Spark study notes: core concepts visualized

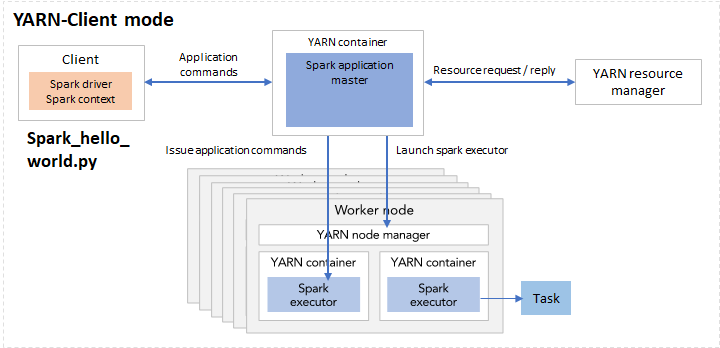
Learning Spark is not an easy thing for a person with less background knowledge on distributed systems. Even though I have been using Spark for quite some time, I find it time-consuming to get a comprehensive grasp of all the core concepts in Spark. The [official Spark documentation](https://spark.apache.org/docs/latest/) 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.



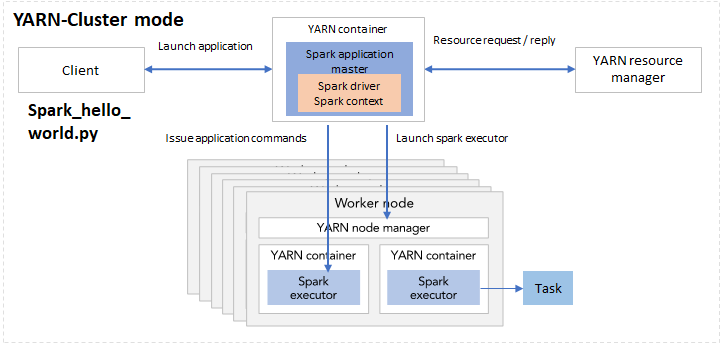


Fig 1. Spark deployment mode YARN-client (left) and YARN-cluster (right)

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.

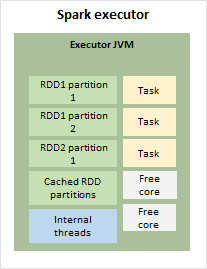


Fig 2. Spark executor internals

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.

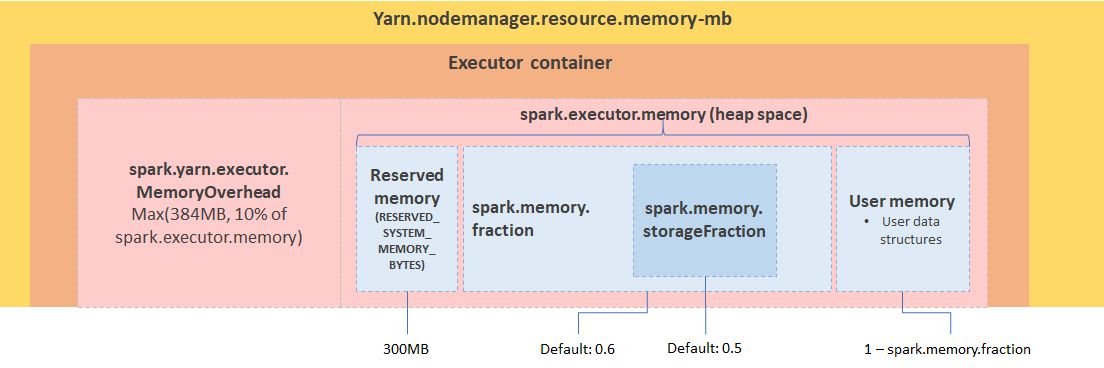


Fig 3. Spark executor memory decomposition

#### 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.

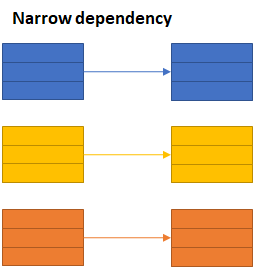


Fig 4–1. narrow dependency in RDD transformation

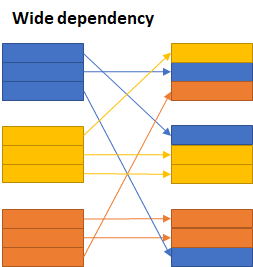


Fig 4–2. Wide dependency in RDD transformation

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.

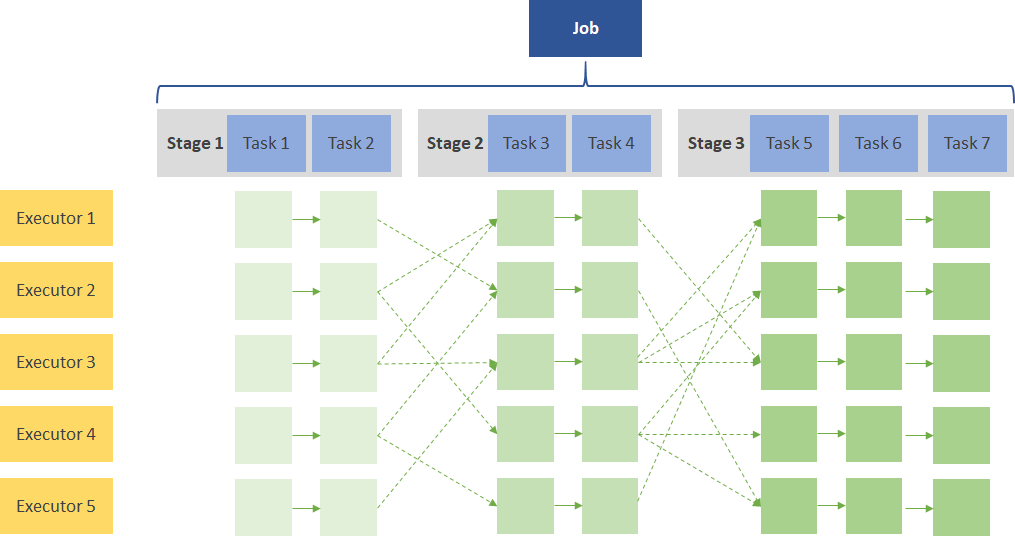


Fig 5. Illustration of one Spark job

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.

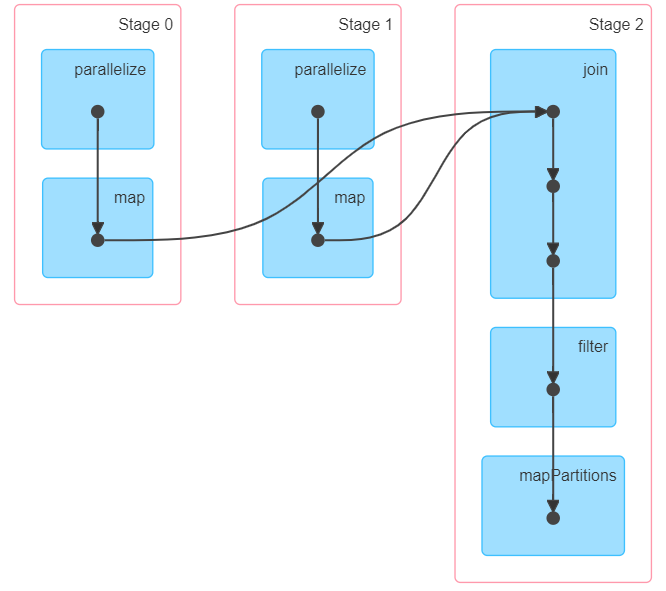


Fig 6. DAG visualization

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://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html>​
* <https://0x0fff.com/spark-memory-management/>​
* <https://www.pgs-soft.com/blog/spark-memory-management-part-1-push-it-to-the-limits/>​
* <https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations>​

<https://blog.usejournal.com/spark-study-notes-core-concepts-visualized-5256c44e4090>

## Distribution of Executors, Cores and Memory for a Spark Application running in Yarn:

spark-submit --class <CLASS\_NAME> --num-executors ? --executor-cores ? --executor-memory ? ....

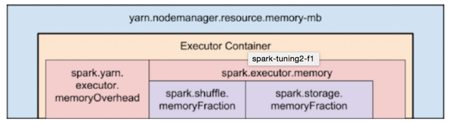
#### Ever wondered how to configure --num-executors, --executor-memory and --execuor-cores spark config params for your cluster?

## Let’s find out how..

1. **Lil bit theory:** Let’s see some key recommendations that will help understand it better
2. **Hands on:** Next, we’ll take an example cluster and come up with recommended numbers to these spark params

## Lil bit theory:

#### Following list captures some recommendations to keep in mind while configuring them:

* **Hadoop/Yarn/OS Deamons:** When we run spark application using a cluster manager like Yarn, there’ll be several daemons that’ll run in the background like NameNode, Secondary NameNode, DataNode, JobTracker and TaskTracker. So, while specifying num-executors, we need to make sure that we leave aside enough cores (~1 core per node) for these daemons to run smoothly.
* **Yarn ApplicationMaster (AM):** ApplicationMaster is responsible for negotiating resources from the ResourceManager and working with the NodeManagers to execute and monitor the containers and their resource consumption. If we are running spark on yarn, then we need to budget in the resources that AM would need (~1024MB and 1 Executor).
* **HDFS Throughput:** HDFS client has trouble with tons of concurrent threads. It was observed that HDFS achieves full write throughput with ~5 tasks per executor . So it’s good to keep the number of cores per executor below that number.
* **MemoryOverhead:** Following picture depicts spark-yarn-memory-usage.   
  

Two things to make note of from this picture:

Full memory requested to yarn per executor =

spark-executor-memory + spark.yarn.executor.memoryOverhead.

spark.yarn.executor.memoryOverhead =

Max(384MB, 7% of spark.executor-memory)

So, if we request 20GB per executor, AM will actually get 20GB + memoryOverhead = 20 + 7% of 20GB = ~23GB memory for us.

* Running executors with too much memory often results in excessive garbage collection delays.
* Running tiny executors (with a single core and just enough memory needed to run a single task, for example) throws away the benefits that come from running multiple tasks in a single JVM.

## Enough theory.. Let’s go hands-on..

Now, let’s consider a 10 node cluster with following config and analyse different possibilities of executors-core-memory distribution:

**\*\*Cluster Config:\*\***

10 Nodes

16 cores per Node

64GB RAM per Node

### First Approach: Tiny executors [One Executor per core]:

Tiny executors essentially means one executor per core. Following table depicts the values of our spar-config params with this approach:

- `--num-executors` = `In this approach, we'll assign one executor per core`

= `total-cores-in-cluster`

= `num-cores-per-node \* total-nodes-in-cluster`

= 16 x 10 = 160

- `--executor-cores` = 1 (one executor per core)

- `--executor-memory` = `amount of memory per executor`

= `mem-per-node/num-executors-per-node`

= 64GB/16 = 4GB

**Analysis:** With only one executor per core, as we discussed above, we’ll not be able to take advantage of running multiple tasks in the same JVM. Also, shared/cached variables like broadcast variables and accumulators will be replicated in each core of the nodes which is **16 times**. Also, we are not leaving enough memory overhead for Hadoop/Yarn daemon processes and we are not counting in ApplicationManager. **NOT GOOD!**

### Second Approach: Fat executors (One Executor per node):

Fat executors essentially means one executor per node. Following table depicts the values of our spark-config params with this approach:

- `--num-executors` = `In this approach, we'll assign one executor per node`

= `total-nodes-in-cluster`

= 10

- `--executor-cores` = `one executor per node means all the cores of the node are assigned to one executor`

= `total-cores-in-a-node`

= 16

- `--executor-memory` = `amount of memory per executor`

= `mem-per-node/num-executors-per-node`

= 64GB/1 = 64GB

**Analysis:** With all 16 cores per executor, apart from ApplicationManager and daemon processes are not counted for, HDFS throughput will hurt and it’ll result in excessive garbage results. Also,**NOT GOOD!**

### Third Approach: Balance between Fat (vs) Tiny

**According to the recommendations which we discussed above:**

* Based on the recommendations mentioned above, Let’s assign 5 core per executors => --executor-cores = 5 (for good HDFS throughput)
* Leave 1 core per node for Hadoop/Yarn daemons => Num cores available per node = 16-1 = 15
* So, Total available of cores in cluster = 15 x 10 = 150
* Number of available executors = (total cores/num-cores-per-executor) = 150/5 = 30
* Leaving 1 executor for ApplicationManager => --num-executors = 29
* Number of executors per node = 30/10 = 3
* Memory per executor = 64GB/3 = 21GB
* Counting off heap overhead = 7% of 21GB = 3GB. So, actual --executor-memory = 21 - 3 = 18GB

**So, recommended config is: 29 executors, 18GB memory each and 5 cores each!!**

**Analysis:** It is obvious as to how this third approach has found right balance between Fat vs Tiny approaches. Needless to say, it achieved parallelism of a fat executor and best throughputs of a tiny executor!!

### Conclusion:

We’ve seen:

* Couple of recommendations to keep in mind which configuring these params for a spark-application like:
  + Budget in the resources that Yarn’s Application Manager would need
  + How we should spare some cores for Hadoop/Yarn/OS deamon processes
  + Learnt about spark-yarn-memory-usage
* Also, checked out and analysed three different approaches to configure these params:
  + Tiny Executors - One Executor per Core
  + Fat Executors - One executor per Node
  + Recommended approach - Right balance between Tiny (Vs) Fat **coupled** with the recommendations.

--num-executors, --executor-cores and --executor-memory.. these three params play a very important role in spark performance as they control the amount of CPU & memory your spark application gets. This makes it very crucial for users to understand the right way to configure them. Hope this blog helped you in getting that perspective…

<https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html>

Spark Memory Management

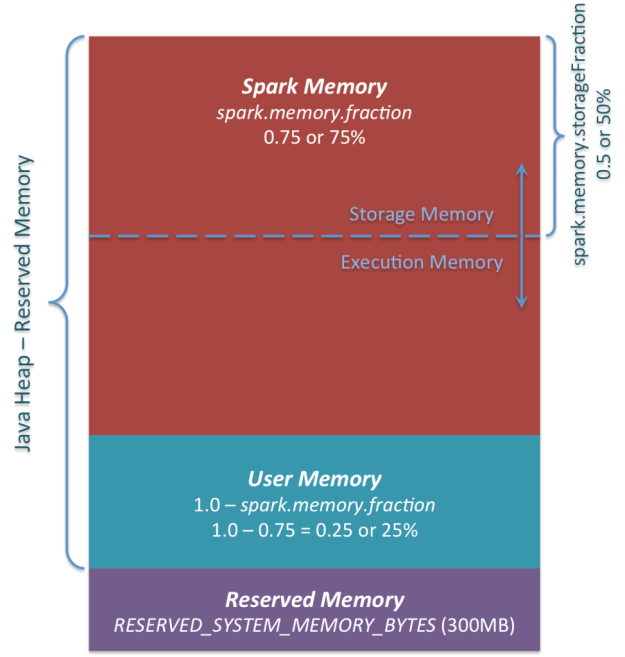
[57 Replies](https://0x0fff.com/spark-memory-management/#comments)

Starting Apache Spark version 1.6.0, memory management model has changed. The old memory management model is implemented by [StaticMemoryManager](https://github.com/apache/spark/blob/branch-1.6/core/src/main/scala/org/apache/spark/memory/StaticMemoryManager.scala" \t "_blank) class, and now it is called “legacy”. “Legacy” mode is disabled by default, which means that running the same code on Spark 1.5.x and 1.6.0 would result in different behavior, be careful with that. For compatibility, you can enable the “legacy” model with *spark.memory.useLegacyMode* parameter, which is turned off by default.

Previously I have described the “legacy” model of memory management in this [article about Spark Architecture](https://0x0fff.com/spark-architecture/) almost one year ago. Also I have written an article on [Spark Shuffle implementations](https://0x0fff.com/spark-architecture-shuffle/) that briefly touches memory management topic as well.

This article describes new memory management model used in Apache Spark starting version 1.6.0, which is implemented as [UnifiedMemoryManager](https://github.com/apache/spark/blob/branch-1.6/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala" \t "_blank).

Long story short, new memory management model looks like this:

[](https://i1.wp.com/0x0fff.com/wp-content/uploads/2016/01/Spark-Memory-Management-1.6.0.png)

*Apache Spark Unified Memory Manager introduced in v1.6.0+*

You can see 3 main memory regions on the diagram:

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 this boundary or not. Not respecting this boundary in your code might cause OOM error.
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.storageFraction*parameter, 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:
   1. ***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.
   2. ***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.

I hope this article helped you better understand Apache Spark memory management principles and design your applications accordingly. If you have any questions, feel free to ask them in comments.

<https://0x0fff.com/spark-memory-management/>

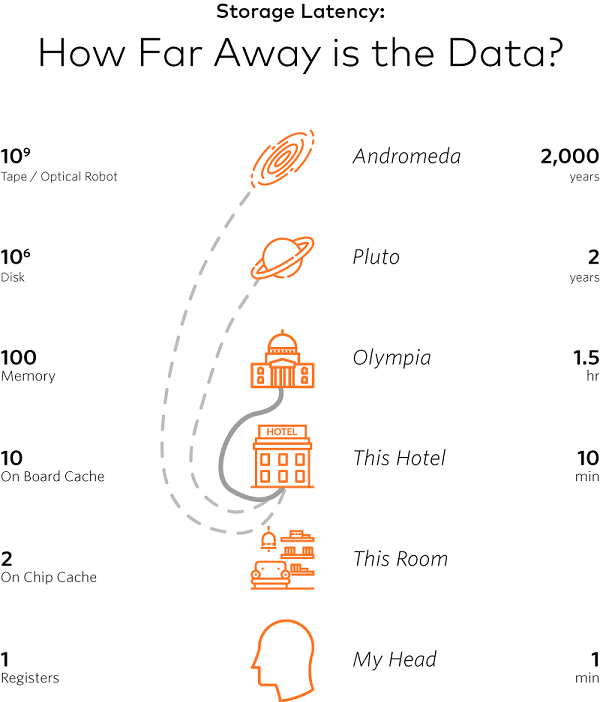
One of the bottlenecks of Hadoop Map-Reduce applications is that data needs to be stored somewhere persistently between each stage – which wastes a lot of time on performing I/O operations.

One of Spark’s selling points is that it takes 10-100x less time to finish a similar job written as Hadoop Map-Reduce. The trick is to store data reliably in in-memory – this makes repeatedly accessing it (ie. for iterative algorithms) incomparably faster.

Efficient memory use is critical for good performance, but the reverse is also true – inefficient memory use leads to bad performance.

In every Big Data application, memory is an extremely indispensable resource. Before proceeding further, it will be good to gain some insight regarding the possible impact of choosing the proper storage for our data.

[Jim Gray](https://www.wikiwand.com/en/Jim_Gray_(computer_scientist)) has an interesting way of explaining this:

[](https://www.pgs-soft.com/app/uploads/2017/06/storage.png)

He compares how our brain accesses information (using neurological pathways) to how a processor reads data from its registers, (it does so almost instantly). Similarly, we can compare reading data from a disk to a two-year trip to Pluto.

(Nowadays, we would probably take into consideration modern SSD drives, but changing an order of magnitude would not make a big difference – we would only get to a closer planet.)

The point is that the cost of **accessing data on a disk is a few orders of magnitude greater** – we need to process as many things in the memory as possible.

However, nothing is free and works perfectly out of the box.

In order to have optimised Spark jobs, developers are required to spend some time understanding how memory is managed and how to make proper adjustments.

There are three main contentions on how to arbitrate it:

* between execution and storage
* across tasks, running in parallel
* across operators, running within the same task

Before analysing each case, let us consider the **executor**.

## Executor memory overview

**An executor** is the Spark application’s JVM process launched on a **worker node**. It runs tasks in threads and is responsible for keeping relevant partitions of data. Each process has an allocated **heap** with available memory (executor/driver).

**Example:** With default configurations (spark.executor.memory=1GB, spark.memory.fraction=0.6), an executor will have about 350 MB allocated for execution and storage regions (unified storage region). The other 40% is reserved for storing various meta-data, user data structures, safeguarding against OOM errors, etc. Also, take note that there is a dedicated hard-coded portion of so-called reserved memory (300 MB \* 1.5), which is used for storing internal Spark objects. Quite often, the exact calculations are not entirely intuitive – for in-depth examples take a look at [this](https://stackoverflow.com/questions/43801062/how-does-web-ui-calculate-storage-memory-in-executors-tab) and [that](https://0x0fff.com/spark-memory-management/)topic.

Spark tasks operate in two main memory regions:

* **execution** – used for shuffles, joins, sorts, and aggregations
* **storage** – used to cache partitions of data

Execution memory tends to be more “short-lived” than storage. It is evicted immediately after each operation, making space for the next ones.

In terms of storage, two main functions handle the persistence of data –RDD’s cache() and persist().

cache() is an alias for persist(StorageLevel.MEMORY\_ONLY)

As you will later see on RDD, partitions can exist in the memory or on the disk – across the cluster, at any given point in time; (you can see this in the Storage tab in the Spark UI).

Hence, using cache()might be dangerous for data sets larger than the clusters’ memory. Each RDD partition might be evicted and consequently rebuilt (which is expensive). A better option to consider in such cases might be to use persist() with a suitable option.

## Off-heap

Even though the best performance is obtained when operating solely in **on-heap** memory, Spark also makes it possible to use **off-heap** storage for certain operations.

**Off-heap** refers to objects (serialised to byte array) that are managed by the operating system but stored outside the process heap in native memory (therefore, they are not processed by the garbage collector). Accessing this data is slightly slower than accessing the on-heap storage but still faster than reading/writing from a disk. The downside is that the user has to manually deal with managing the allocated memory.

A common problem with bigger memory configurations is that the application tends to freeze due to GC scans (sometimes referred to as “GC storms“). The main benefit of activating off-heap memory is that we can mitigate this issue by using native system memory (which is not supervised by JVM).

Off-heap memory usage is available for execution and storage regions (since Apache Spark 1.6 and 2.0, respectively).

## Properties

**Essentials:**

* spark.executor.memory – specifies the executor’s process memory heap (default 1 GB)
* spark.driver.memory – specifies the driver’s process memory heap (default 1 GB)
* spark.memory.fraction – a fraction of the heap space (minus 300 MB \* 1.5) reserved for execution and storage regions (default 0.6)

**Off-heap:**

* spark.memory.offHeap.enabled – the option to use off-heap memory for certain operations (default false)
* spark.memory.offHeap.size – the total amount of memory in bytes for off-heap allocation. It has no impact on heap memory usage, so make sure not to exceed your executor’s total limits (default 0)

## To be continued…

In the next part of this article, we will look through three memory contentions Spark needs to deal with and how Spark SQL can come to the rescue.