This information applies to version 2.x of Elasticsearch. For the most up to date information, see the current version of the **Elasticsearch Reference (https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html)**.

Heap: Sizing and Swappingedit (https://github.com/elastic/elasticsearch-definitive-guide/edit/2.x/510_Deployment/50_heap.asciidoc

The default installation of Elasticsearch is configured with a 1 GB heap. For just about every deployment, this number is usually too small. If you are using the default heap values, your cluster is probably configured incorrectly.

There are two ways to change the heap size in Elasticsearch. The easiest is to set an environment variable called `ES_HEAP_SIZE`. When the server process starts, it will read this environment variable and set the heap accordingly. As an example, you can set it via the command line as follows:

```
export ES_HEAP_SIZE=10g
```

Alternatively, you can pass in the heap size via JVM flags when starting the process, if that is easier for your setup:

```
ES_JAVA_OPTS="-Xms10g -Xmx10g" ./bin/elasticsearch ❶
```

❶ Ensure that the min (`Xms`) and max (`Xmx`) sizes are the same to prevent the heap from resizing at runtime, a very costly process.

Generally, setting the `ES_HEAP_SIZE` environment variable is preferred over setting explicit `-Xmx` and `-Xms` values.

Give (less than) Half Your Memory to Luceneedit (https://github.com/elastic/elasticsearch-definitive-guide/edit/2.x/510_Deployment/50_heap.asciidoc)

A common problem is configuring a heap that is *too* large. You have a 64 GB machine—and by golly, you want to give Elasticsearch all 64 GB of memory. More is better!

Heap is definitely important to Elasticsearch. It is used by many in-memory data structures to provide fast operation. But with that said, there is another major user of memory that is *off heap*: Lucene.

Lucene is designed to leverage the underlying OS for caching in-memory data structures. Lucene segments are stored in individual files. Because segments are immutable, these files never change. This makes them very cache friendly, and the underlying OS will happily keep hot segments resident in memory for faster access. These segments include both the inverted index (for fulltext search) and doc values (for aggregations).

Lucene's performance relies on this interaction with the OS. But if you give all available memory to Elasticsearch's heap, there won't be any left over for Lucene. This can seriously impact the performance.

The standard recommendation is to give 50% of the available memory to Elasticsearch heap, while leaving the other 50% free. It won't go unused; Lucene will happily gobble up whatever is left over.

If you are not aggregating on analyzed string fields (e.g. you won't be needing **fielddata (aggregations-and-analysis.html)**) you can consider lowering the heap even more. The smaller you can make the heap, the better performance you can expect from both Elasticsearch (faster GCs) and Lucene (more memory for caching).

Don't Cross 32 GB!edit (https://github.com/elastic/elasticsearch-definitive-guide/edit/2.x/510_Deployment/50_heap.asciidoc)

There is another reason to not allocate enormous heaps to Elasticsearch. As it turns out, the HotSpot JVM uses a trick to compress object pointers when heaps are less than around 32 GB.

In Java, all objects are allocated on the heap and referenced by a pointer. Ordinary object pointers (OOP) point at these objects, and are traditionally the size of the CPU's native *word*: either 32 bits or 64 bits, depending on the processor. The pointer references the exact byte location of the value.

For 32-bit systems, this means the maximum heap size is 4 GB. For 64-bit systems, the heap size can get much larger, but the overhead of 64-bit pointers means there is more wasted space simply because the pointer is larger. And worse than wasted space, the larger pointers eat up more bandwidth when moving values between main memory and various caches (LLC, L1, and so forth).

Java uses a trick called **compressed oops (https://wikis.oracle.com/display/HotSpotInternals/CompressedOops)** to get around this problem. Instead of pointing at exact byte locations in memory, the pointers reference *object offsets*. This means a 32-bit pointer can reference four billion *objects*, rather than four billion bytes. Ultimately, this means the heap can grow to around 32 GB of physical size while still using a 32-bit pointer.

Once you cross that magical ~32 GB boundary, the pointers switch back to ordinary object pointers. The size of each pointer grows, more CPU-memory bandwidth is used, and you effectively lose memory. In fact, it takes until around 40–50 GB of allocated heap before you have the same *effective* memory of a heap just under 32 GB using compressed oops.

The moral of the story is this: even when you have memory to spare, try to avoid crossing the 32 GB heap boundary. It wastes memory, reduces CPU performance, and makes the GC struggle with large heaps.

Just how far under 32gb should I set the JVM?edit (https://github.com/elastic/elasticsearch-definitive-guide/edit/2.x/510_Deployment/50_heap.asciidoc)

Unfortunately, that depends. The exact cutoff varies by JVMs and platforms. If you want to play it safe, setting the heap to `31gb` is likely safe. Alternatively, you can verify the cutoff point for the HotSpot JVM by adding `-XX:+PrintFlagsFinal` to your JVM options and checking that the value of the UseCompressedOops flag is true. This will let you find the exact cutoff for your platform and JVM.

For example, here we test a Java 1.7 installation on MacOSX and see the max heap size is around 32600mb (~31.83gb) before compressed pointers are disabled:

```
$ JAVA_HOME=`/usr/libexec/java_home -v 1.7` java -Xmx32600m -
XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops
     bool UseCompressedOops   := true
$ JAVA_HOME=`/usr/libexec/java_home -v 1.7` java -Xmx32766m -
XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops
     bool UseCompressedOops   = false
```

In contrast, a Java 1.8 installation on the same machine has a max heap size around 32766mb (~31.99gb):

```
$ JAVA_HOME=`/usr/libexec/java_home -v 1.8` java -Xmx32766m -
XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops
     bool UseCompressedOops   := true
$ JAVA_HOME=`/usr/libexec/java_home -v 1.8` java -Xmx32767m -
XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops
     bool UseCompressedOops   = false
```

The moral of the story is that the exact cutoff to leverage compressed oops varies from JVM to JVM, so take caution when taking examples from elsewhere and be sure to check your system with your configuration and JVM.

Beginning with Elasticsearch v2.2.0, the startup log will actually tell you if your JVM is using compressed OOPs or not. You'll see a log message like:

```
[2015-12-16 13:53:33,417][INFO ][env] [Illyana Rasputin] heap size [989.8mb],
compressed ordinary object pointers [true]
```

Which indicates that compressed object pointers are being used. If they are not, the message will say `[false]`.

I Have a Machine with 1 TB RAM!

The 32 GB line is fairly important. So what do you do when your machine has a lot of memory? It is becoming increasingly common to see super-servers with 512–768 GB of RAM.

First, we would recommend avoiding such large machines (see **Hardware (hardware.html)**).

But if you already have the machines, you have two practical options:

- Are you doing mostly full-text search? Consider giving 4-32 GB to Elasticsearch and letting Lucene use the rest of memory via the OS filesystem cache. All that memory will cache segments and lead to blisteringly fast full-text search.
- Are you doing a lot of sorting/aggregations? Are most of your aggregations on numerics, dates, geo_points and `not_analyzed` strings? You're in luck! Give Elasticsearch somewhere from 4-32 GB of memory and leave the rest for the OS to cache doc values in memory.
- Are you doing a lot of sorting/aggregations on analyzed strings (e.g. for word-tags, or SigTerms, etc)? Unfortunately that means you'll need fielddata, which means you need heap space. Instead of one node with more than 512 GB of RAM, consider running two or more nodes on a single machine. Still adhere to the 50% rule, though. So if your machine has 128 GB of RAM, run two nodes, each with just under 32 GB. This means that less than 64 GB will be used for heaps, and more than 64 GB will be left over for Lucene.

If you choose this option, set `cluster.routing.allocation.same_shard.host: true` in your config. This will prevent a primary and a replica shard from colocating to the same physical machine (since this would remove the benefits of replica high availability).

Swapping Is the Death of Performanceedit (https://github.com/elastic/elasticsearch-definitive-guide/edit/2.x/510_Deployment/50_heap.asciidoc)

It should be obvious, but it bears spelling out clearly: swapping main memory to disk will *crush* server performance. Think about it: an in-memory operation is one that needs to execute quickly.

If memory swaps to disk, a 100-microsecond operation becomes one that take 10 milliseconds. Now repeat that increase in latency for all other 10us operations. It isn't difficult to see why swapping is terrible for performance.

The best thing to do is disable swap completely on your system. This can be done temporarily:

```
sudo swapoff -a
```

To disable it permanently, you'll likely need to edit your `/etc/fstab`. Consult the documentation for your OS.

If disabling swap completely is not an option, you can try to lower `swappiness`. This value controls how aggressively the OS tries to swap memory. This prevents swapping under normal circumstances, but still allows the OS to swap under emergency memory situations.

For most Linux systems, this is configured using the `sysctl` value:

```
vm.swappiness=1 ❶
```

❶ A `swappiness` of `1` is better than `0`, since on some kernel versions a `swappiness` of `0` can invoke the OOM-killer.

Finally, if neither approach is possible, you should enable `mlockall`. file. This allows the JVM to lock its memory and prevent it from being swapped by the OS. In your `elasticsearch.yml`, set this:

```
bootstrap.mlockall: true
```

« Don't Touch These Settings! (_don_8217_t_touch_these_settings.html) File Descriptors and MMap » (_file_descriptors_and_mmap.html)