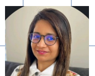
Understanding Spark Memory Management

Introduction

Apache Spark is a powerful, open-source distributed computing system that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. One of the critical aspects of achieving optimal performance in Spark applications is effective memory management. This section delves into the key components and concepts involved in Spark memory management.

Contents



1. Overview of Spark Memory Management

Memory management in Spark is essential for ensuring efficient resource utilization, minimizing garbage collection overhead, and balancing the needs of various Spark components like caching, shuffling, and computation.

Importance of Memory Management

- Resource Utilization: Efficient memory management ensures that the available resources are utilized optimally, preventing scenarios where certain tasks are starved of memory while others are over-provisioned.
- Performance Optimization: Proper memory allocation and management can significantly reduce garbage collection overhead, leading to better application performance.
- Reliability: By managing memory effectively, Spark applications can avoid common issues such as out-of-memory errors, which can cause job failures and reduce overall system reliability.

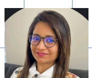
In the below diagram the important thing is major operation is run on On-Heap Memory which is by the JVM.

What is JVM?

- JAVA VIRTUAL MACHINE it is the virtual computer that is use for running a Java programs. So spark is written in Scala. Which runs on the JVM. but JVM server as an execution environment not just for Java but also for other programming languages including Scala.

- So when we write the code in Python using Pyspark. That time we are using the wrapper around the Java API's of spark.

- This is essentially done to be able to use all of the features of spark. Because the underlying operation will still going happen on JVM. And that is the reason why On-Heap memory is managed by the JVM.



2. Memory Components in Spark

On-Heap Memory is divided into the 4 sections

1. Execution memory: it is the place where all of the joins, shuffles, sorting and group By take place.

2. Storage memory : it is the place where caching take place and this caching can be on RDD or Dataframes and it also store the broadcast variables.

   Unified memory : **Execution memory + Storage memory**

3. User memory : it is use to store the user object for example : variables , list, sets, dictionary that we define and even for storing UDF(USER DEFINED FUNCTIONS)

4. Reserved memory : this spark needs for running itself. And for storing internal object.

Overhead memory is used for some internal system level operation.

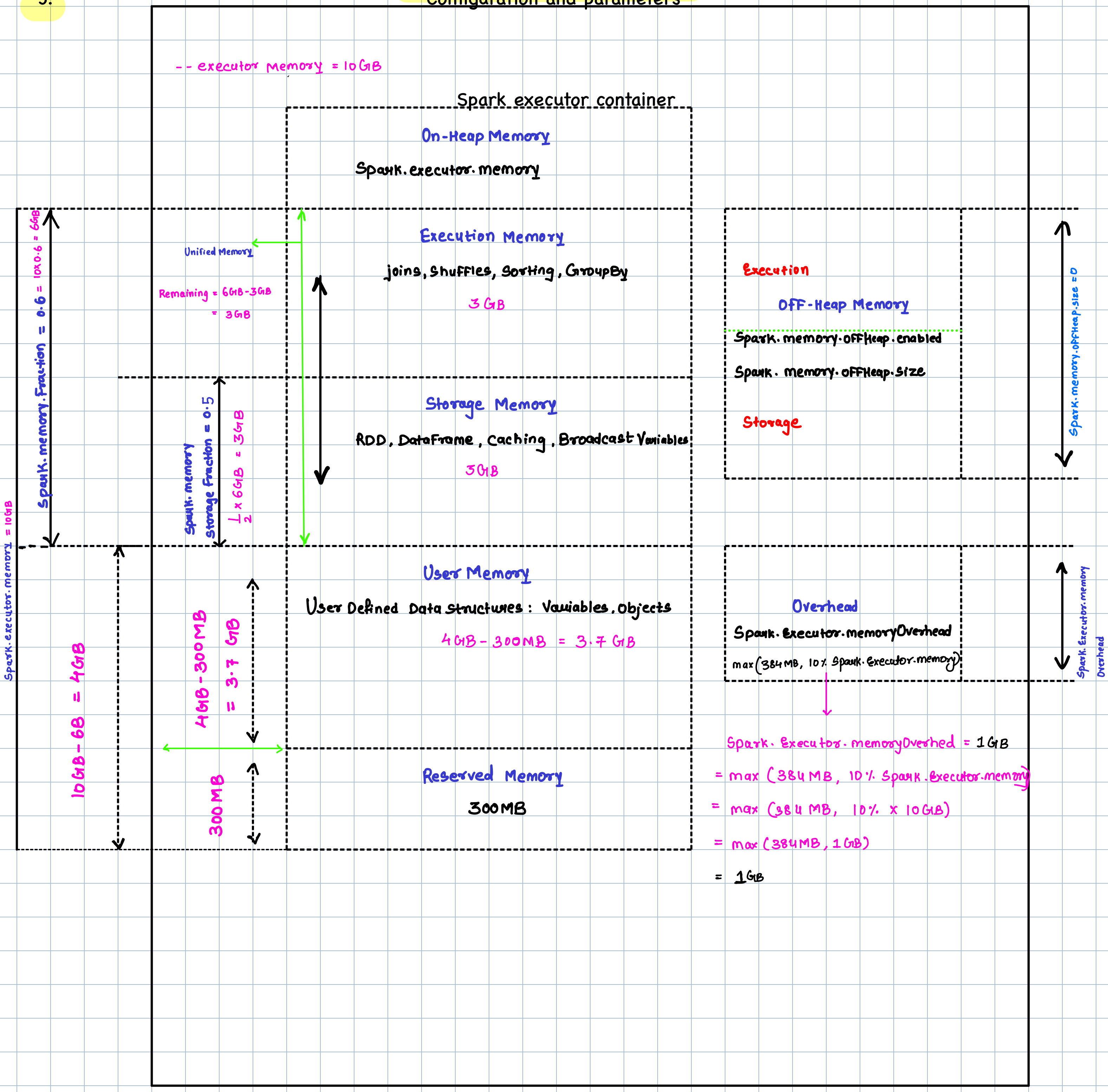
Off-Heap memory is used to reduce the garbage collection

.

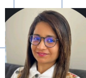
Follow me on
https://www.linkedin.com/in/dhwanimehta05/

3.

Configuration and parameters


-- executor memory = 10GB
Spark executor container
On-Heap Memory
Spark.executor.memory
Execution Memory
joins, shuffles, sorting, GroupBy
3 GB
Storage Memory
RDD, DataFrame, caching, Broadcast Variables
3GB
User Memory
User Defined Data Structures : Variables, objects
4GB - 300MB = 3.7 GB
Reserved Memory
300MB
Unified Memory
Remaining = 6GB-3GB
= 3GB
Spark.memory.Fraction = 0.6 = 10x0.6 = 6GB
Spark.memory.Storage Fraction = 0.5
1/2 x 6GB = 3GB
Spark.executor.memory = 10GB
10GB - 6B = 4GB
4GB - 300MB
= 3.7 GB
300 MB
Execution
OFF-Heap Memory
Spark.memory.OFFHeap.enabled
Spark.memory.OFFHeap.size
Storage
Spark.memory.offHeap.size = 0
Overhead
Spark.Executor.memoryOverhead
max(384MB, 10% Spark.Executor.memory)
Spark.Executor.memory Overhead
Spark.Executor.memoryOverhead = 1GB
= max (384 MB, 10% Spark.Executor.memory)
= max (384 MB, 10% x 10GB)
= max (384MB, 1GB)
= 1GB

Follow me on
https://www.linkedin.com/in/dhwanimehta05/



Calculation Configuration and calculations

When we run our spark Job we specify : On-Heap Memory Calculation

- - executor memory : **10 GB**

Spark.memory.fraction = **10GB * 0.6 = 6GB**

Spark.memory.storagefraction = **6GB * 0.5 = 3GB. (STORAGE MEMORY)**

Spark.memory.executionmemory = **6GB - 3GB = 3GB (EXECUTION MEMORY)**

Overhead memory (max 384 or 10% of spark.executor.memory) = **MAX (384 MB OR 10% * 10GB) =** 1GB

Off-Heap-Memory Calculation

Spark.memory.offHeap.size = 0
But we can set this to non-zero number and allocate memory but the way
off-heap memory is structured this also has 2 parts like execution and storage so it looks the same as the unified memory. Normally it is disabled but we can enable it by setting the parameter shown in the diagram.

In starting we can provide like 10% to 20% of the execution memory
Spark.memeory.offHeapenabled = true
Spark.memory.offHeap.size = 10GB * 10% = 1GB

Questions:

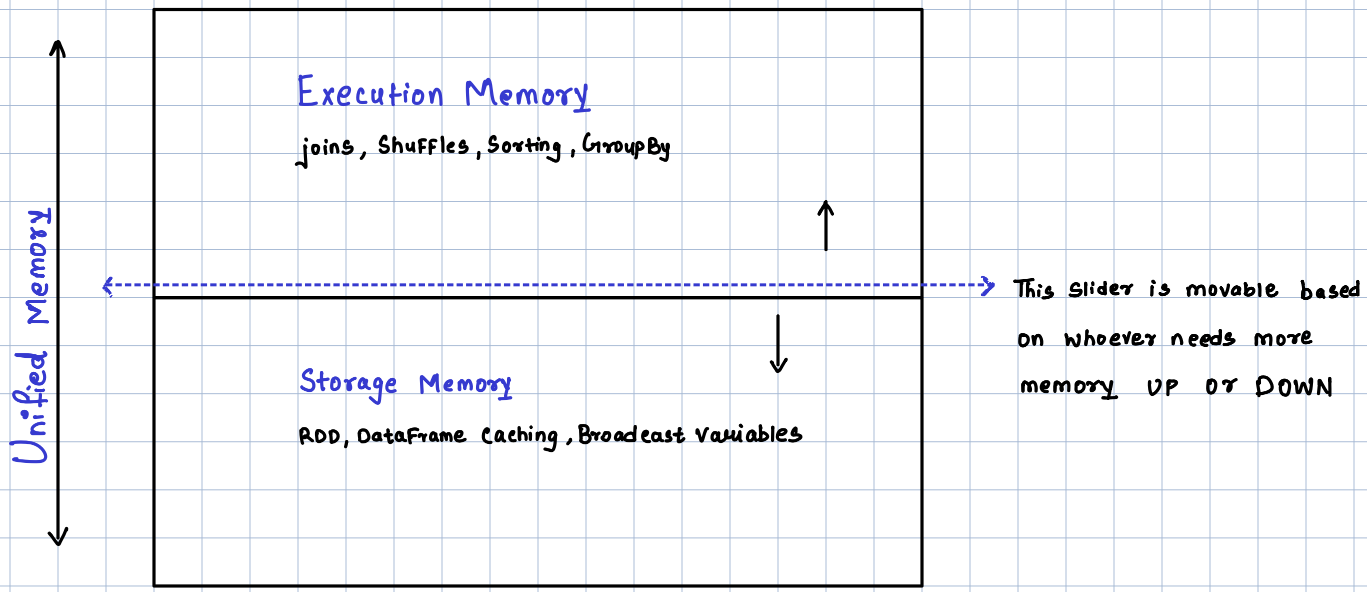
If 10GB of executor memory is allocated only to the On-Heap memory then what memory will be allocated to Off-Heap Memory?
So important thing to note here is executor memory is allocated to On-Heap memory but when spark request to the YARN cluster manager for the memory then spark will add on the executor memory + overhead memory + Off-Heap Memory (if enabled) to requested YARN.

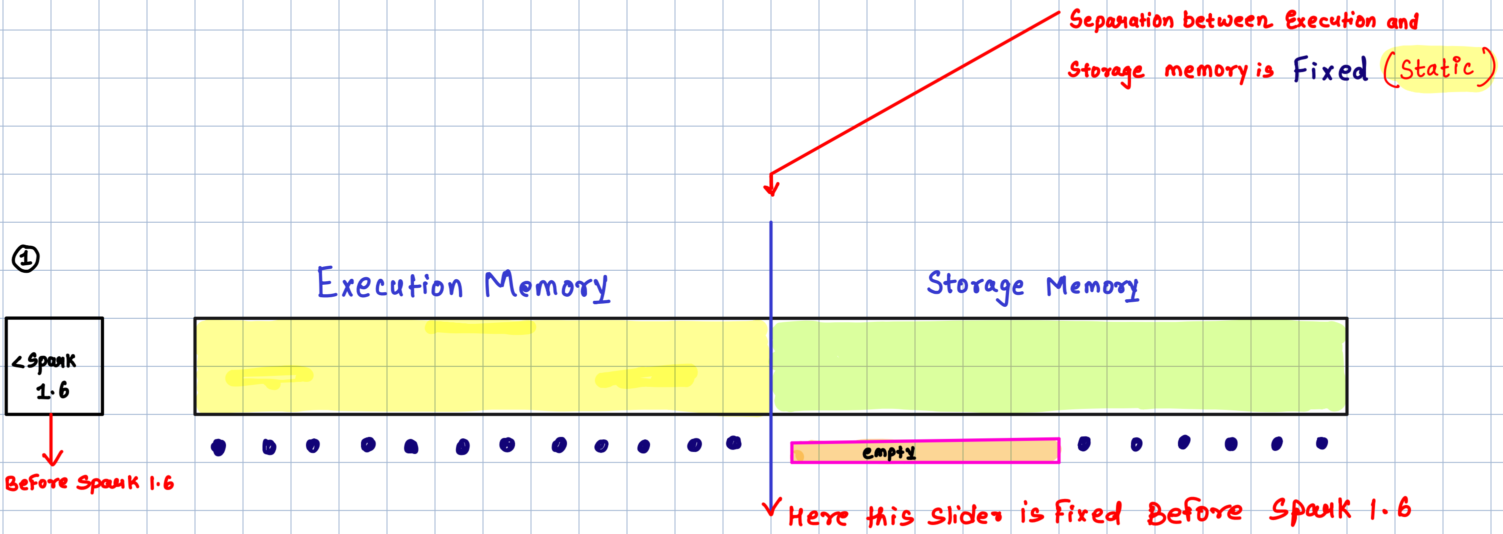
In the above case let's assume that Off-Heap Memory is disabled then YARN cluster manager will take total of executor memory + overhead memory = 10GB + 1GB = **11GB** will be requested for the container.

4. Memory Management Techniques

understand unified memory in detail:

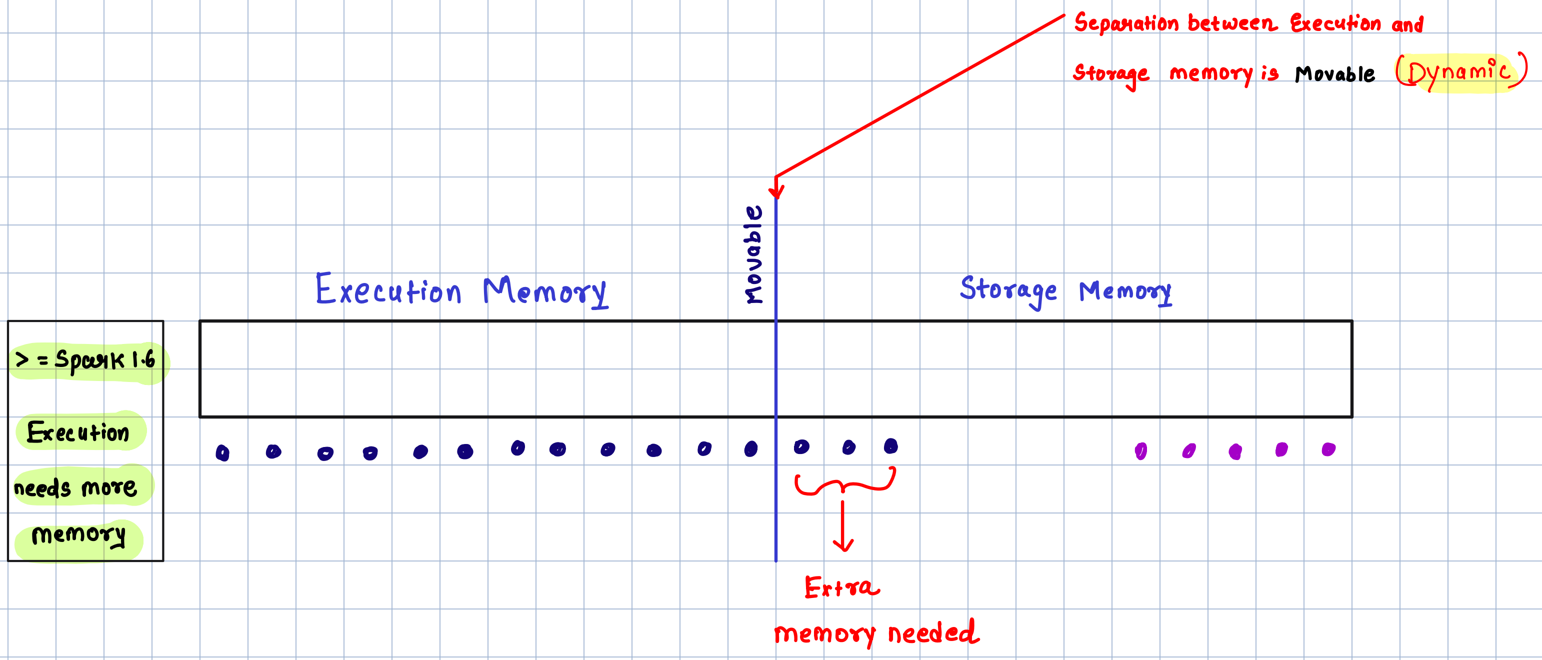
Together of execution memory and storage memory is unified memory because of the dynamic memory management strategy of the spark. Which means if Execution needs more memory then it will simply use some of the storage memory. And if storage wanted more memory then it will simply go ahead and use Execution memory. But there is a priority which is given to the Execution memory because all of the joins, shuffle , sorting and group by all this happens here.


Unified Memory
Execution Memory
joins, Shuffles, Sorting, GroupBy
Storage Memory
RDD, DataFrame caching, Broadcast variables
This slider is movable based on whoever needs more memory UP or DOWN


Separation between Execution and Storage memory is Fixed (Static)
①
Execution Memory
Storage Memory
< Spark 1.6
Before Spark 1.6
empty
Here this slider is Fixed Before Spark 1.6

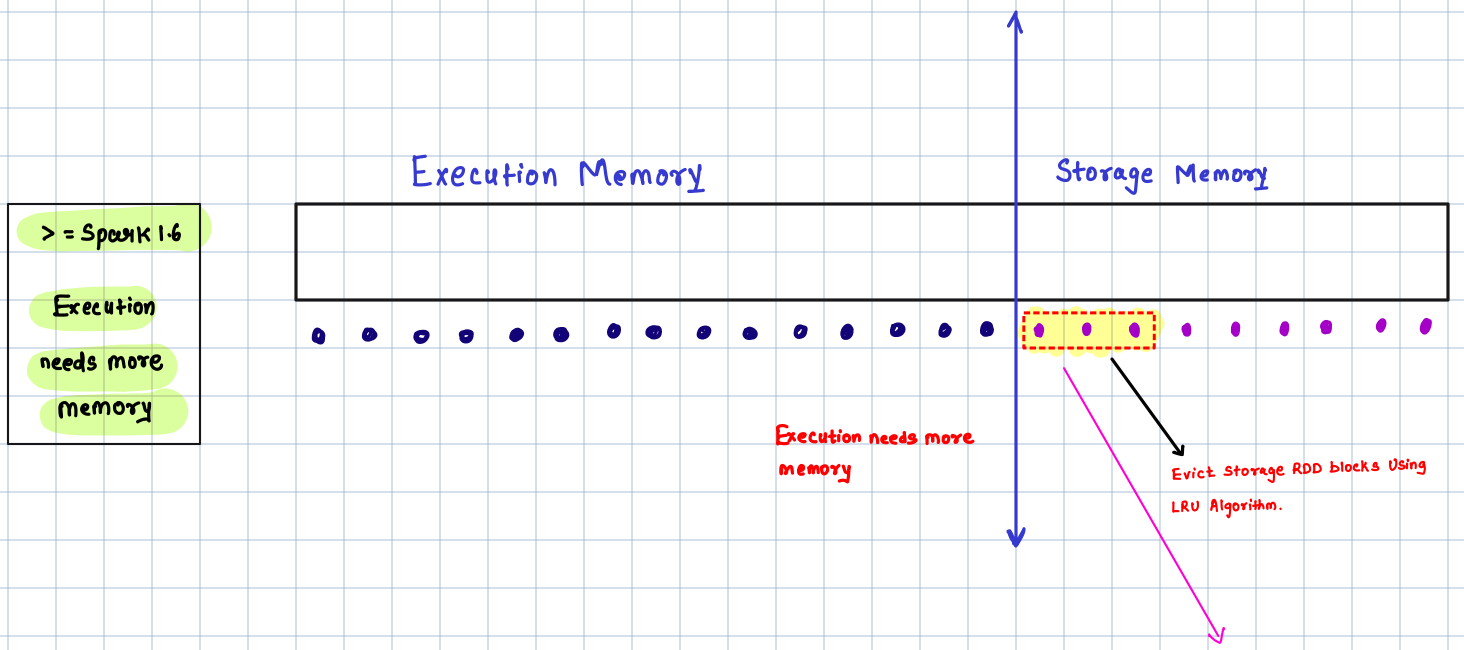
So, if Execution Memory is full and if it need more memory despite Storage being empty It won't be able to use the empty part of the storage Memory.
So, this is the big waste of Memory available.

Follow me on
https://www.linkedin.com/in/dhwanimehta05/


Separation between Execution and Storage memory is Movable (Dynamic)
Movable
Execution Memory
Storage Memory
> = Spark 1.6
Execution needs more memory
Extra memory needed

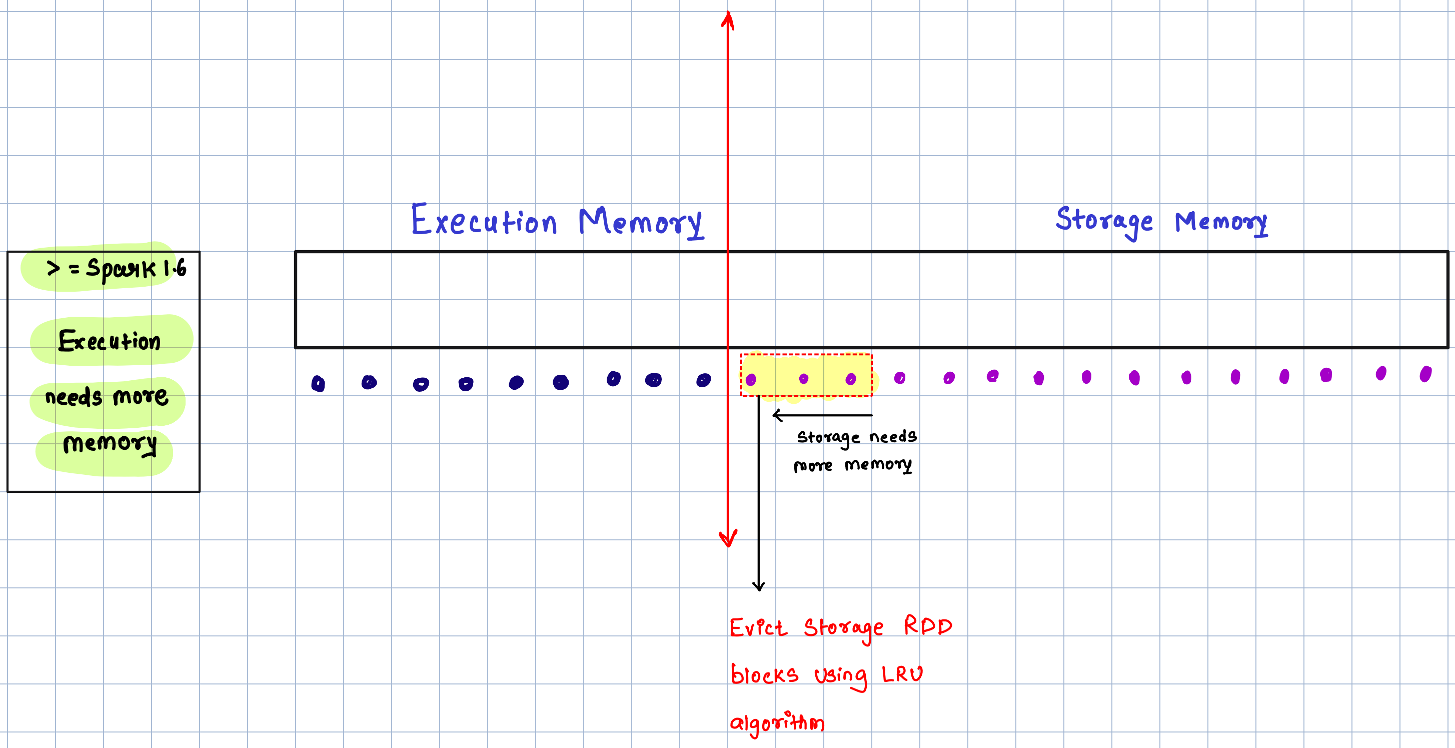
Rules of the slider

Let's say execution needs more memory and there is a vacant memory and storage is not using that then in that case we scan simply go and use that portion of storage memory.


Execution Memory
Storage Memory
> = Spark 1.6
Execution needs more memory
Execution needs more memory
Evict Storage RDD blocks Using LRU Algorithm.

In this case as the execution needs more memory execution has occupied some part of the storage memory for more joins or shuffles or aggregation etc so in this case storage will going to evict some of his block and give it to execution memory and this eviction will take place using LRU algorithm which LEAST RECENTLY USED BLOCKS will be evicted. So this space now will be going to available for execution memory.

Follow me on
https://www.linkedin.com/in/dhwanimehta05/


Execution Memory
Storage Memory
> = Spark 1.6
Execution needs more memory
Storage needs more memory
Evict Storage RDD blocks using LRU algorithm

In this case storage has occupied lot of memory for caching , broadcasting purpose and now Execution memory also needs memory and **execution memory** has **priority** because all of the aggregation , joins and shuffle operation takes place in the execution memory. so in that case it will tell storage memory to evict the part of its memory in this case storage will evict memories based on LRU (least recently used block).

So now we know that majority of the spark operation takes place in On-Heap Memory all types of joins, aggregation, shuffles and sorting and this On-Heap Memory is managed by the JVM

Follow me on

https://www.linkedin.com/in/dhwanimehta05/

5. GARBAGE COLLECTION

Now in the event of On-Heap Memory is full there is going to be a garbage collection cycle. And in garbage collection it will pause the current operation of the program and then it will clean up all of the unwanted object shows in order to make room for the program to resume.

So this pauses which garbage collection done from time to time may take a tall on the performance of the program and it is in this case we will make use of Off-Heap Memory.

And Off-Heap Memory is managed by the operation system and therefore it is not subject to the garbage collection cycle on the executor. So that is major reason spark developers has take into consideration the allocation and de-allocation of the memory this makes things little complicated but this will only going to avoid the memory spills. So that's why Off-Heap-Memory should be used with extra caution.

Off-Heap Memory Is memory is slower then On-Heap Memory. Where as On-Heap Memory is closest to spark. However if spark has 2 option such as to spill to disk or use the Off-Heap Memory then better choice would be to use the Off-Heap Memory.

6. Common Memory Management Issues

Out of Memory Errors

Out of memory (OOM) errors occur when Spark tasks require more memory than what is available. This can happen due to improper configuration, large data sets, or inefficient code.

Memory Leaks

Memory leaks occur when objects are not properly released, leading to gradual memory consumption over time. Identifying and fixing memory leaks is crucial for long-running Spark applications.

Strategies to Mitigate Issues

- Use data serialization formats that reduce memory footprint.
- Optimize Spark job configurations.
- Regularly monitor and profile Spark applications to detect memory issues.

7. Best Practices for Spark Memory Management

- Monitor and Profile: Use Spark's built-in tools and external monitoring systems to track memory usage and identify bottlenecks.
- Tuning Parameters: Adjust Spark and JVM parameters based on workload characteristics and performance metrics.
- Efficient Coding: Write efficient Spark code by avoiding wide transformations and using actions judiciously.
- Data Serialization: Choose appropriate data serialization formats (like Kryo) to reduce memory overhead.
- Cache Wisely: Cache only the necessary data and use the appropriate storage levels to balance memory and performance.

LIKE

&

FOLLOW

for more

Follow me on

https://www.linkedin.com/in/dhwanimehta05/