Distributed Systems Architecture

brought to you by Alexey Grishchenko

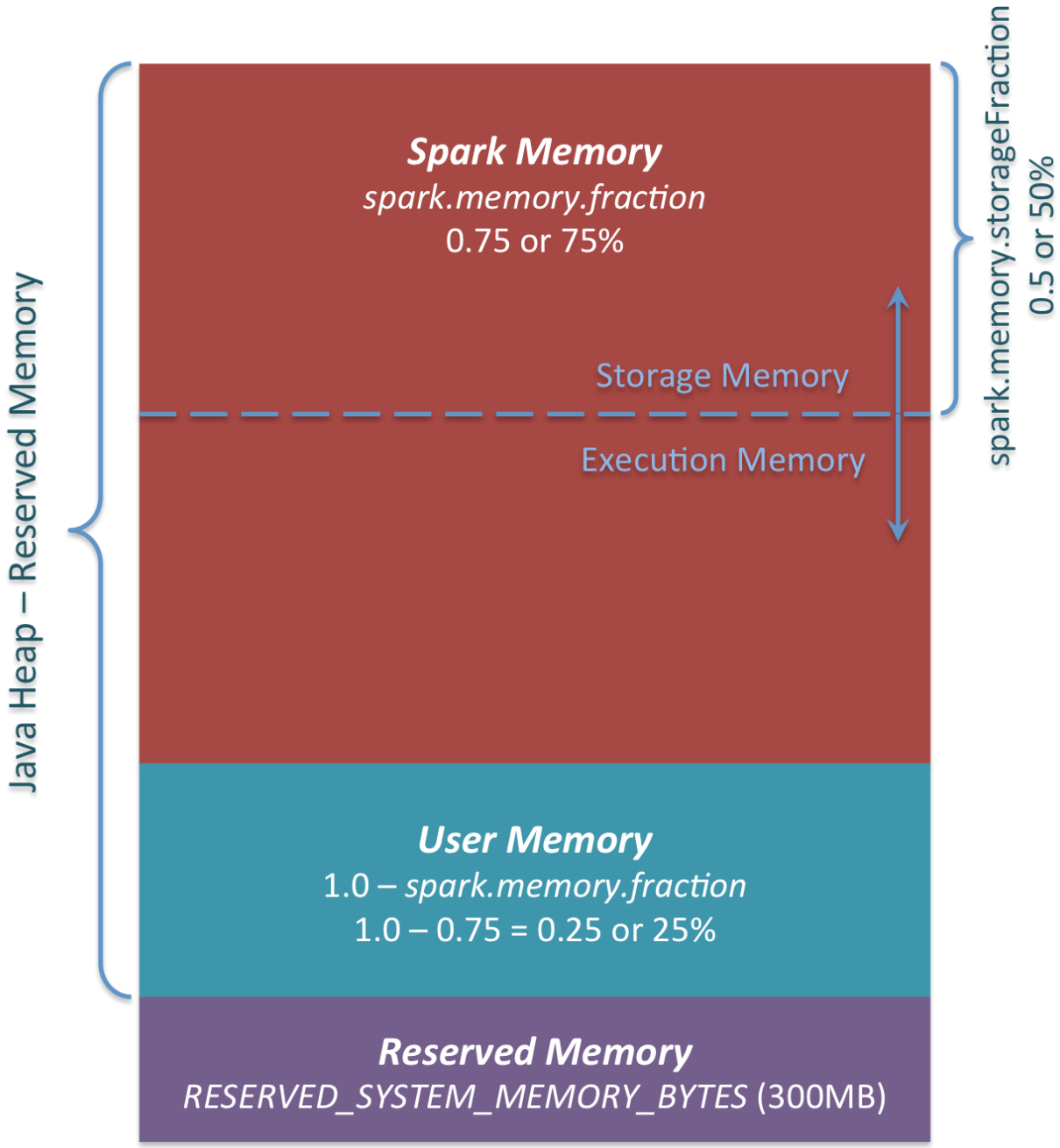
Spark Memory Management

Starting Apache Spark version 1.6.0, memory management model has changed. The old memory management model is implemented by StaticMemoryManager 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 almost one year ago. Also I have written an article on Spark Shuffle implementations 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.

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



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. 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 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.

This entry was posted in Hadoop, Spark and tagged apache spark, memory management, Spark, spark 1.6.0, spark architecture on January 28, 2016 [https://0x0fff.com/spark-memory-management/] .

27 thoughts on "Spark Memory Management"

Pingback: Spark Memory Management | Filling the gaps in Big Data

Alejandro
January 31, 2016 at 7:51 pm

Thanks for the very detailed article! It will be very helpful when using Spark. As a user, how do I know the current state of how memory is being used?

For example, if I want to cache an RDD and I calculated that memory would be enough when allocating resources, how would I verify that my assumptions are still correct after the work I've done in the context?

Thanks!

0x0FFF Post author
February 1, 2016 at 8:40 am

Thank you
As of RDD cache status, you can always check this in the Spark WebUI, on the “Storage” tab. It will show you RDD storage level, percentage of RDD cached, size of the cache in memory, size of the cache on disk
If you want to get it from your code, you have to use the method SparkContext.getRDDStorageInfo() – this would return you the same information (basically array of RDDInfo http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.storage.RDDInfo). But this is developer API, so the signature of the function might be changed in the future releases

nellaivijay
March 19, 2016 at 11:00 pm

Thanks Nice explanation

MANDAR VAIDYA
March 22, 2016 at 10:44 am

really nice explanation.

When i am executing spark job after every task GC(Garbage collector) is calling and job is taking more time for execution.Is their any spark configuration which can avoid this scenario.

Regards,
Mandar Vaidya.

0x0FFF Post author
March 22, 2016 at 2:04 pm

There is no Java setting to prevent garbage collection. But you should be able to analyze heap dump of a running process to see which structures are causing big garbage collection. Most likely the root cause of this is your code that creates objects for each row, for example. This can be fixed by fixing your code