GC tuning Suggestions

What is Garbage Collection ?

Java garbage collection is an automatic process to manage the runtime memory used by application. By doing it automatic, JVM relieves the programmer of the overhead of assigning and freeing up memory resources in a application.

Garbage collection is the process of reclaiming the unused memory space and making it available for the future instances.

The GC operation is based on the premise that most objects used in the Java code are short-lived and can be reclaimed shortly after their creation. Because unreferenced objects are automatically removed from the heap memory, GC makes Java memory-efficient.

Unlike C language the developers need not write code for garbage collection in Java. This is one among the many features that made Java popular and helps programmers to write better Java applications.

Basic steps of garbage collection: Mark, sweep, compact

Mark: The garbage collector scans the heap memory segment and marks all the live objects—that is, objects to which the application holds references. All the objects that have no references to them are eligible for removal.

Sweep: The garbage collector recycles all the unreferenced objects from the heap.

Compact: The sweep step tends to leave many empty regions in heap memory, causing memory fragmentation. Therefore, the compact phase helps arranging the objects into the contiguous blocks at the start of the heap. This in turn helps with the allocation of new objects in sequence.

Generational garbage collection

Garbage collection is more complicated than just running through these steps, however, because most objects are short-lived. Running the mark and compact steps frequently on all the objects on the heap would be inefficient and time-consuming.

To handle short-lived objects efficiently, we expand the simple garbage collection sequence by adding different levels for objects that have been present for different lengths of time. This algorithm is called *generational garbage collection*.

Generational garbage collection divides heap memory into two major partitions, the *young* (also called *nursery*) generation and the *old* (also called *tenured*) generation. There are also several types of garbage collection: *minor* collections for the young generation, *major* collections for the old generation, and *full* collections that do both minor and major collections along with compacting.

Heap Memory

This is where JVM stores object or dynamic data. This is the biggest block of memory area and this is where Garbage Collection(GC) takes place.

The size of heap memory can be controlled using the `Xms`(Initial) and `Xmx`(Max) flags. The entire heap memory is not committed to the Virtual Machine(VM) as some of it is reserved as virtual space and the heap can grow to use this. Heap is further divided into “Young” and “Old” generation space.

- **Young generation:** Young generation or “New Space” is where new objects live and is further divided into “Eden Space” and “Survivor Space”. This space is managed by “Minor GC” also sometimes called “Young GC”
  - **Eden Space**: This is where new objects are created. When we create a new object, memory is allocated here.
  - **Survivor Space**: This is where objects that survived the minor GC are stored. This is divided into two parts, S0 and S1.
- **Old generation:** Old generation or “Tenured Space” is where objects that reached the maximum tenure threshold during minor GC live. This space is managed up by “Major GC”.

Thread Stacks

This is the stack memory area and there is one stack memory per thread in the process. This is where thread-specific static data including method/function frames and pointers to objects are stored. The stack memory limit can be set using the `Xss` flag.

MetaSpace

Stores information about the classes and methods used in the application. This storage area was called Permanent Generation or *perm* in the HotSpot JVM prior to JDK 8, and the area was contiguous with the Java heap. From JDK 8 onward, Permanent Generation has been replaced by Metaspace, which is not contiguous with the Java heap.

Metaspace is allocated in native memory. The MaxMetaspaceSize parameter limits the JVM's use of Metaspace. By default, there is no limit for Metaspace, which starts with a very low size default and grows gradually as needed. Metaspace contains only class metadata; all live Java objects are moved to heap memory. So the size of Metaspace is much lower than Permanent Generation was. Usually, there is no need to specify the maximum Metaspace size unless you face a large Metaspace leak.

Does JVM consumes more memory than -Xmx?

The value of the -Xmx parameter controls the maximum size of the Java heap, which is not the only memory that the JVM allocates. In addition there is the Permanent Generation or Metaspace, CodeCache, native C++ heap used by the other JVM internals, space for the thread stacks (which is 1mb per thread by default on a 64 bit JVM in Java 1.6), direct byte buffers, GC overhead, and other things.

The memory used by a JVM process can be better computed as follows

```
~~
JvmProcessMemory = JvmHeap + PermGen|Metaspace + CodeCache + (ThreadStackSize* Number-of-Threads) +
DirectByteBuffers + Jvm-native-c++-heap
~~
```

The Java process consuming more memory than the amount given by -Xmx is **perfectly normal**.

JVM Memory management: Garbage collection

VM manages the heap memory by garbage collection. In simple terms, it frees the memory used by orphan objects, i.e, objects that are no longer referenced from the Stack directly or indirectly(via a reference in another object) to make space for new object creation.

The garbage collector in JVM is responsible for:

- Memory allocation from OS and back to OS.
- Handing out allocated memory to the application as it requests it.
- Determining which parts of the allocated memory is still in use by the application.
- Reclaiming the unused memory for reuse by the application.

JVM garbage collectors are generational(Objects in Heap are grouped by their age and cleared at different stages). There are many different algorithms available for garbage collection but Mark & Sweep is the most commonly used one.

This type of GC is also referred to as stop-the-world GC as they introduce pause-times in the application while performing GC.

JVM offers few different algorithms to choose from when it comes to GC and there might be few more options available depending on the JDK vendor you use(Like the Shenandoah GC available on OpenJDK, ZGC is available on Oracle JDK).

Key Performance Indicators for choosing right GC

Java offers many garbage collectors to meet different application needs. Choosing the right garbage collector for your application majorly impacts its performance.

The essential criteria are:

- ***Throughput***: The percentage of total time spent in useful application activity versus memory allocation and garbage collection. For example, if your throughput is 95%, that means the application code is running 95% of the time and garbage collection is running 5% of the time. Everyone want higher throughput for any high-load business application.

- ***Latency***: Application's responsiveness, which is affected by garbage collection pauses. In any application interacting with a human or some active process , you want the lowest possible latency.

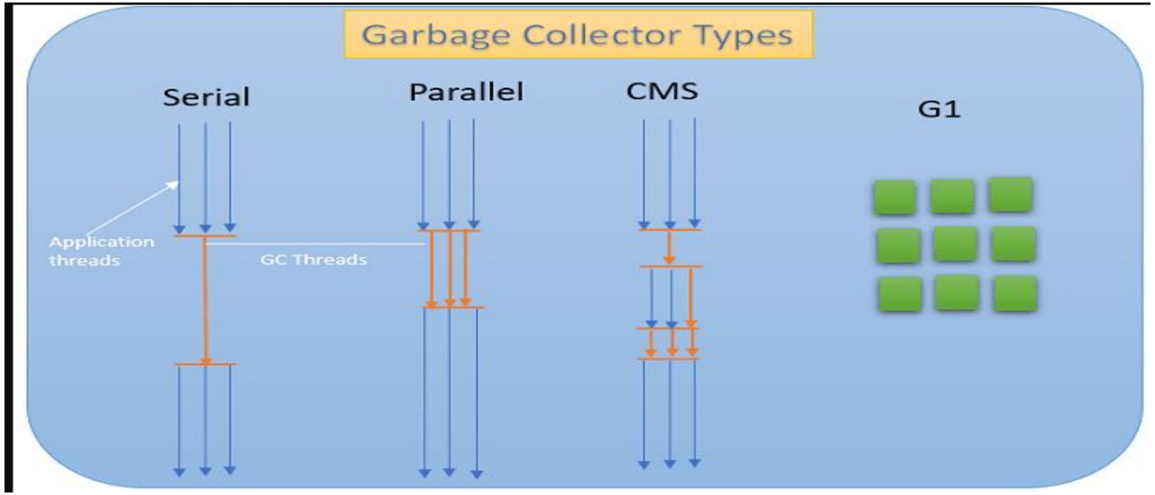
- ***Footprint***: The working set of a process, measured in pages and cache lines. Size of the heap requires for running application.

Different users and applications have different requirements. Some want higher throughput and can bear longer latencies in exchange, whereas others need low latency because even very short pause times would negatively impact their user experience. On systems with limited physical memory or many processes, the footprint might dictate scalability.

Garbage Collectors available

The below garbage collectors are available and the default used is chosen by JVM based on hardware and OS used. We can always specify the GC to be used with the `-XX` switch as well.

- Serial collector :
- Parallel collector
- Garbage-first (G1) collector
- Z collector
- Shenandoah collector
- Concurrent Mark Sweep (CMS) collector (deprecated from JDK 9)



Garbage Collectors available

Serial Collector: It uses a single thread for GC and is efficient for applications with small data sets and is most suitable for single-processor machines. This can be enabled using the `-XX:+UseSerialGC` .

Parallel Collector: This one is focused on high throughput and uses multiple threads to speed up the GC process. This is intended for applications with medium to large data sets running on multi-threaded/multi-processor hardware. This can be enabled using the `-XX:+UseParallelGC` .

Garbage-First(G1) Collector: The G1 collector is mostly concurrent (Means only expensive work is done concurrently). This is for multi-processor machines with a large amount of memory and is enabled as default on most modern machines and OS. It has a focus on low pause times and high throughput. This can be enabled using the `-XX:+UseG1GC.`

Z Garbage Collector: This is a new experimental GC introduced in JDK11. It is a scalable low-latency collector. It’s concurrent and does not stop the execution of application threads, hence no stop-the-world. It is intended for applications that require low latency and/or use a very large heap(multi-terabytes). This can be enabled using the `-XX:+UseZGC`.

Shenandoah collector : Shenandoah is another garbage collector with very short pause times. It reduces pause times by performing more garbage collection work concurrently with the application, including concurrent compaction. Shenandoah's pause time is independent of the heap size. Garbage collecting a 2GB heap or a 200GB heap should have a similar short pause behavior. Shenandoah is best suited to an application that needs responsiveness and short pause times, irrespective of heap size requirements. You can enable this collector through the -XX:+UseShenandoahGC compiler option.

GC process

Regardless of the collector used, JVM has two types of GC process depending on when and where its performed, the minor GC and major GC.

Minor GC

This type of GC keeps the young generation space compact and clean. This is triggered when the below conditions are met:

- JVM is not able to get the required memory from the Eden space to allocate a new object

Initially, all the areas of heap space are empty. Eden memory is the first one to be filled, followed by survivor space and finally by tenured space.

Major GC

Developer calls `System.gc()`, or `Runtime.getRunTime().gc()` from the program or JVM decides there is not enough tenured space as it gets filled up from minor GC cycles the Major GC occurs.

Full GC is cleaning the entire Heap – both Young and Tenured spaces.

Garbage collection logs

Garbage collection behaviour significantly impacts application performance. In order to optimize memory and garbage collection settings and to troubleshoot memory-related problems, one has to analyze Garbage Collection logs. One can always enable garbage collection logs in the production environment due to less overhead & usefulness in analyzing critical performance issues.

The **garbage collector log** is a text file produced by the Java Virtual Machine that describes the work of the garbage collector. It contains all the information you could need to see how the memory cleaning process works. It also shows how the garbage collector behaves and how much resources it uses. Though we can monitor our application using an APM provider or in-house built monitoring tool, the garbage collector log will be invaluable to quickly identify any potential issues and bottlenecks when it comes to heap memory utilization.

Even a very small period of time can provide a lot of information. You see allocation failures, young garbage collection, threads being stopped, memory before and after garbage collection, each event leading to the promotion of the objects inside the heap memory.

Garbage collection logs

Until Java 8: `-XX:+PrintGCDetails -Xloggc:<gc-log-file-path>`

From Java 9: `-Xlog:gc*:file=<gc-log-file-path>`

Refer JVM config lab for optimum JVM configurations for different JDK & GC algorithm. You can refer how to enable GC logs for enabling GC logs.

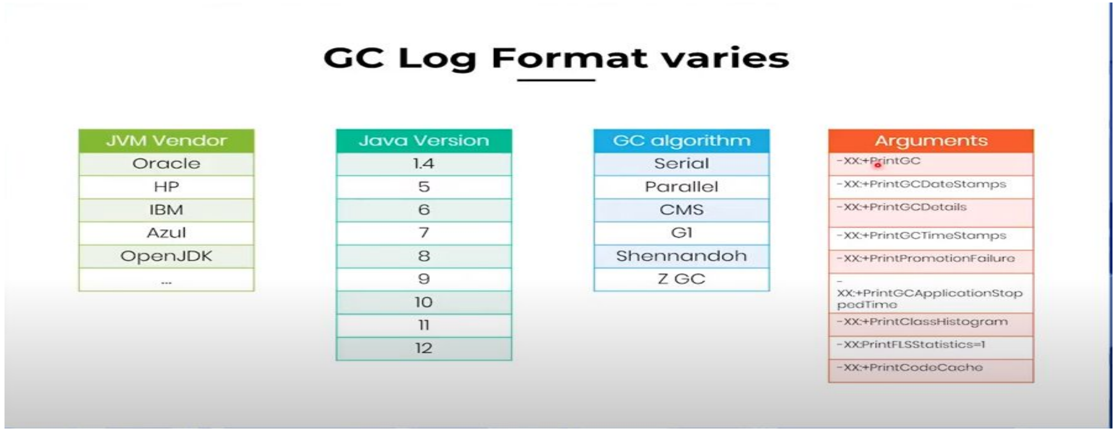
-XX:+PrintGCApplicationStoppedTime. This is required to determine overall throughput and identify throughput and pause issues not related to garbage collection.

The GC log file size (-XX:GCLogFileSize=N[K|M|G]) is small (< 5M). Data needed for troubleshooting could be lost due to excessive log rotation. Consider increasing the size.

-XX:+ExplicitGCInvokesConcurrent so explicit garbage collection is handled concurrently by the CMS and G1 collectors.

GC log format

GC log has rich information, however, understanding GC log is not easy. There isn't sufficient documentation to explain GC log format. On top of it, GC log format is not standardized. It varies by JVM vendor (Oracle, IBM, HP, Azul, …), Java version (1.4, 5, 6, 7, 8, 9), GC algorithm (Serial, Parallel, CMS, G1, Shenandoah), GC system properties that you pass (-XX:+PrintGC, -XX:+PrintGCDetails, -XX:+PrintGCDateStamps, -XX:+PrintHeapAtGC …). Based on this permutation and combination, there are easily 60+ different GC log formats.



GC Log Format varies

JVM Vendor
Oracle
HP
IBM
Azul
OpenJDK
...

Java Version
1.4
5
6
7
8
9
10
11
12

GC algorithm
Serial
Parallel
CMS
G1
Shennandoh
Z GC

Arguments
-XX:+PrintGC
-XX:+PrintGCDateStamps
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-XX:+PrintPromotionFailure
-XX:+PrintGCApplicationStoppedTime
-XX:+PrintClassHistogram
-XX:PrintFLSStatistics=1
-XX:+PrintCodeCache

GC log analysis

To analyze GC logs, it’s highly recommended to use GC log analysis tools such as Garbagecat GCViewer IBM PMAT . These tools parse GC logs and generate great graphical visualizations of data, reports Key Performance Indicators and several other useful metrics.

Troubleshooting Performance Issues related to Garbage collection:

How to determine issue is related to Garbage collection?

-An out-of-memory exception is thrown? e.g. "java.lang.OutOfMemoryError: Java heap space" , "java.lang.OutOfMemoryError: GC overhead limit exceeded".

-The process uses too much memory.

- Garbage collection pauses are too long & multiple Full GC occurrences present.

- CPU usage during a garbage collection is too high.

How to determine issue is related to Garbage collection?

- Frequent humongous allocations

Humongous allocations are allocations over 50% of the G1 region size (the region size is another statistic printed in the GC log at the end of each collection). If your app uses humongous allocations heavily it can lead to excessive memory fragmentation and OutOfMemoryError exceptions.

Humongous allocations problems identified by looking in the GC log for "humongous allocation." Here's an example:

```
[4.765s][info ][gc ] GC(122) Pause Young (Concurrent Start) (G1 Humongous Allocation) 889M->5M(1985M) 3.339ms
```

If your app uses humongous allocations and you think G1GC is spending too much time managing them, you can increase the region size to avoid costly collections of humongous objects by using the **-XX:G1HeapRegionSize** option.

-Young objects are promoted too quickly

In contrast to the previous problem where objects are too large to fight in a G1 region, if the young generation size is too small minor collections will occur less frequently and instead full collections (which take more time to execute) will occur more often.

If you see a large number of **Pause Full** messages in your log you may need to use the **-XX:NewRatio** option to increase the size of the young generation and avoid the more costly full collections.

Garbage Collection tuning tips :

1.Start from scratch

Remove all the unnecessary JVM arguments which might cause bad side effects on GC performance. Also avoid mixing up the different collectors GC JVM options to avoid any negative impacts. Start from scratch or with optimum JVM options with JVM config lab.

2.Study GC causes

One of the effective ways to optimize GC performance is to study the causes triggering the GC and provide solutions to reduce them. Enable the GC logs & analyze them to study the GC causes .

Probable causes for GC , e.g. for G1GC

- Full GC – Allocation Failure
- G1 Evacuation Pause or Evacuation Failure
- G1 Humongous Allocation
- System.gc()

PS: The options should not be directly applied to Production environment. Do pre-prod performance test as per applications requirement .

Garbage Collection tuning tips :

Probable causes for GC , e.g. for G1GC

- Full GC – Allocation Failure

Full GC – Allocation failures happen for two reasons:

- Application is creating too many objects that can’t be reclaimed quickly enough.
- When heap is fragmented, direct allocations in the Old generation may fail even when there is a lot of free space

Here are the potential solutions to address this problem:

1. Increase the number of concurrent marking threads by setting ‘-XX:ConcGCThreads’ value. Increasing the concurrent marking threads will make garbage collection run fast.
2. Force G1 to start marking phase earlier. This can be achieved by lowering ‘-XX:InitiatingHeapOccupancyPercent’ value. Default value is 45. It means the **G1 GC** marking phase will begin only when heap usage reaches 45%. By lowering the value, the G1 GC marking phase will get triggered earlier so that **Full GC** can be avoided.
3. Even though there is enough space in a heap, a Full GC can also occur due to lack of a contiguous set of space. This can happen because of a lot of humongous objects present in the memory . A potential solution to solve this problem is to increase the heap region size by using the option ‘-XX:G1HeapRegionSize’ to decrease the amount of memory wasted by humongous objects.

PS: These are subject to proper performance/load testing and specification and should not be directly applied to Production environment.

Garbage Collection tuning tips :

G1 Evacuation Pause or Evacuation Failure

-When you see G1 Evacuation pause, then G1 GC does not have enough memory for either survivor or promoted objects or both. The Java heap cannot expand since it is already at its max. Below are the potential solutions to fix this problem:

-Increase the value of the '-XX:G1ReservePercent' argument. Default value is 10%. It means the G1 garbage collector will try to keep 10% of memory free always. When you try to increase this value, GC will be triggered earlier, preventing the Evacuation pauses.

-Start the marking cycle earlier by reducing the '-XX:InitiatingHeapOccupancyPercent'. The default value is 45. Reducing the value will start the marking cycle earlier. GC marking cycles are triggered when heap's usage goes beyond 45%. On the other hand, if the marking cycle is starting early and not reclaiming, increase the '-XX:InitiatingHeapOccupancyPercent' threshold above the default value.

-You can also increase the value of the '-XX:ConcGCThreads' argument to increase the number of parallel marking threads. Increasing the concurrent marking threads will make garbage collection run fast.

-If the problem persists you may consider increasing JVM heap size (i.e. -Xmx)

PS: These are subject to proper performance/load testing and specification and should not be directly applied to Production environment.

Garbage Collection tuning tips :

G1 Humongous Allocation

Any object that is more than half a region size is considered a “Humongous object”. If the regions contain humongous objects, space between the last humongous object in the region and the end of the region will be unused. If there are multiple such humongous objects, this unused space can cause the heap to become fragmented. Heap fragmentation is detrimental to application performance. If you see several Humongous allocations, please increase your ‘-XX:G1HeapRegionSize’. The value will be a power of 2 and can range from 1MB to 32MB.

3. Avoid System.gc calls from application

When ‘System.gc()’ or ‘Runtime.getRuntime().gc()’ API calls are invoked from your application, stop-the-world Full GC events will be triggered.

Refer explicit GC for Java explicit garbage collection disable.

Garbage Collection tuning tips :

4. Choice of Garbage collector

One common reason for GC issues is using the wrong collector for the type of application you're developing.

HotSpot's default is the Parallel / Throughput collector and often it's not the best choice for your application.

The act of choosing the right collector (via JVM flags) is a statement of your priorities from the system and usually the first issue to consider. Generally, the CMS and G1 collectors who are mostly concurrent will cause less frequent pauses. Although when a pause do comes, its duration will probably be longer than the one caused by the Parallel collector as their fallback mechanism is single threaded .On the other hand, the parallel collector will achieve higher throughput for the same size of heap.

Another guideline relates to predictability, if predictable performance is an important factor and the heap size isn't large the Parallel collector might be the answer. And if the average response time / latency is your top priority, then CMS or G1 are most likely the answer. Choosing right garbage collector is totally dependent on application behaviour & what are application's priority such as latency or throughput.

Garbage Collection tuning tips :

5. Tuning the heap size.

The heap size, the new to old gen ratio, the survivor space size and the max tenuring threshold , tenure space , How many GC cycles does it take for an object to move to old gen are the key important things in JVM performance.

It is recommended that you test your applications under load in a development environment to determine the maximum memory usage. Your production heap size should be at least 25% higher than the tested maximum to allow room for overhead.

It is recommended to increase the Java heap space only up to half of the total RAM available on the server. Increasing the Java heap space beyond that value might cause performance problems. For example, if your server has 8 GB of RAM available, then the maximum heap space you should use is 4 GB.

Garbage Collection tuning tips :

5. Write a GC efficient application code

- Avoid using duplicate strings.
- Avoid creating high number objects .
- Inefficient collections such as developer initialize something but not using it causing wastage of resources.
- Use Immutable Objects for greater throughput due to less GC pauses.

Thank You