserved memory
* User memory: for storing things such as user data structures and internal metadata in Spark.
* Storage and execution memory: for storing all the RDD partitions and allocating run-time memory for tasks.

Figure 3 shows the relevant parameters for each memory block. Suppose we set spark.executor.memory to 4 GB, then Spark will request 4.4 GB memory in total from the resource manager. Out of the 4 GB executor memory, we actually get 3.7 GB because the rest is reserved. And by default, we get 2.2 GB (0.6 \* 3.7) as execution + storage memory. Out of this, 1.1 GB will be used for storage such as storing RDDs, and the rest will be execution memory.

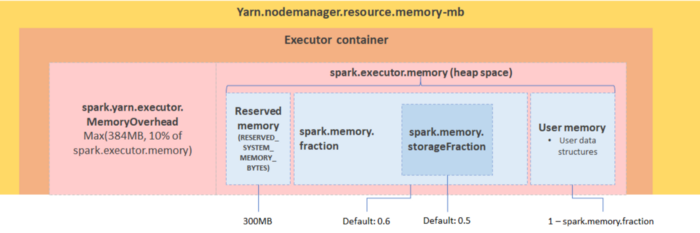


Fig 3. Spark executor memory decomposition

**RDD, jobs, stages and tasks**

If you have already started debugging Spark application using Spark UI, then probably keywords like jobs, stages and tasks sound familiar. So how are they relevant with RDDs?

We know that there are two operations on RDDs, *transformations* (e.g. filter, union, distinct, intersection) by which a new RDD is produced from the existing one virtually without actual execution, and *actions* (e.g. take, show, collect, foreach) which triggers the execution. When transforming an RDD, based on the relationship between the parent RDD and the transformed RDD, the dependency can be *narrow* or *wide*. With narrow dependency, in the parent RDD one or many partition will be mapped to one partition in the new RDD. While with wide dependency, such as when doing a join or sortBy, we need to *shuffle* partitions in order to compute the new RDD.

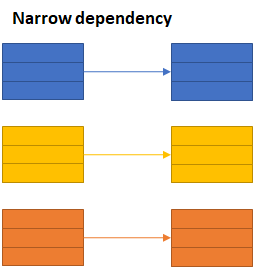


Fig 4–1. narrow dependency in RDD transformation

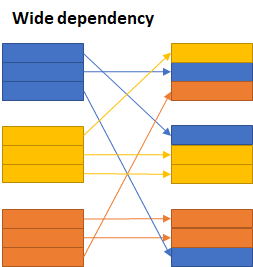


Fig 4–2. Wide dependency in RDD transformation

The *jobs*, *stages* and *tasks* are therefore determined by the type of operations and the type of transformations. A *job* is created when there is an *action* on an RDD. Within the job, there could be multiple *stages*, depending on whether or not we need to perform a wide transformation (i.e. shuffles). In each stage there can be one or multiple transformations, mapped to *tasks* in each executor.

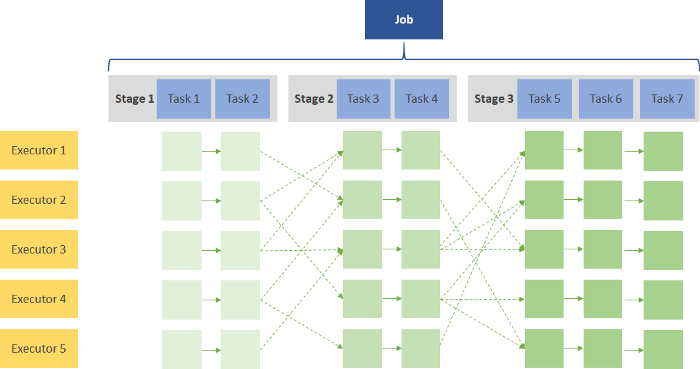


Fig 5. Illustration of one Spark job

To understand it practically let’s look at the following simple code snippet.

1. val RDD1 = sc.parallelize(Array('1', '2', '3', '4', '5')).map{ x => val xi = x.toInt; (xi, xi+1) }  
2. val RDD2 = sc.parallelize(Array('1', '2', '3', '4', '5')).map{ x => val xi = x.toInt; (xi, xi\*10) }  
3. val joinedData = RDD2.join(RDD1)  
4. val filteredRDD = joinedData.filter{case (k, v) => k % 2 == 0}  
5. val resultRDD = filteredRDD.mapPartitions{ iter => iter.map{ case (k, (v1, v2) ) => (k, v1+v2) } }  
6. resultRDD.take(2)

There are a few operations in this code, i.e. *map*, *join*, *filter*, *mapPartitions*and *take*. When creating the RDDs Spark will generate two stages for RDD1 and RDD2 separately, as shown in stage 0 and 1. Since *map* function contains a narrow dependency, the mapped RDDs will also be included in stage 0 and 1 respectively. Then we join RDD1 and RDD2, because *join* is a wide transformation containing shuffles, Spark creates another stage for this operation. Afterwards, *filter* and *mapPartition* are again a narrow transformations in stage 2, and by calling *take* (which is an action), we trigger Spark’s execution.

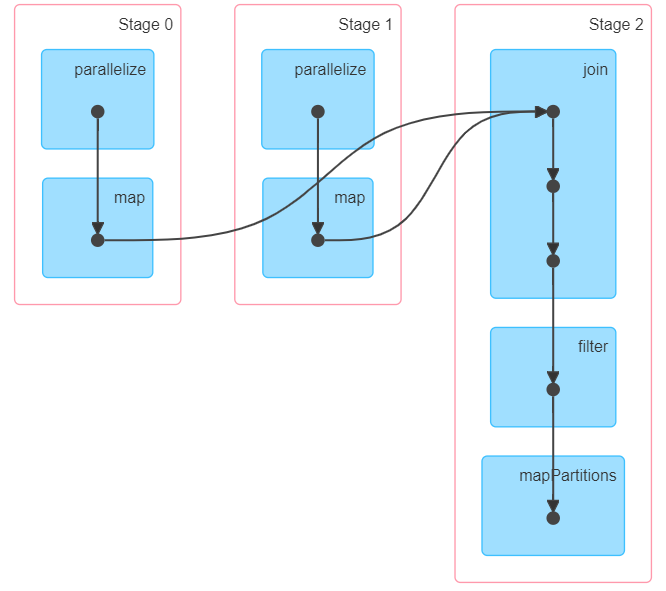


Fig 6. DAG visualization

So, that is all the basic stuff for Spark. Hope after reading this article these concepts are more clear for you. Happy learning!

**References**

* [https://spark.apache.org/docs/latest/](https://spark.apache.org/docs/latest/?source=post_page---------------------------)
* [https://spoddutur.github.io/spark-notes/distribution\_of\_executors\_cores\_and\_memory\_for\_spark\_application.html](https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html?source=post_page---------------------------)​
* [https://0x0fff.com/spark-memory-management/](https://0x0fff.com/spark-memory-management/?source=post_page---------------------------)​
* [https://www.pgs-soft.com/blog/spark-memory-management-part-1-push-it-to-the-limits/](https://www.pgs-soft.com/blog/spark-memory-management-part-1-push-it-to-the-limits/?source=post_page---------------------------)​
* [https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations](https://spark.apache.org/docs/latest/rdd-programming-guide.html?source=post_page---------------------------#rdd-operations)​

**Any feedback and comments are welcome. Your support means a lot to an author! ❤**