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

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

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

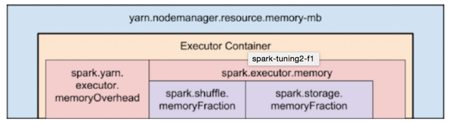
#### Ever wondered how to configure --num-executors, --executor-memory and --execuor-cores spark config params for your cluster?

## 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…

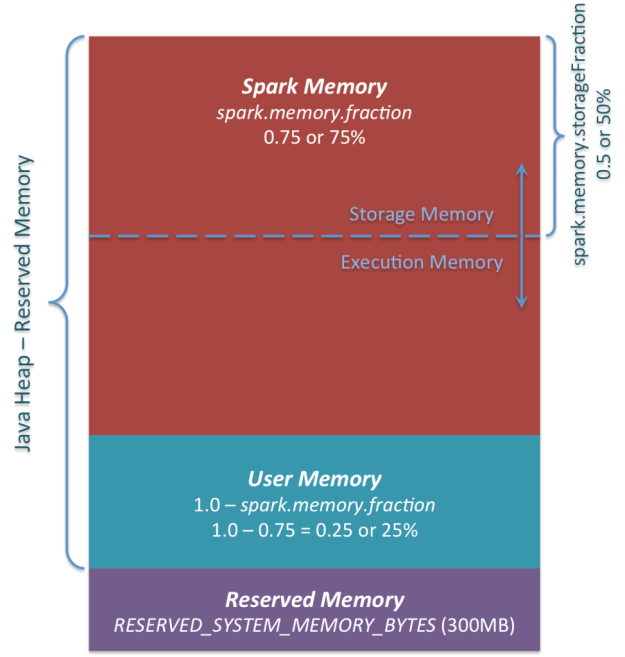
# Spark Memory Management

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.

## Task Memory Management

Tasks are the basically the threads that run within the Executor JVM of a Worker node to do the needed computation. It is the smallest unit of execution that operates on a partition in our dataset. Given that Spark is an in-memory processing engine where all of the computation that a task does happens in-memory, its important to understand Task Memory Management...

To understand this topic better, we’ll section Task Memory Management into 3 parts:

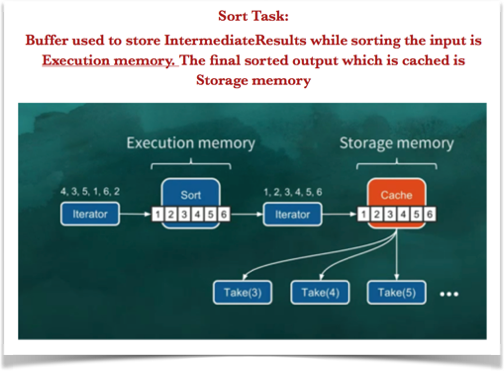
1. **What are the memory needs of a task?**
2. **Memory Management within a Task -** How does spark arbitrate memory within a task?
3. **Memory Management across the Tasks -** How is memory shared among different tasks running on the same worker node?

### 1. What are the memory needs of a task?

Every task needs 2 kinds of memory:

1. **Execution Memory:**
   * Execution Memory is the memory used to buffer Intermediate results.
   * As soon as we are done with the operation, we can go ahead and release it. Its short lived.
   * For example, a task performing Sort operation, would need some sort of collection to store the Intermediate sorted values.
2. **Storage Memory:**
   * Storage memory is more about reusing the data for future computation.
   * This is where we store cached data and its long-lived.
   * Until the allotted storage gets filled, Storage memory stays in place.
   * LRU eviction is used to spill the storage data when it gets filled.

**Following picture illustrates it with an example task of “Sorting a collection of Int’s”**

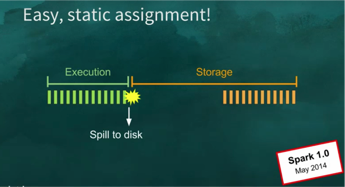
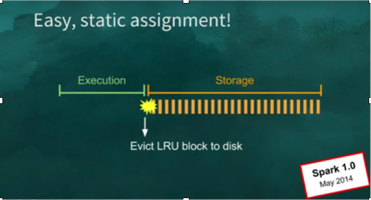


Now that we’ve seen the memory needs of a task, Let’s understand how Spark manages it..

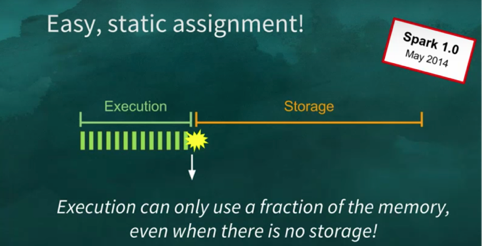
### 2. Memory Management within a Task

**How does Spark arbitrate between ExecutionMemory(EM) and StorageMemory(SM) within a Task?**

Simplest Solution – **Static Assignment**

* Static Assignment - This approach basically splits the total available on-heap memory (size of your JVM) into 2 parts, one for ExecutionMemory and the other for StorageMemory.
* As the name says, this memory split is static and doesn’t change dynamically.
* This has been the solution since spark 1.0.
* While running our task, if the execution memory gets filled, it’ll get spilled to disk as shown below: 
* Likewise, if the Storage memory gets filled, its evicted via LRU (Least recently Used) 

**Disadvantage:** Because of the hard split of memory between Execution and Storage, even if the task doesn’t need any StorageMemory, ExecutionMemory will still be using only its chunk of the total available free memory..

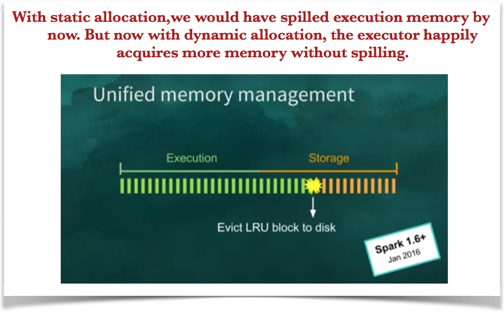


**How to fix this?**

UNIFIED MEMORY MANAGEMENT - This is how Unified Memory Management works:

* Express execution and storage memory as one single unified region.
* So, there’s no splitting of memory in this approach.
* Execution and Storage share it combinedly with this agreement: Keep acquiring execution memory and evict storage as u need more execution memory.

Following picture depicts Unified memory management..



**But, why to evict storage than execution memory?**

Spilled execution data is always going to be read back from disk where as cached data may or may not be read back. (User might tend to aggressively cache data at times with/without its need.. )

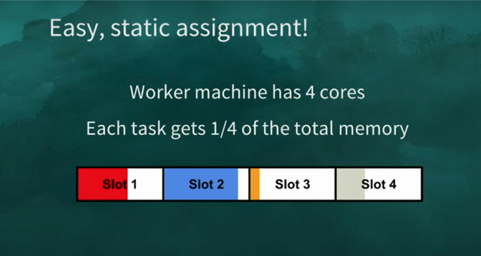
**What if application relies on caching like a Machine Learning application?**

We can’t just blow away cached data like that in this case. So, for this usecase, spark allows user to specify minimal unevictable amount of storage a.k.a cache data. Notice this is not a reservation meaning, we don’t pre-allocate a chunk of storage for cache data such that execution cannot borrow from it. Rather, only when there’s cached data this value comes into effect..

### 3.Memory Management across the Tasks

**How is memory shared among different tasks running on the same worker node?**

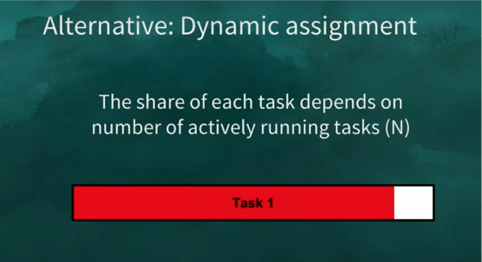
Ans: **Static Assignment (again!!)** - No matter how many tasks are currently running, if the worker machine has 4 cores, we’ll have 4 fixed slots.



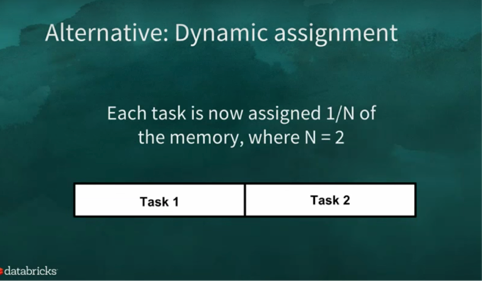
**Drawback:** Even if there’s only 1 task running, its going to get only one-quarter of the total memory.

### Better Solution – Dynamic Assignment (Again)!!

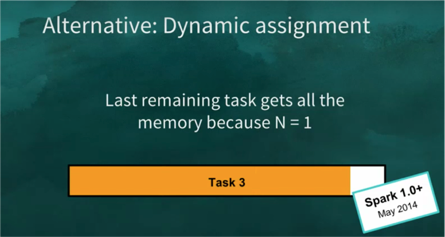
More efficient alternative is Dynamic allocation where how much memory a task gets is dependent on total number of tasks running. If there is only one task running, it can feel free to acquire all the available memory.



As soon as another task comes in, task1 will have to spill to disk and free space for task2 for fairness. So, number of slots are determined dynamically depending on active running tasks.



**Key Advantage:** One notable behaviour here is - What happens to a straggler which is a last remaining task. These straggler tasks are potentially expensive because everybody is already done but then this is the last remaining task. This model allocates all the memory to the straggler because number of actively running tasks is one. This has been there since spark 1.0 and its been working fine since then. So, Spark haven’t found a reason to change it.



## CONCLUSION - TASK MEMORY MANAGEMENT

We understood:

* Two kinds of memory needs per task
* How to arbitrate within a task (i.e., between execution and storage memory of a single task)
* How to arbitrate memory between multiple tasks
* **Common Solution:** Instead of statically reserving memory, force memory to spill when there’s memory contention. So, essentially, solve memory contention lazily rather than eagerly.
* Static assignment is simpler
* Dynamic allocation handles stragglers better

[My HomePage](https://spoddutur.github.io/spark-notes/)

## References:

* [Deep dive Apache Spark Memory Mangement](https://spark-summit.org/2016/events/deep-dive-apache-spark-memory-management/)
* [Spark Memory Management](https://www.youtube.com/watch?v=dPHrykZL8Cg)