Iqbal Singh
29 Followers · About Follow

Apache Spark Executor Memory Architecture

Iqbal Singh May 17 · 4 min read

Apache Spark framework is a distributed processing framework for big data. It is based on the Map Reduce Algorithm for processing the big data workloads by shipping the processing logic to the data.

Spark reads all the data in the memory of an executor and performs the in-memory transformations, resulting in 10x faster processing compared to the old Hadoop map-reduce processing engine.

However, loading and processing the data in-memory for the bigger workloads is a pain and the famous Java OOM exception is common in memory-intensive spark jobs. Understanding Spark memory architecture is very important for optimizing the workloads in production.

Memory Architecture

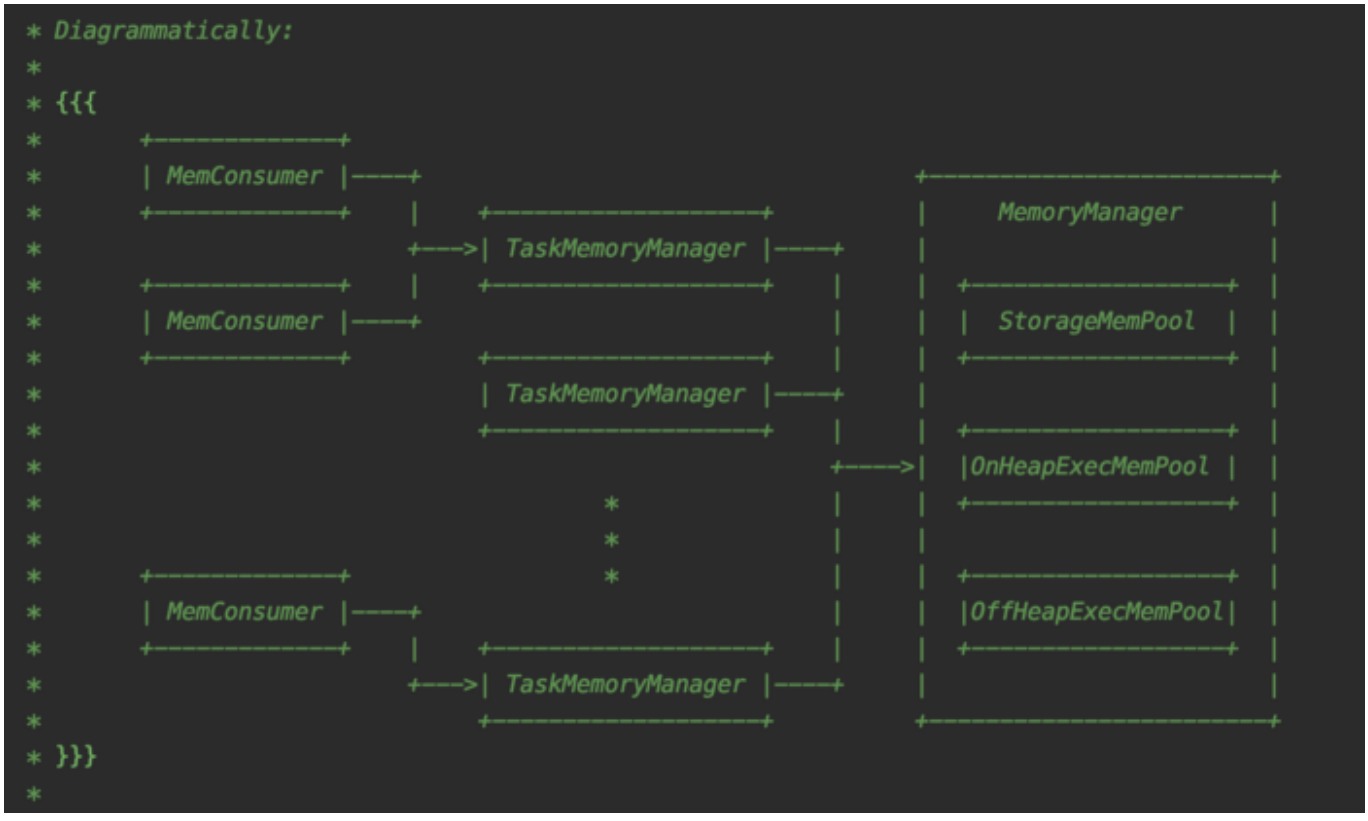
Spark memory architecture consists of below components

- **Memory Manager**: *It manages Spark's overall memory usage within a JVM. This component implements the policies for dividing the available memory across tasks and for allocating memory between storage (memory used caching and data transfer) and execution (memory used by computations, such as shuffles, joins, sorts, and aggregations).*

- **Task Memory Manager**: *It manages the memory allocated by individual tasks. Tasks interact with TaskMemoryManager and never directly interact with the JVM-wide*

- **Memory Consumer**: *These are clients of the TaskMemoryManager and correspond to individual operators and data structures within a task. The TaskMemoryManager receives memory allocation requests from MemoryConsumers and issues callbacks to consumers in order to trigger spilling when running low on memory.*

- **Memory Pool:** *Pools are a bookkeeping abstraction used by the Memory Manager to track the division of memory between storage and execution.*



Spark Memory Package Document Screenshot.

Unified Memory Manager

Spark 1.6+ used the new `UnifiedMemoryManager` for the memory architecture, However, it came with the backwards compatibility for the old memory manager `StaticMemoryManager`.

There are two implementations of `org.apache.spark.memeory.MemoryManager` which vary in how they handle the sizing of their memory pools:

- `org.apache.spark.memory.UnifiedMemoryManager` , the default in **Spark 1.6+**, enforces soft boundaries between storage and execution memory, allowing requests for memory in one region to be fulfilled by borrowing memory from the other.

- `org.apache.spark.memory.StaticMemoryManager` enforces hard boundaries between storage and execution memory by statically partitioning Spark's memory and preventing storage and execution from borrowing memory from each other. This mode is retained only for legacy compatibility purposes.

We will be discussing the `UnifiedMemoryManager` in this article, ***In Spark 3.0, StaticMemoryManager support is removed.***

A MemoryManager that enforces a soft boundary between execution and storage such that either side can borrow memory from the other.

Execution memory *refers to that used for computation in shuffles, joins, sorts and aggregations.*

Storage memory *refers to that used for caching and propagating internal data across the cluster. There exists one MemoryManager per JVM.*

The region shared between execution and storage is a fraction of (the total heap space — 300MB) configurable through `spark.memory.fraction` (0.6).

The position of the boundary within this space is further determined by `spark.memory.storageFraction(` 0.5). This means the size of the storage region is 0.6 *

0.5 = 0.3 of the heap space by default.

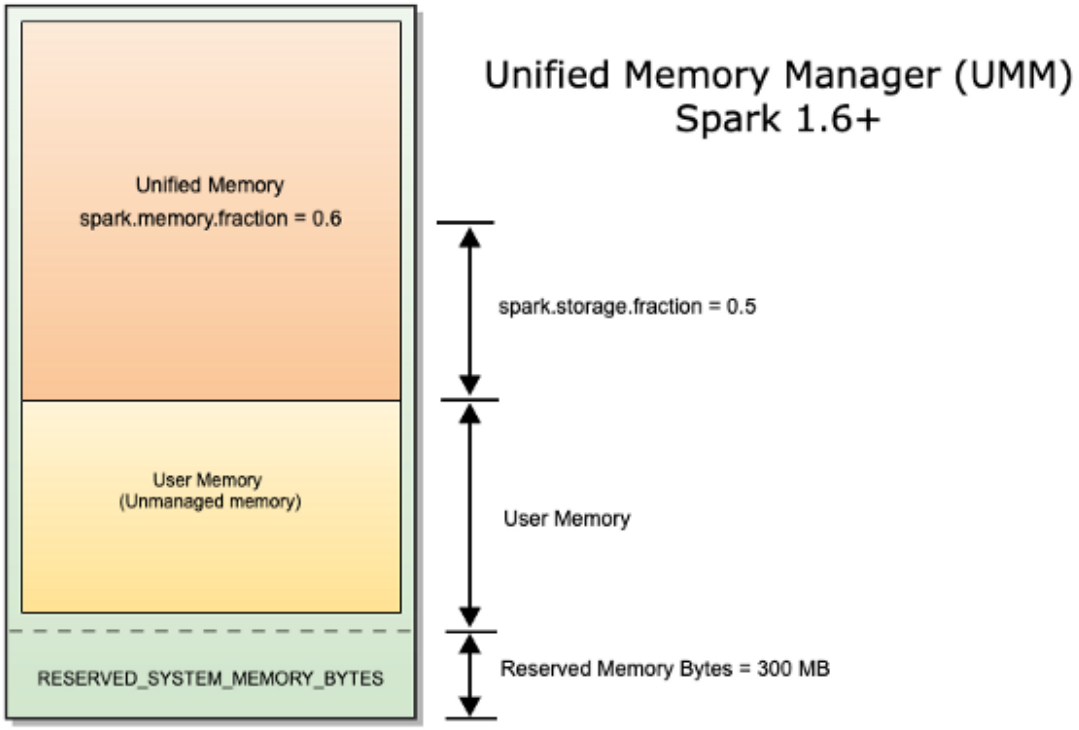
Storage can borrow as much execution memory as is free until execution reclaims its space. When this happens, cached blocks will be evicted from memory until sufficient borrowed memory is released to satisfy the execution memory request. Similarly, execution can borrow as much storage memory as is free.

However, execution memory is never evicted by storage due to the complexities involved in implementing this. The implication is that attempts to cache blocks may fail if execution has already eaten up most of the storage space, in which case the new blocks will be evicted immediately according to their respective storage levels.

```
/**
 * Grow the execution pool by evicting cached blocks, thereby
shrinking the storage pool.
 * When acquiring memory for a task, the execution pool may need to
make multiple
 * attempts. Each attempt must be able to evict storage in case
another task jumps in
 * and caches a large block between the attempts. This is called once
per attempt.
 */

def maybeGrowExecutionPool(extraMemoryNeeded: Long): Unit = {
  if (extraMemoryNeeded > 0) {

// There is not enough free memory in the execution pool, so try to
reclaim memory from storage. We can reclaim any free memory from the
storage pool. If the storage pool has grown to become larger than
`storageRegionSize`, we can evict blocks and reclaim the memory that
storage has borrowed from execution.

    val memoryReclaimableFromStorage = math.max(
      storagePool.memoryFree,
      storagePool.poolSize - storageRegionSize)

    if (memoryReclaimableFromStorage > 0) {
      // Only reclaim as much space as is necessary and available:
      val spaceToReclaim = storagePool.freeSpaceToShrinkPool(
        math.min(extraMemoryNeeded, memoryReclaimableFromStorage))
      storagePool.decrementPoolSize(spaceToReclaim)
      executionPool.incrementPoolSize(spaceToReclaim)
    }
  }
}
```

Spark Executor JVM Structure:

- Reserved Memory Bytes: By default, 300 MB is allocated to run the container by the spark. The value is hardcoded in the code using a constant and cannot be updated.
- 60% is assigned to both storage and execution memory by default.
- (Container size -RESERVER_BYTES)-0.6 is assigned for the user memory for metadata and handling OOM errors.



Apache Spark Executor memory Architecture

Spark Memory Managers implemented the separation of the JVM heap into two parts of storage and execution. Ols Static Memory Manager provided a lot of flexibility to the engineers for tuning the memory usage of a job and the transition to the current Unified Memory Manager changed the design to provide an abstraction for complex memory tunning to the JVM by dynamically changing the partition boundary between the storage and execution pools during run time.

Spark Apache Big Data