Spark Architecture

73 Replies

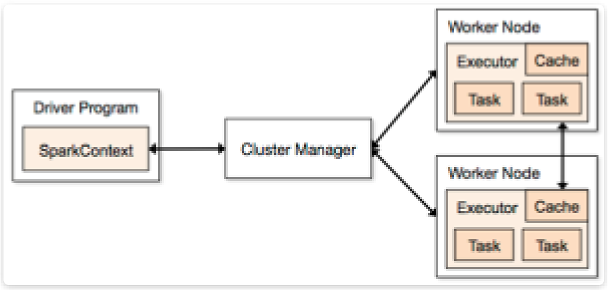
Edit from 2015/12/17: Memory model described in this article is deprecated starting Apache Spark 1.6+, the new memory model is based on UnifiedMemoryManager and described in this article

Over the recent time I've answered a series of questions related to ApacheSpark architecture on StackOverflow. All of them seem to be caused by the absence of a good general description of the Spark architecture in the internet. Even official guide does not have that many details and of cause it lacks good diagrams. Same for the "Learning Spark" book and the materials of official workshops.

In this article I would try to fix this and provide a single-stop shop guide for Spark architecture in general and some most popular questions on its concepts. This article is not for complete beginners – it will not provide you an insight on the Spark main programming abstractions (RDD and DAG), but requires their knowledge as a prerequisite.

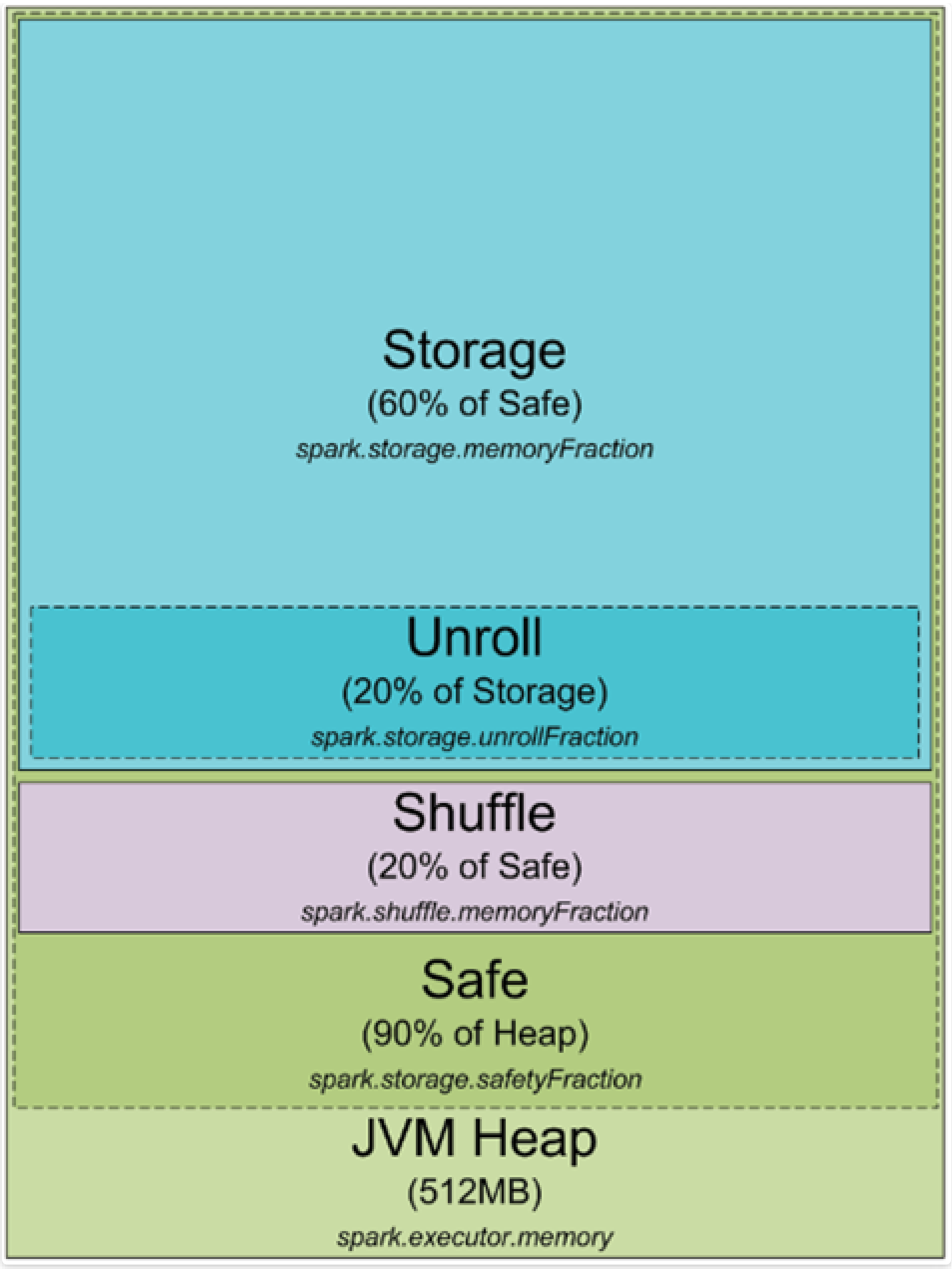
This is the first article in a series. The second one regarding shuffle is available here. The third one about new memory management model is available here.

Let's start with the official picture available on the http://spark.apache.org/docs/1.3.0/cluster-overview.html:



As you might see, it has many terms introduced at the same time – "executor", "task", "cache", "Worker Node" and so on. When I started to learn the Spark concepts some time ago, it was almost the only picture about Spark architecture available over the internet and now the things didn't change much. I personally don't really like this because it does not show some important concepts or shows them not in the best way.

Let's start from the beginning. Any, any Spark process that would ever work on your cluster or local machine is a JVM process. As for any JVM process, you can configure its heap size with *-Xmx* and *-Xms* flags of the JVM. How does this process use its heap memory and why does it need it at all? Here's the diagram of Spark memory allocation inside of the JVM heap:

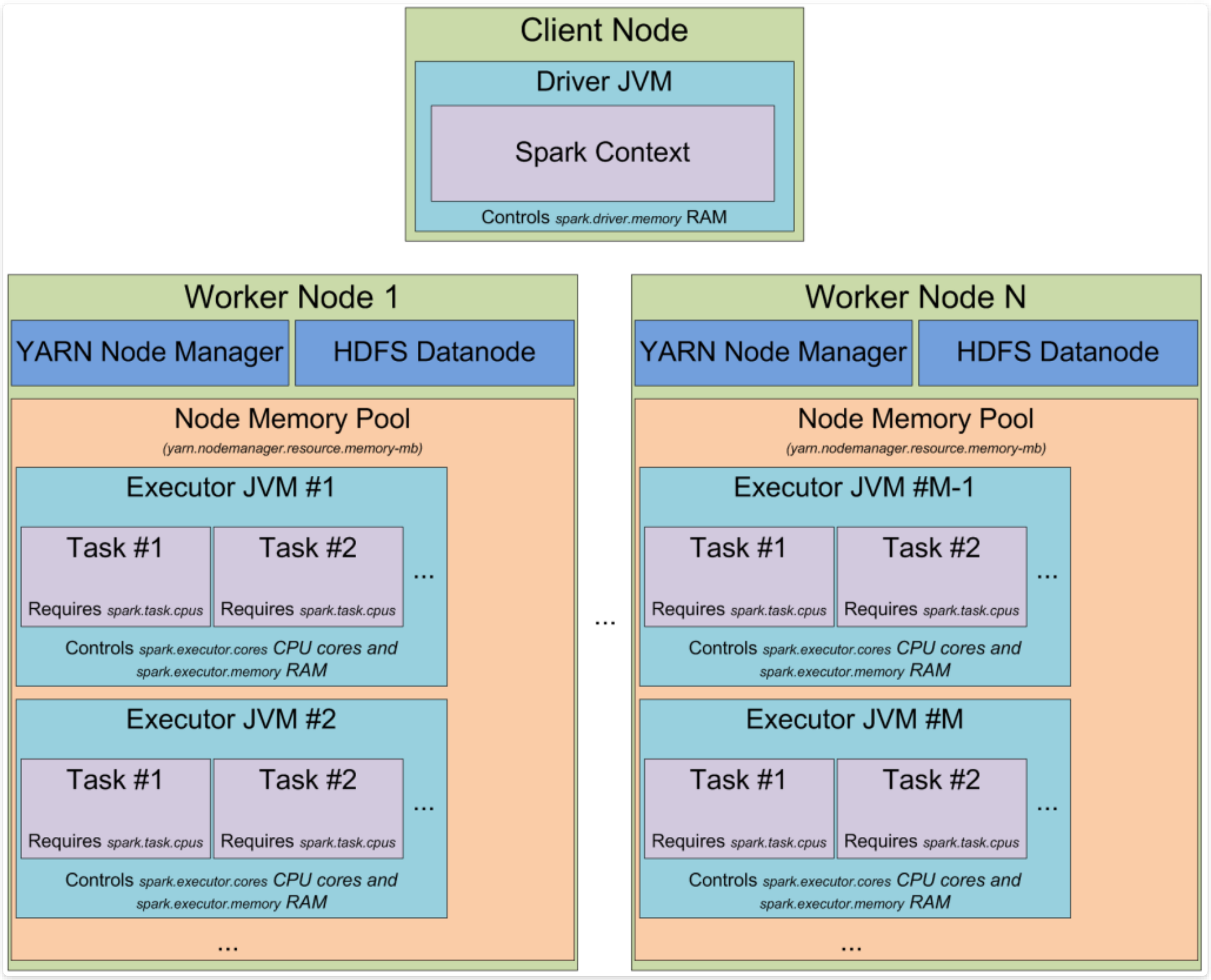


By default, Spark starts with 512MB JVM heap. To be on a safe side and avoid OOM error Spark allows to utilize only 90% of the heap, which is controlled by the *spark.storage.safetyFraction* parameter of Spark. Ok, as you might have heard of Spark as an in-memory tool, Spark allows you to store some data in memory. If you have read my article here https://0x0fff.com/spark-misconceptions/, you should understand that Spark is not really in-memory tool, it just utilizes the memory for its LRU cache (http://en.wikipedia.org/wiki/Cache_algorithms). So some amount of memory is reserved for the caching of the data you are processing, and this part is usually 60% of the safe heap, which is controlled by the *spark.storage.memoryFraction*parameter. So if you want to know how much data you can cache in Spark, you should take the sum of all the heap sizes for all the executors, multiply it by *safetyFraction* and by *storage.memoryFraction*, and by default it is 0.9 * 0.6 = 0.54 or 54% of the total heap size you allow Spark to use.

Now a bit more about the shuffle memory. It is calculated as “Heap Size” * *spark.shuffle.safetyFraction* * *spark.shuffle.memoryFraction*. Default value for *spark.shuffle.safetyFraction* is 0.8 or 80%, default value for *spark.shuffle.memoryFraction*is 0.2 or 20%. So finally you can use up to 0.8*0.2 = 0.16 or 16% of the JVM heap for the shuffle. But how does Spark uses this memory? You can get more details on this here (https://github.com/apache/spark/blob/branch-1.3/core/src/main/scala/org/apache/spark/shuffle/ShuffleMemoryManager.scala), but in general Spark uses this memory for the exact task it is called after – for Shuffle. When the shuffle is performed, sometimes you as well need to sort the data. When you sort the data, you usually need a buffer to store the sorted data (remember, you cannot modify the data in the LRU cache in place as it is there to be reused later). So it needs some amount of RAM to store the sorted chunks of data. What happens if you don’t have enough memory to sort the data? There is a wide range of algorithms usually referenced as "external sorting" (http://en.wikipedia.org/wiki/External_sorting) that allows you to sort the data chunk-by-chunk and then merge the final result together.

The last part of RAM I haven't yet cover is "unroll" memory. The amount of RAM that is allowed to be utilized by unroll process is *spark.storage.unrollFraction* * *spark.storage.memoryFraction* * *spark.storage.safetyFraction*, which with the default values equal to 0.2 * 0.6 * 0.9 = 0.108 or 10.8% of the heap. This is the memory that can be used when you are unrolling the block of data into the memory. Why do you need to unroll it after all? Spark allows you to store the data both in serialized and deserialized form. The data in serialized form cannot be used directly, so you have to unroll it before using, so this is the RAM that is used for unrolling. It is shared with the storage RAM, which means that if you need some memory to unroll the data, this might cause dropping some of the partitions stored in the Spark LRU cache.

This is great, because at the moment you know what exactly Spark process is and how it utilizes the memory of its JVM processes. Now let's switch to the cluster mode – when you start a Spark cluster, how does it really look like? I like YARN so I would cover how it works in YARN, but in general it is the same for any cluster manager you use:



When you have a YARN cluster, it has a YARN Resource Manager daemon that controls the cluster resources (practically memory) and a series of YARN Node Managers running on the cluster nodes and controlling node resource utilization. From the YARN standpoint, each node represents a pool of RAM that you have a control over. When you request some resources from YARN Resource Manager, it gives you information of which Node Managers you can contact to bring up the execution containers for you. Each execution container is a JVM with requested heap size. JVM locations are chosen by the YARN Resource Manager and you have no control over it – if the node has 64GB of RAM controlled by YARN (*yarn.nodemanager.resource.memory-mb* setting in yarn-site.xml) and you request 10 executors with 4GB each, all of them can be easily started on a single YARN node even if you have a big cluster.