**Understanding spark architecture in Deep with YARN**

**OVERVIEW**

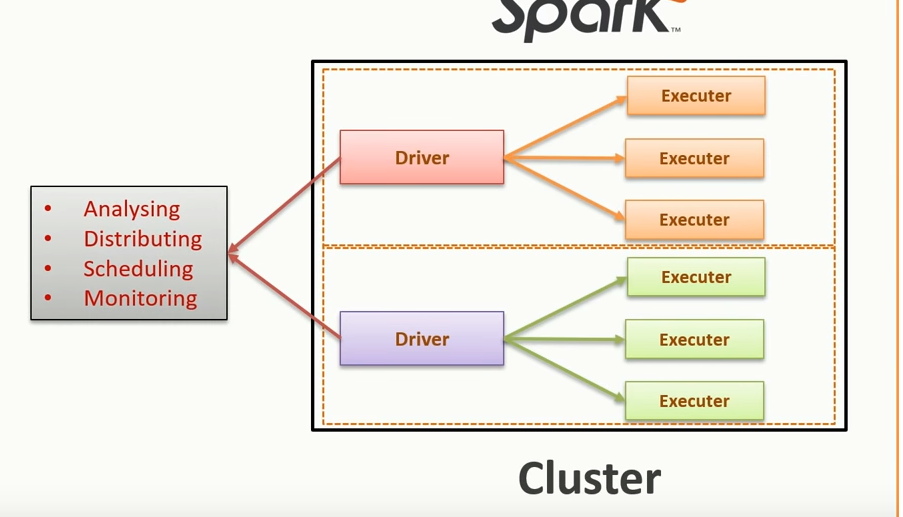
Apache spark is a Distributed Computing Platform.Its distributed doesn’t imply that it can run only on a cluster. Spark can be configured on our local system also. But Since spark works great in clusters and in real time , it is being implemented in multi node clusters like Hadoop, we will consider a Hadoop cluster for explaining spark here.

We can Execute spark on a spark cluster in following ways

* Interactive clients(scala shell,pyspark etc): Usually used for exploration while coding like python shell
* Submit a job (using spark submit utility):Always used for submitting a production application

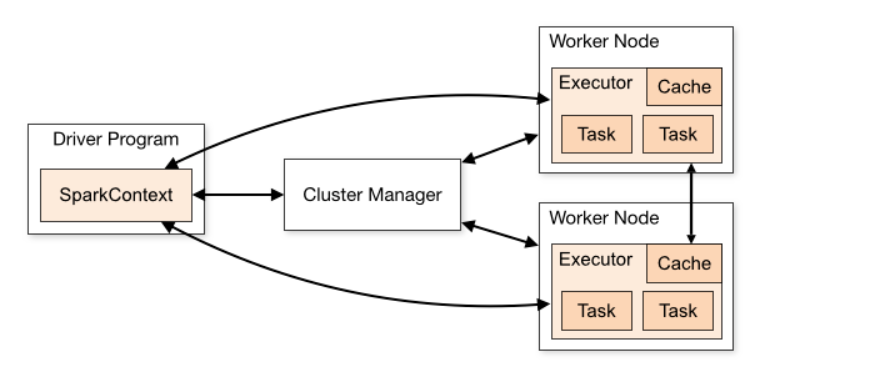
**Basic Architecture**

Spark follows a Master/Slave Architecture. That is For every submitted application, it creates a Master Process and multiple slave processes. Master is the Driver and Slaves are the executors.

Say If from a client machine, we have submitted a spark job to a cluster. Spark will create a driver process and multiple executors. Similraly if another spark job is submitted to same cluster, it will create again “one Driver- Many executors” combo.Thus for every program it will do the same. Driver is responsible for Analyzing, distributing, scheduling and monitoring work across the cluster.Driver is also responsible for maintaining necessary information to executors during the lifetime of the application

**Detailed Architecture**

Below is the general architectural diagram for spark cluster.



Main Components of a Spark Cluster are :

* Worker Nodes
* Cluster Manager
* Driver Program(Spark Context)
* Executor
* Task

**Worked Nodes:**

These are nothing but physical nodes with RAM,CPU,HDD(SSD) etc. RAM configured will be usually high since spark utilizes in-memory computation of high volumes of data. And these RAM,CPU,HDD,Network Bandwidth etc are called resources.

**Cluster Manger:**

This component will control entire resource management and scheduling of cluster. When you submit a spark job , the driver component (spark Context) will regiter with the master node of the cluster which will ask cluster manager to allocate available resources asked by driver. So Cluster Manger only provides the executors for doing data processing in each worker nodes.

It find the worker nodes where the executors will be launched.

Spark comes with a default cluster manager called “Stand alone cluster manager”. But Spark can run on other cluster managers like YARN,MESOS etc. Most widely used is YARN in Hadoop cluster.

**Driver Program:**

This is nothing but sparkContext of your spark program. When you submit a spark job to cluster, the spark Context defined (whch is usually a line of code) inside the spark Code will run first and it is called the driver. This controls entire lie cycle of spark code. Killing the driver code implies, killing the entire execution. If you are writing the entire code for doing a spark submit, sparkContext is defined as first line of code inside the program. But if we are launching an interactive shell, sparkContext is automatically instantiated.

**Executor:**

Executor is nothing but a JVM container with required resources to execute the code inside each worker node.

**Task:**

Task is the smallest unit of work done by spark under each Executor.

Apache Spark has a well-defined layered architecture where all the spark components and layers are loosely coupled. This architecture is further integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions:

* *Resilient Distributed Dataset (RDD)*
* *Directed Acyclic Graph (DAG)*

**RDD:**

* ***Resilient:*** Fault tolerant and is capable of rebuilding data on failure
* ***Distributed:*** Distributed data among the multiple nodes in a cluster
* ***Dataset:*** Collection of partitioned data with values

Resilient Distributed Datasets. Resilient Distributed Datasets (**RDD**) is a fundamental data structure of **Spark**. It is an immutable distributed collection of objects. Each dataset in **RDD** is divided into logical partitions, which may be computed on different nodes of the cluster.

**Transformation and Actions:**

Two types of**Apache Spark** RDD operations are- Transformations and Actions. A **Transformation**is a function that produces new**RDD** from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed. When the action is triggered after the result, new RDD is not formed like transformation.

**Transformation:**

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

Applying transformation built an RDD lineage, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as RDD operator graph or RDD dependency graph. It is a logical execution plan i.e., it is Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().

After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).There are two types of transformation

**Narrow transformation** – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of map(), filter().

**Wide transformation** – In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. Wide transformations are the result of groupbyKey() and reducebyKey().

**Actions:**

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task.

count(),collect(),take(),top(),reduce(),fold()

**Spark Job Life Cycle**

When you submit a job on a spark cluster , first sparkContext will start running which is nothing but your Driver Program.Under sparkContext only , all other tranformation and actions takes place. As per requested by driver code only , resources will be allocated And output of every action is received by driver or JVM only. So its important that how you are submitting your job . There are two ways of submitting your job to a cluster

* Clinet Mode
* Cluster Mode

**Client Mode** is nothing but you will be submitting your job through edge Node or Gate Way node which is associated to your cluster.

Any, any Spark process that would ever work on your cluster or local machine is a JVM process. So what is a JVM

**JVM**

JVM is a engine that provides runtime environment to drive the Java Code or applications. It converts Java bytecode into machines language. JVM is a part of JRE(Java Run Environment). It stands for Java Virtual Machine

* In other programming languages, the compiler produces machine code for a particular system. However, Java compiler produces code for a Virtual Machine known as Java Virtual Machine.
* First, Java code is complied into bytecode. This bytecode gets interpreted on different machines
* Between host system and Java source, Bytecode is an intermediary language.
* JVM is responsible for allocating memory space.

The JVM memory consists of the following segments:

* Heap Memory, which is the storage for Java objects
* Non-Heap Memory, which is used by Java to store loaded classes and other meta-data
* JVM code itself, JVM internal structures, loaded profiler agent code and data, etc.

**Heap:**

The JVM has a *heap* that is the runtime data area from which memory for all class instances and arrays are allocated. It is created at the JVM start-up.

The heap size may be configured with the following VM options:

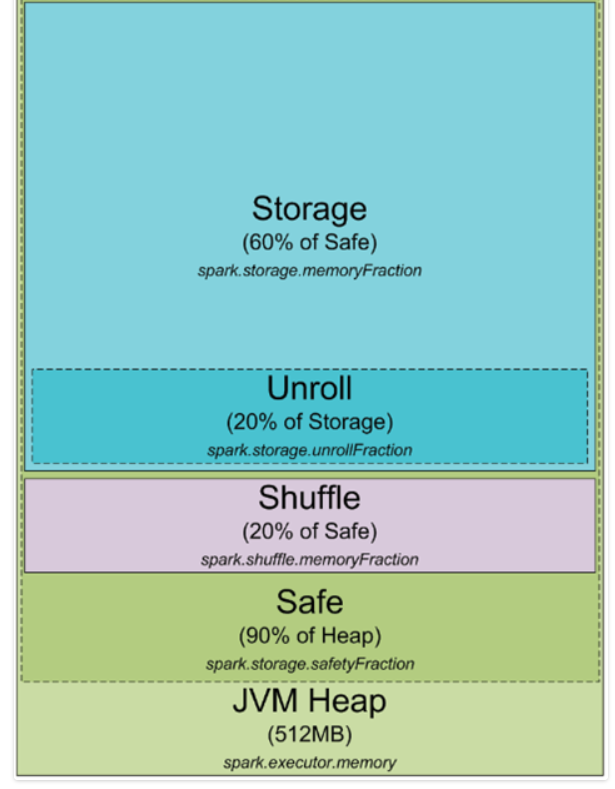
* -Xmx<size> - to set the maximum Java heap size
* -Xms<size> - to set the initial Java heap size

By default, the maximum heap size is 64 Mb.

Heap memory for objects is reclaimed by an automatic memory management system which is known as a garbage collector. The heap may be of a fixed size or may be expanded and shrunk, depending on the garbage collector's strategy.

**Memory management in spark(versions below 1.6)**

So for spark , a 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.safetyFractionparameter of Spark. Ok, as you might have heard of Spark as an in-memory tool, Spark allows you to store some data in memory.But this misconception is discussed and corrected in this blog:  <https://0x0fff.com/spark-misconceptions/>

* 1. First thing is that, any calculation that happens in any modern day computing is in-memory.Spark also doing the same thing, reads from some source cache it in memory ,process it and writes back to some target. So its utilizing the cache effectively.
  2. Second thing is that , Do you think that Spark processes all the transformations in memory? You would be disappointed, but the heart of Spark, “shuffle”, writes data to disks. If you have a “group by” statement in your SparkSQL query or you are just transforming RDD to PairRDD and calling on it some aggregation by key, you are forcing Spark to distribute data among the partitions based on the hash value of the key. The “shuffle” process consists of two phases, usually referred as “map” and “reduce”. “Map” just calculates hash values of your key (or other partitioning function if you set it manually) and outputs the data to **N** separate files on the local filesystem, where **N** is the number of partitions on the “reduce” side. “Reduce” side polls the “map” side for the data and merges it in new partitions. So if you have an RDD of **M** partitions and you transform it to pair RDD with **N** partitions, there would be **M\*N** files created on the local filesystems in your cluster, holding all the data of the specific RDD. There are some optimizations available to reduce amount of files. Also there are some work undergo to pre-sort them and then “merge” on “reduce” side, but this does not change the fact that each time you need to “shuffle” you data you are putting it to the HDDs.

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 in general Spark uses this memory for the exact task it is called after – for Shufflle

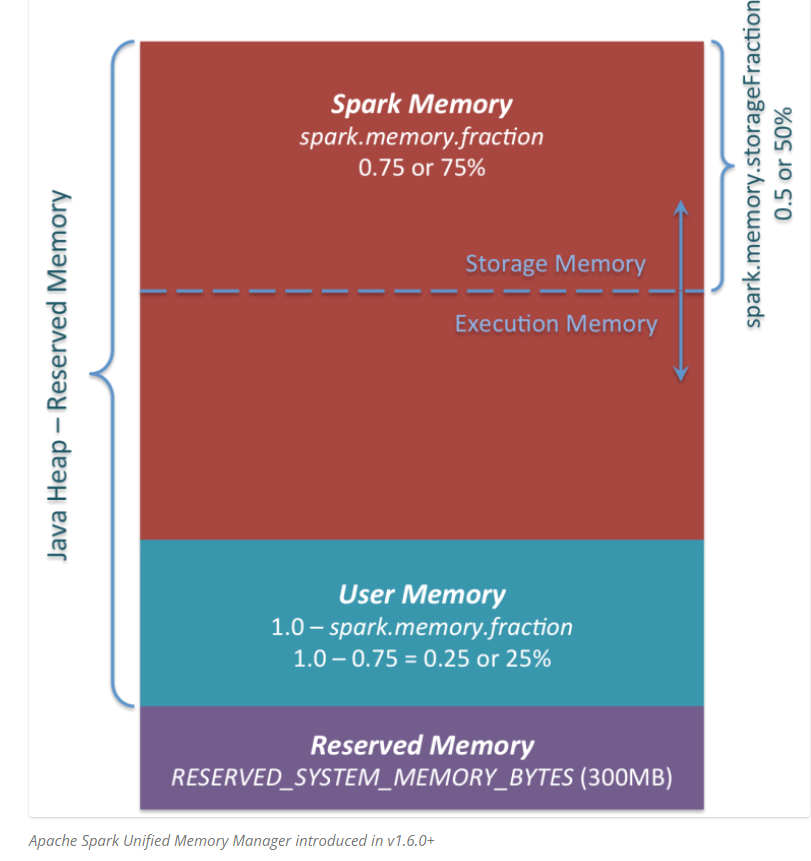
When the shuffle is performed, sometimes you as well need to sort the data. When you sort the data, you usually need a buffer to store the sorted data (remember, you cannot modify the data in the LRU cache in place as it is there to be reused later). So it needs some amount of RAM to store the sorted 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.

Now that’s all about memory management in spark. I would like to

**Memory management in spark(versions above 1.6)**

From spark 1.6.0+, we have unified memory manager. Diagram is given below



1. ***Reserved Memory***. This is the memory reserved by the system, and its size is hardcoded. As of Spark 1.6.0, its value is 300MB, which means that this 300MB of RAM does not participate in Spark memory region size calculations, and its size cannot be changed in any way without Spark recompilation or setting *spark.testing.reservedMemory*, which is not recommended as it is a testing parameter not intended to be used in production. Be aware, this memory is only called “reserved”, in fact it is not used by Spark in any way, but it sets the limit on what you can allocate for Spark usage. Even if you want to give all the Java Heap for Spark to cache your data, you won’t be able to do so as this “reserved” part would remain spare (not really spare, it would store lots of Spark internal objects). For your information, if you don’t give Spark executor at least *1.5 \* Reserved Memory = 450MB* heap, it will fail with “please use larger heap size” error message.
2. ***User Memory***. This is the memory pool that remains after the allocation of Spark Memory, and it is completely up to you to use it in a way you like. You can store your own data structures there that would be used in RDD transformations. For example, you can rewrite Spark aggregation by using mapPartitions transformation maintaining hash table for this aggregation to run, which would consume so called User Memory. In Spark 1.6.0 the size of this memory pool can be calculated as (“Java Heap” – “Reserved Memory”) \* (1.0 – spark.memory.fraction), which is by default equal to (“Java Heap” – 300MB) \* 0.25. For example, with 4GB heap you would have 949MB of User Memory. And again, this is the User Memory and its completely up to you what would be stored in this RAM and how, Spark makes completely no accounting on what you do there and whether you respect
3. ***Spark Memory***. Finally, this is the memory pool managed by Apache Spark. Its size can be calculated as (“Java Heap” – “Reserved Memory”) \* spark.memory.fraction, and with Spark 1.6.0 defaults it gives us (“Java Heap” – 300MB) \* 0.75. For example, with 4GB heap this pool would be 2847MB in size. This whole pool is split into 2 regions – Storage Memory and Execution Memory, and the boundary between them is set by spark.memory.storageFractionparameter, which defaults to 0.5. The advantage of this new memory management scheme is that this boundary is not static, and in case of memory pressure the boundary would be moved, i.e. one region would grow by borrowing space from another one. I would discuss the “moving” this boundary a bit later, now let’s focus on how this memory is being used:

* ***Storage Memory***. This pool is used for both storing Apache Spark cached data and for temporary space serialized data “unroll”. Also all the “broadcast” variables are stored there as cached blocks. In case you’re curious, here’s the code of [unroll](https://github.com/apache/spark/blob/branch-1.6/core/src/main/scala/org/apache/spark/storage/MemoryStore.scala#L249). As you may see, it does not require that enough memory for unrolled block to be available – in case there is not enough memory to fit the whole unrolled partition it would directly put it to the drive if desired persistence level allows this. As of “broadcast”, all the broadcast variables are stored in cache with *MEMORY\_AND\_DISK*persistence level.
* ***Execution Memory***. This pool is used for storing the objects required during the execution of Spark tasks. For example, it is used to store [shuffle intermediate buffer on the Map side](https://0x0fff.com/spark-architecture-shuffle/) in memory, also it is used to store hash table for hash aggregation step. This pool also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other threads (tasks).

Ok, so now let’s focus on the moving boundary between Storage Memory and Execution Memory. Due to nature of Execution Memory, you cannot forcefully evict blocks from this pool, because this is the data used in intermediate computations and the process requiring this memory would simply fail if the block it refers to won’t be found. But it is not so for the Storage Memory – it is just a cache of blocks stored in RAM, and if we evict the block from there we can just update the block metadata reflecting the fact this block was evicted to HDD (or simply removed), and trying to access this block Spark would read it from HDD (or recalculate in case your persistence level does not allow to spill on HDD).

So, we can forcefully evict the block from *Storage Memory*, but cannot do so from *Execution Memory*. When *Execution Memory* pool can borrow some space from *Storage Memory*? It happens when either:

* There is free space available in *Storage Memory* pool, i.e. cached blocks don’t use all the memory available there. Then it just reduces the *Storage Memory* pool size, increasing the *Execution Memory* pool.
* *Storage Memory* pool size exceeds the initial *Storage Memory* region size and it has all this space utilized. This situation causes forceful eviction of the blocks from *Storage Memory* pool, unless it reaches its initial size.

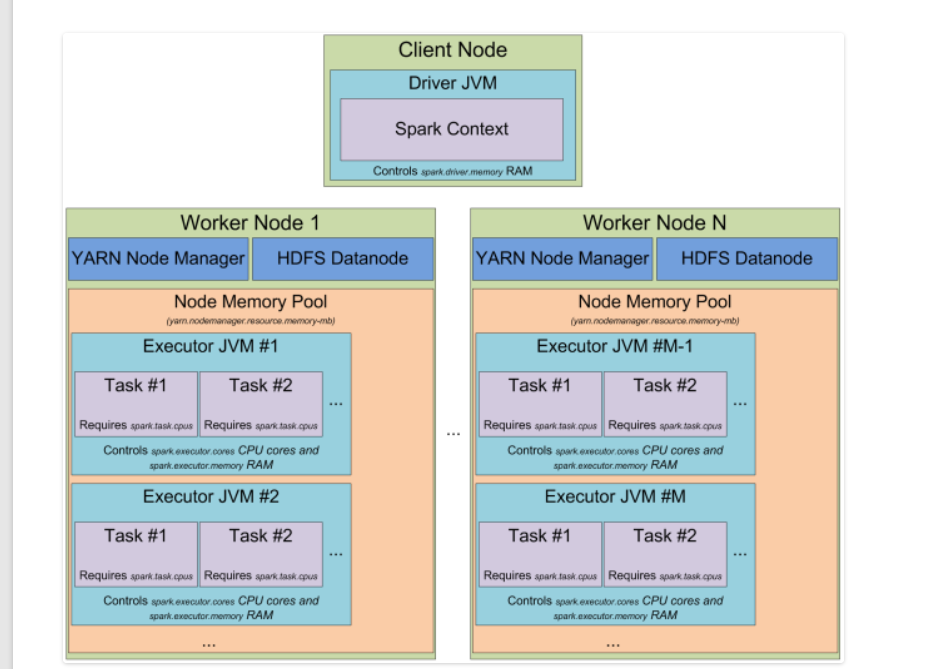
In turn, *Storage Memory* pool can borrow some space from *Execution Memory* pool only if there is some free space in *Execution Memory* pool available.

Initial *Storage Memory* region size, as you might remember, is calculated as “*Spark Memory” \* spark.memory.storageFraction =*(“*Java Heap*” – “*Reserved Memory*”) \* *spark.memory.fraction \* spark.memory.storageFraction*. With default values, this is equal to (“*Java Heap*” – 300MB) \* 0.75 \* 0.5 = (“*Java Heap*” – 300MB) \* 0.375. For 4GB heap this would result in 1423.5MB of RAM in initial *Storage Memory* region.

This implies that if we use Spark cache and the total amount of data cached on executor is at least the same as initial *Storage Memory* region size, we are guaranteed that storage region size would be at least as big as its initial size, because we won’t be able to evict the data from it making it smaller. However, if your *Execution Memory*region has grown beyond its initial size before you filled the *Storage Memory* region, you won’t be able to forcefully evict entries from *Execution Memory*, so you would end up with smaller *Storage Memory* region while execution holds its blocks in memory.

**Architecture of spark with YARN as cluster manager**

When you start a spark cluster with YARN as cluster manager, it looks like as below



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

**TASK**

Task 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.

**PARTITIONS**

Now let’s focus on another Spark abstraction called “***partition***”. All the data you work with in Spark is split into partitions. 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.

**SHUFFLE**

What is the shuffle in general? Imagine that you have a list of phone call detail records in a table and you want to calculate amount of calls happened each day. This way you would set the “day” as your key, and for each record (i.e. for each call) you would emit “1” as a value. After this you would sum up values for each key, which would be an answer to your question – total amount of records for each day. But when you store the data across the cluster, how can you sum up the values for the same key stored on different machines? The only way to do so is to make all the values for the same key be on the same machine, after this you would be able to sum them up.

There are many different tasks that require shuffling of the data across the cluster, for instance table join – to join two tables on the field “id”, you must be sure that all the data for the same values of “id” for both of the tables are stored in the same chunks. Imagine the tables with integer keys ranging from 1 to 1’000’000. By storing the data in same chunks I mean that for instance for both tables values of the key 1-100 are stored in a single partition/chunk, this way instead of going through the whole second table for each partition of the first one, we can join partition with partition directly, because we know that the key values 1-100 are stored only in these two partitions. To achieve this both tables should have the same number of partitions, this way their join would require much less computations. So now you can understand how important shuffling is.

Discussing this topic, I would follow the MapReduce naming convention. In the shuffle operation, the task that emits the data in the source executor is “mapper”, the task that consumes the data into the target executor is “reducer”, and what happens between them is “shuffle”.

Shuffling in general has 2 important compression parameters: **spark.shuffle.compress** – whether the engine would compress shuffle outputs or not, and **spark.shuffle.spill.compress** – whether to compress intermediate shuffle spill files or not. Both have the value “true” by default, and both would use **spark.io.compression.codec** codec for compressing the data, which is [snappy](https://en.wikipedia.org/wiki/Snappy_(software)) by default.