Distributed Systems Architecture

brought to you by Alexey Grishchenko

Spark Architecture

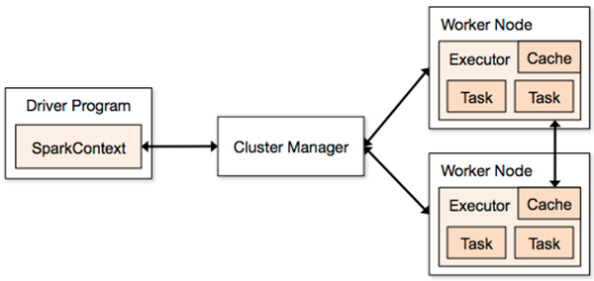
Edit from 2015/12/17: Memory model described in this article is deprecated starting Apache Spark 1.6+, the new memory model is based on UnifiedMemoryManager and described in this article

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.

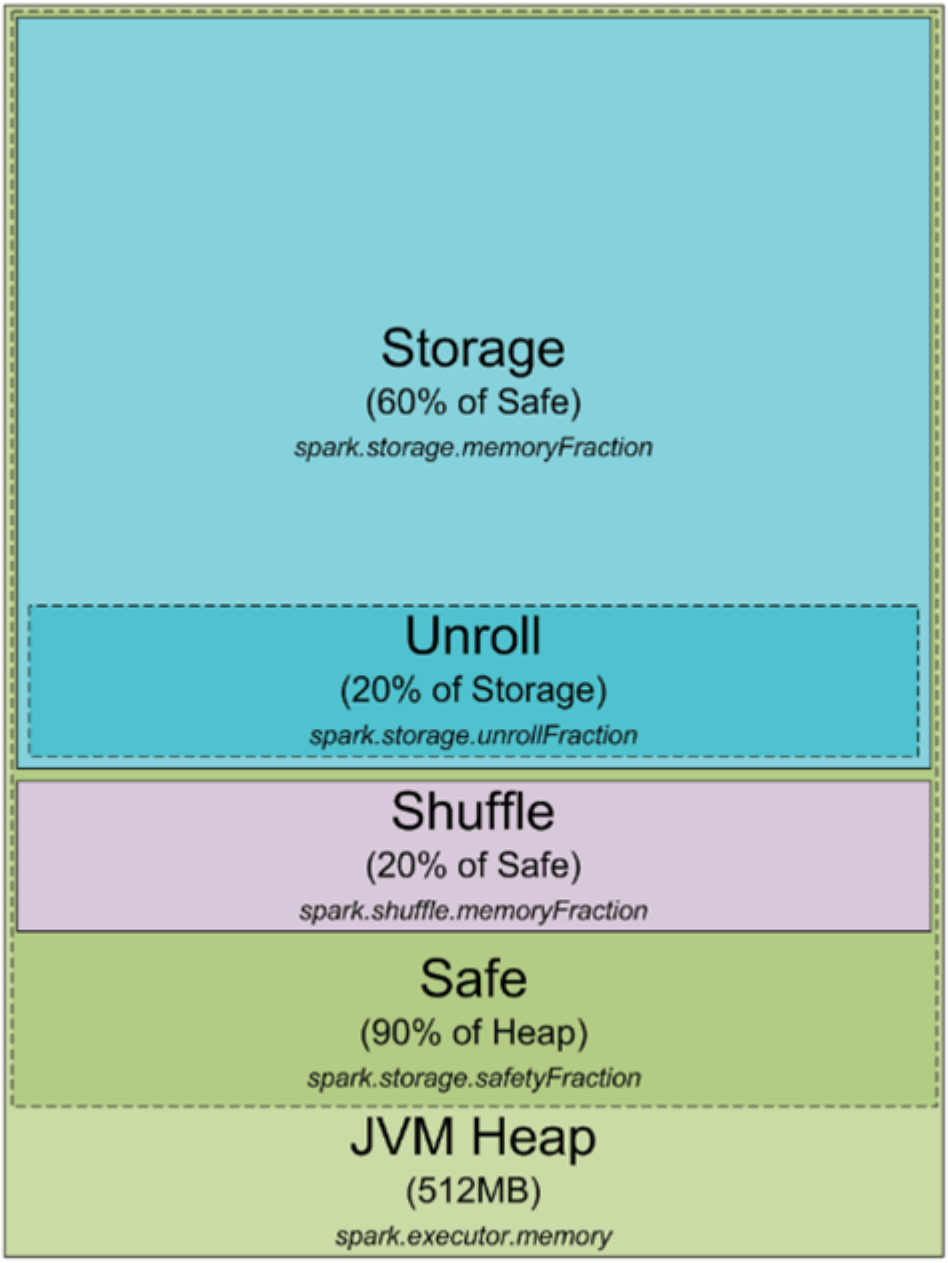
This is the first article in a series. The second one regarding shuffle is available here. The third one about new memory management model is available here.

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:



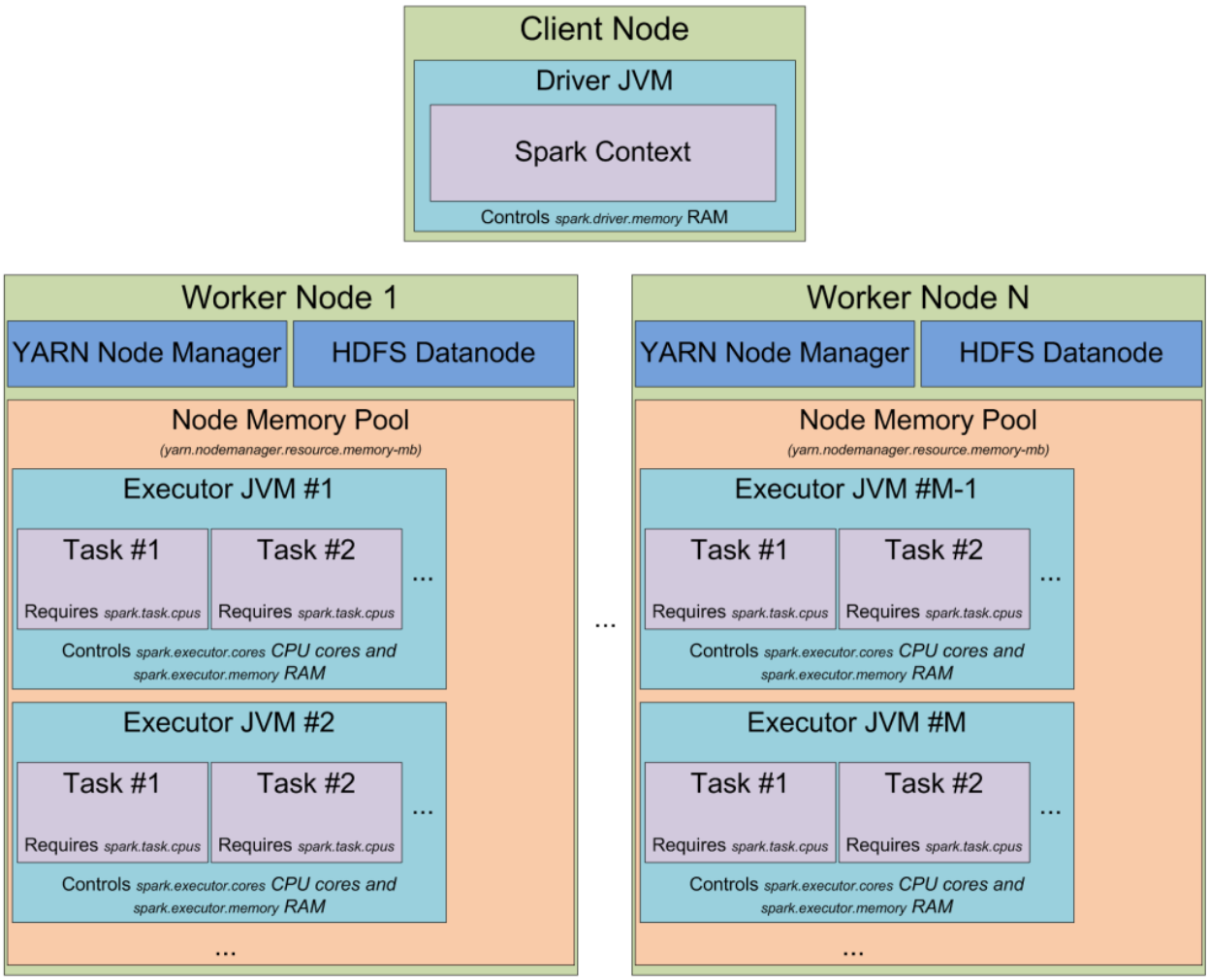
By default, Spark starts with 512MB JVM heap. To be on a safe side and avoid OOM error Spark allows to utilize only 90% of the heap, which is controlled by the *spark.storage.safetyFraction* parameter of Spark. Ok, as you might have heard of Spark as an in-memory tool, Spark allows you to store some 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.

When you start Spark cluster on top of YARN, you specify the amount of executors you need (*–num-executors* flag or *spark.executor.instances* parameter), amount of memory to be used for each of the executors (*–executor-memory* flag or *spark.executor.memory* parameter), amount of cores allowed to use for each executors (*–executor-cores* flag of *spark.executor.cores* parameter), and amount of cores dedicated for each task's execution (*spark.task.cpus* parameter). Also you specify the amount of memory to be used by the driver application (*–driver-memory* flag or *spark.driver.memory* parameter).

When you execute something on a cluster, the processing of your job is split up into stages, and each stage is split into tasks. Each task is scheduled separately. You can consider each of the JVMs working as executors as a pool of task execution slots, each executor would give you *spark.executor.cores* / *spark.task.cpus* execution slots for your tasks, with a total of *spark.executor.instances* executors. Here's an example. The cluster with 12 nodes running YARN Node Managers, 64GB of RAM each and 32 CPU cores each (16 physical cores with hyper threading). This way on each node you can start 2 executors with 26GB of RAM each (leave some RAM for system processes, YARN NM and DataNode), each executor with 12 cores to be utilized for tasks (leave some cores for system processes, YARN NM and DataNode). So In total your cluster would handle 12 machines * 2 executors per machine * 12 cores per executor / 1 core for each task = 288 task slots. This means that your Spark cluster would be able to run up to 288 tasks in parallel thus utilizing almost all the resources you have on this cluster. The amount of RAM you can use for caching your data on this cluster is 0.9 *spark.storage.safetyFraction* * 0.6 *spark.storage.memoryFraction* * 12 machines * 2 executors per machine * 26 GB per executor = 336.96 GB. Not that much, but in most cases it is enough.

So far so good, now you know how the Spark uses its JVM's memory and what are the execution slots you have on your cluster. As you might already noticed, I didn't stop in details on what the "task" really is. This would be a subject of the next article, but basically it is a single unit of work performed by Spark, and is executed as a ***thread*** in the executor JVM. This is the secret under the Spark low job startup time – forking additional thread inside of the JVM is much faster that bringing up the whole JVM, which is performed when you start a MapReduce job in Hadoop.

Now let's focus on another Spark abstraction called "***partition***". All the data you work with in Spark is split into partitions. What a single partition is and how is it determined? Partition size completely depends on the data source you use. For most of the methods to read the data in Spark you can specify the amount of partitions you want to have in your RDD. When you read a file from HDFS, you use Hadoop's InputFormat to make it. By default each input split returned by InputFormat is mapped to a single partition in RDD. For most of the files on HDFS single input split is generated for a single block of data stored on HDFS, which equals to approximately 64MB of 128MB of data. Approximately, because the data in HDFS is split on exact block boundaries in bytes, but when it is processed it is split on the record splits. For text file the splitting character is the newline char, for sequence file it is the block end and so on. The only exception of this rule is compressed files – if you have the whole text file compressed, then it cannot be split into records and the

whole file would become a single input split and thus a single partition in Spark and you have to manually repartition it.

And what we have now is really simple – to process a single partition of data Spark spawns a single task, which is executed in task slot located close to the data you have (Hadoop block location, Spark cached partition location).

This information is more than enough for a single article. In the next one I would cover how Spark splits the execution process into stages and stages into tasks, how Spark shuffles the data through the cluster and some more useful things.

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.

Share this:

This entry was posted in Hadoop, Spark and tagged apache spark, architecture, design, hadoop, HDFS, Spark, yarn on March 14, 2015 [https://0x0fff.com/spark-architecture/] .

61 thoughts on “Spark Architecture”

Raja
March 17, 2015 at 5:06 pm

Nice observation.I feel that enough RAM size or nodes will save, despite using LRU cache.I think incorporating Tachyon helps a little too, like de-duplicating in-memory data and some more features not related like speed, sharing, safe. It is just my opinion.

0x0FFF Post author
March 18, 2015 at 5:09 am

Of course the more RAM you have the faster would be your jobs: more data cached in Linux cache, more data is loaded to RAM in Spark, etc. But in general Tachyon can speed up only in terms of IO when you persist the data, and it is not the main bottleneck in Spark (writing shuffle data to local filesystems usually is).