# [Memory Tuning](http://spark.apache.org/docs/latest/tuning.html" \l "memory-tuning)

## Memory Tuning

There are three considerations in tuning memory usage: the amount of memory used by your objects (you may want your entire dataset to fit in memory), the cost of accessing those objects, and the overhead of garbage collection (if you have high turnover in terms of objects).

By default, Java objects are fast to access, but can easily consume a factor of 2-5x more space than the “raw” data inside their fields. This is due to several reasons:

* Each distinct Java object has an “object header”, which is about 16 bytes and contains information such as a pointer to its class. For an object with very little data in it (say one Int field), this can be bigger than the data.
* Java Strings have about 40 bytes of overhead over the raw string data (since they store it in an array of Chars and keep extra data such as the length), and store each character as two bytes due to String’s internal usage of UTF-16 encoding. Thus a 10-character string can easily consume 60 bytes.
* Common collection classes, such as HashMap and LinkedList, use linked data structures, where there is a “wrapper” object for each entry (e.g.Map.Entry). This object not only has a header, but also pointers (typically 8 bytes each) to the next object in the list.
* Collections of primitive types often store them as “boxed” objects such as java.lang.Integer.

This section will start with an overview of memory management in Spark, then discuss specific strategies the user can take to make more efficient use of memory in his/her application. In particular, we will describe how to determine the memory usage of your objects, and how to improve it – either by changing your data structures, or by storing data in a serialized format. We will then cover tuning Spark’s cache size and the Java garbage collector.

## Memory Management Overview

Memory usage in Spark largely falls under one of two categories: execution and storage. Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations, while storage memory refers to that used for caching and propagating internal data across the cluster. In Spark, execution and storage share a unified region (M). When no execution memory is used, storage can acquire all the available memory and vice versa（反之亦然）. Execution may evict storage if necessary, but only until total storage memory usage falls under a certain threshold (R). In other words, R describes a subregion within M where cached blocks are never evicted. Storage may not evict execution due to complexities in implementation.

This design ensures several desirable properties. First, applications that do not use caching can use the entire space for execution, obviating（排除） unnecessary disk spills. Second, applications that do use caching can reserve a minimum storage space (R) where their data blocks are immune to being evicted. Lastly, this approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise of how memory is divided internally.

Although there are two relevant configurations, the typical user should not need to adjust them as the default values are applicable to most workloads:

* spark.memory.fraction expresses the size of M as a fraction of the (JVM heap space - 300MB) (default 0.75). The rest of the space (25%) is reserved for user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records.
* spark.memory.storageFraction expresses the size of R as a fraction of M (default 0.5). R is the storage space within M where cached blocks immune to being evicted by execution.

## Garbage Collection Tuning

The main point to remember here is that the cost of garbage collection is proportional to the number of Java objects（GC的消耗和Java对象是成正比的）, so using data structures with fewer objects (e.g. an array of Ints instead of a LinkedList) greatly lowers this cost. An even better method is to persist objects in serialized form, as described above: now there will be only one object (a byte array) per RDD partition. Before trying other techniques, the first thing to try if GC is a problem is to use serialized caching.

GC can also be a problem due to interference between your tasks’ working memory (the amount of space needed to run the task) and the RDDs cached on your nodes. We will discuss how to control the space allocated to the RDD cache to mitigate this

### Measuring the Impact of GC

The first step in GC tuning is to collect statistics on how frequently garbage collection occurs and the amount of time spent GC. This can be done by adding -verbose:gc -Xloggc:/tmp/executor-gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps to the Java options.

**Advanced GC Tuning**

To further tune garbage collection, we first need to understand some basic information about memory management in the JVM:

* Java Heap space is divided in to two regions Young and Old. The Young generation is meant to hold short-lived objects while the Old generation is intended for objects with longer lifetimes.
* The Young generation is further divided into three regions [Eden, Survivor1, Survivor2].
* A simplified description of the garbage collection procedure: When Eden is full, a minor GC is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2. The Survivor regions are swapped. If an object is old enough or Survivor2 is full, it is moved to Old. Finally when Old is close to full, a full GC is invoked.

The goal of GC tuning in Spark is to ensure that only long-lived RDDs are stored in the Old generation and that the Young generation is sufficiently sized to store short-lived objects. This will help avoid full GCs to collect temporary objects created during task execution. Some steps which may be useful are:

* Check if there are too many garbage collections by collecting GC stats. If a full GC is invoked multiple times for before a task completes, it means that there isn’t enough memory available for executing tasks.
* In the GC stats that are printed, if the OldGen is close to being full, reduce the amount of memory used for caching by loweringspark.memory.storageFraction; it is better to cache fewer objects than to slow down task execution!
* If there are too many minor collections but not many major GCs, allocating more memory for Eden would help. You can set the size of the Eden to be an over-estimate of how much memory each task will need（可以将Eden的大小设置高于task memory的预估值）. If the size of Eden is determined to be E, then you can set the size of the Young generation using the option -Xmn=4/3\*E（将Young generation设置为预估Eden大小的4/3倍）. (The scaling up by 4/3 is to account for space used by survivor regions as well.)
* As an example, if your task is reading data from HDFS, the amount of memory used by the task can be estimated using the size of the data block read from HDFS. Note that the size of a decompressed block is often 2 or 3 times the size of the block. So if we wish to have 3 or 4 tasks’ worth of working space, and the HDFS block size is 64 MB, we can estimate size of Eden to be 4\*3\*64MB.（解压后的HDFS block所占用的空间 往往是HDFS block的2到3倍，如果想在working space上跑3到4个task，HDFS block size为64M，则Eden的预估值为4\*3\*64M）
* Monitor how the frequency and time taken by garbage collection changes with the new settings.

### Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide

<http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>

<http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html>