Analysis Report

Tips to reduce GC Time

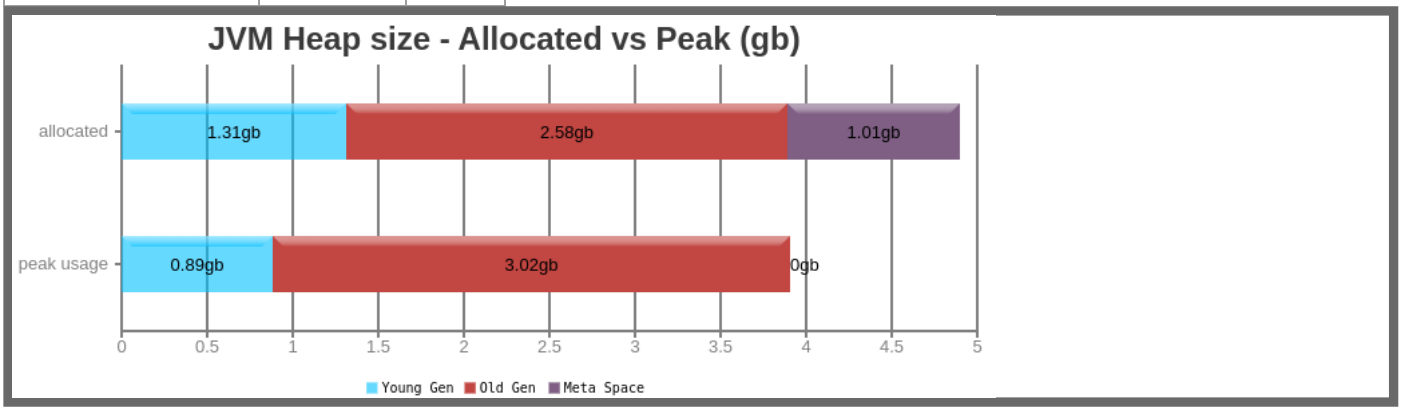
(**CAUTION:** Please do thorough testing before implementing out the recommendations. These are generic recommendations & may not be applicable for your application.)

- **29.71%** of GC time (i.e 1 sec 255 ms) is caused by **'G1 Humongous Allocation'**. Humongous allocations are allocations that are larger than 50% of the region size in G1. Frequent humongous allocations can cause couple of performance issues:1. 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.2. Until Java 1.8u40 reclamation of humongous regions were only done during full GC events. Where as in the newer JVMs, clearing humongous objects are done in cleanup phase..
  Solution:
  You can increase the G1 region size so that allocations would not exceed 50% limit. By default region size is calculated during startup based on the heap size. It can be overriden by specifying '-XX:G1HeapRegionSize' property. Region size must be between 1 and 32 megabytes and has to be a power of two. Note: Increasing region size is sensitive change as it will reduce the number of regions. So before increasing new region size, do thorough testing.

- **25.98%** of GC time (i.e 1 sec 98 ms) is caused by **'Evacuation Failure'**. When there are no more free regions to promote to the old generation or to copy to the survivor space, and the heap cannot expand since it is already at its maximum, an evacuation failure occurs. For G1 GC, an evacuation failure is very expensive - a. For successfully copied objects, G1 needs to update the references and the regions have to be tenured. b. For unsuccessfully copied objects, G1 will self-forward them and tenure the regions in place.
  Solution:
  1. Evacuation failure might happen because of over tuning. So eliminate all the memory related properties and keep only min and max heap and a realistic pause time goal (i.e. Use only -Xms, -Xmx and a pause time goal -XX:MaxGCPauseMillis). Remove any additional heap sizing such as -Xmn, -XX:NewSize, -XX:MaxNewSize, -XX:SurvivorRatio, etc.
  2. If the problem still persists then increase JVM heap size (i.e. -Xmx)
  3. If you can't increase the heap size and if you notice that the marking cycle is not starting early enough to reclaim the old generation then reduce -XX:InitiatingHeapOccupancyPercent. The default value is 45%. Reducing the value will start the marking cycle earlier. On the other hand, if the marking cycle is starting early and not reclaiming, increase the -XX:InitiatingHeapOccupancyPercent threshold above the default value.
  4. If concurrent marking cycles are starting on time, but takes long time to finish then increase the number of concurrent marking thread count using the property: '-XX:ConcGCThreads'.
  5. If there are lot of 'to-space exhausted' or 'to-space overflow' GC events, then increase the -XX:G1ReservePercent. The default is 10% of the Java heap. Note: G1 GC caps this value at 50%.

JVM Heap Size

Generation	Allocated	Peak
Young Generation	1.31 gb	907 mb
Old Generation	2.58 gb	3.02 gb
Meta Space	1.01 gb	3.82 mb
Young + Old + Meta space	4.9 gb	3.02 gb



Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here (https://blog.gceasy.io/2016/10/01/garbage-collection-kpi/))

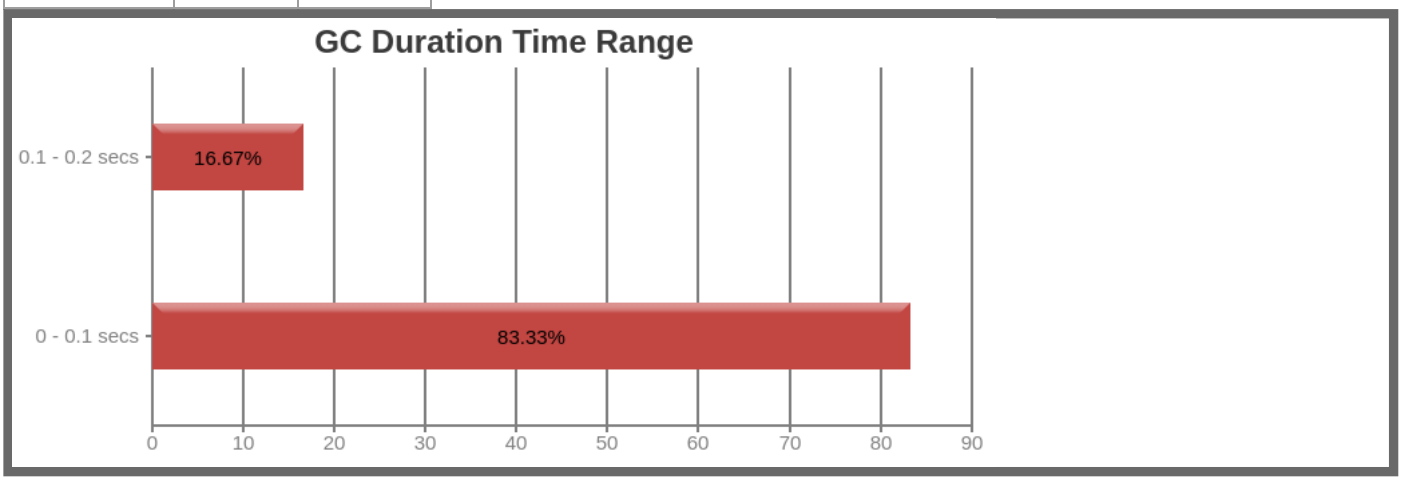
1 **Throughput : 72.62%**

2 **Latency:**

Avg Pause GC Time	**33 ms**
Max Pause GC Time	**200 ms**

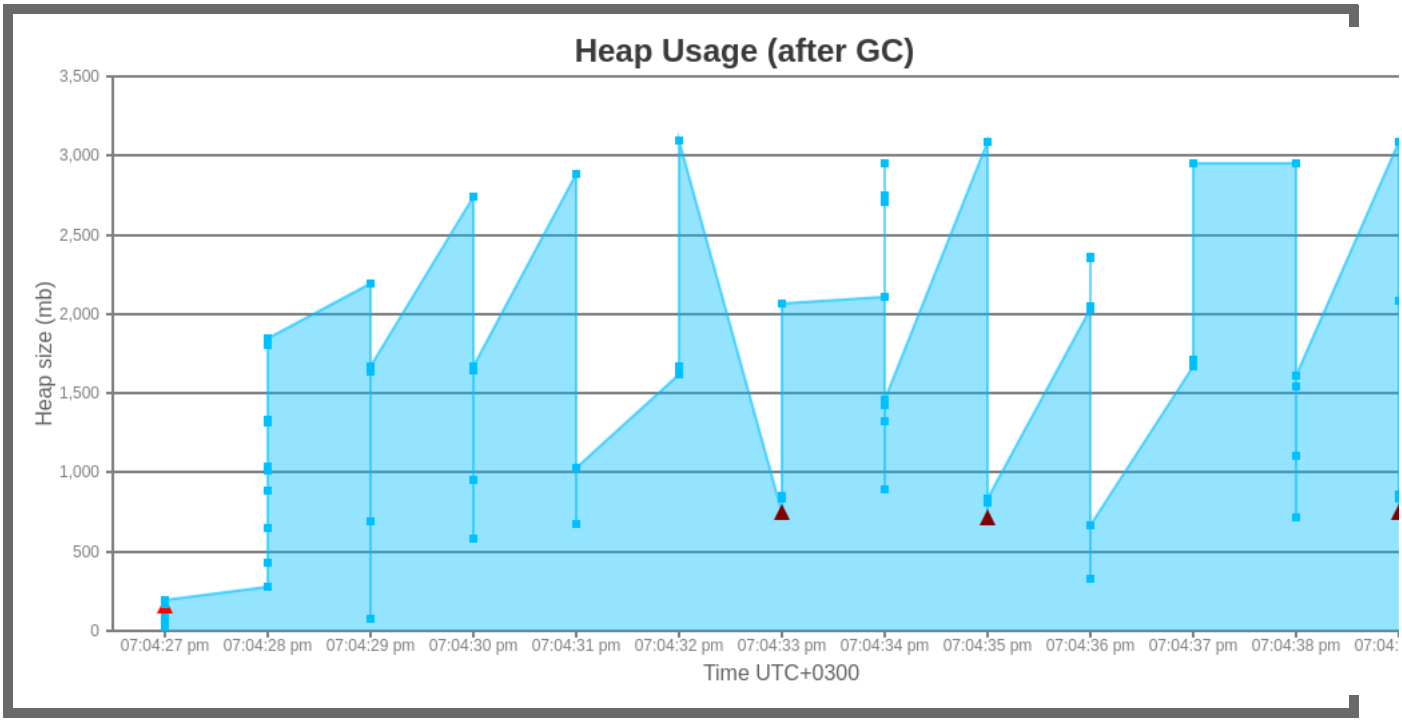
GC **Pause** Duration Time Range :

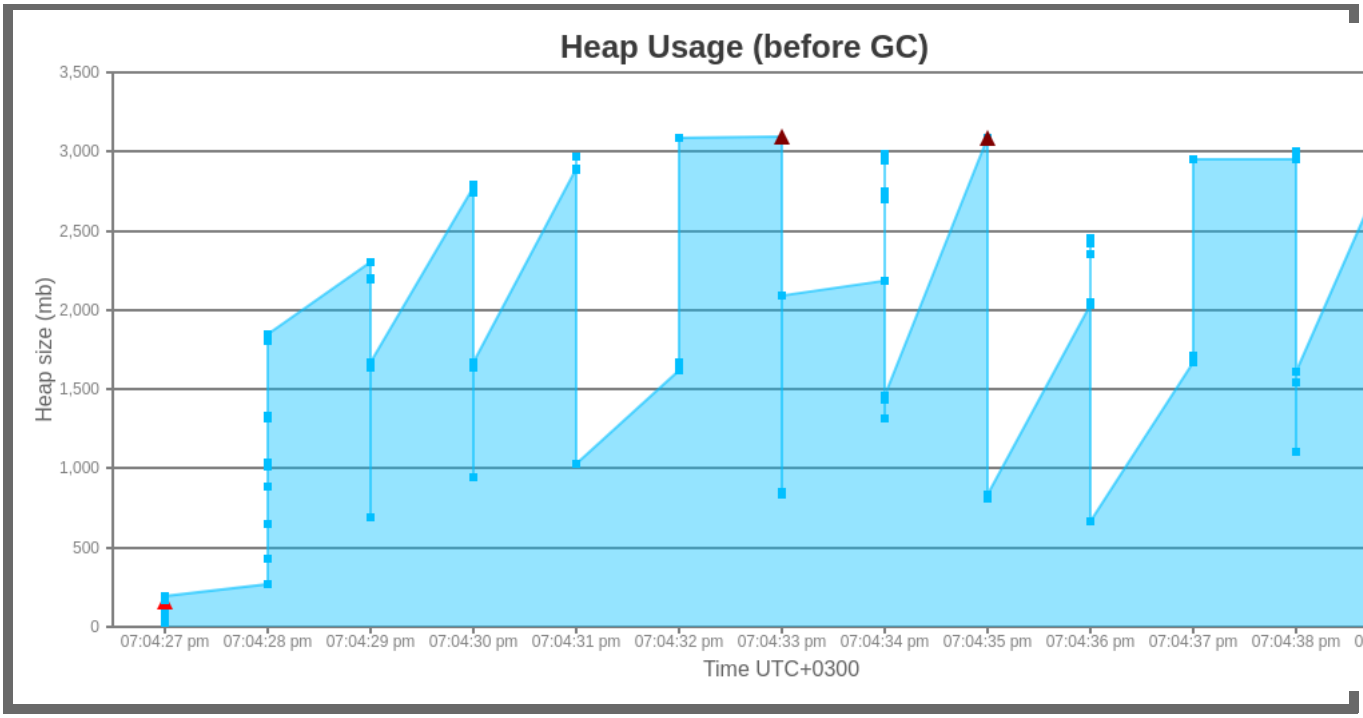
Duration (secs)	No. of GCs	Percentage
0 - 0.1	55	83.333%
0.1 - 0.2	11	100.0%

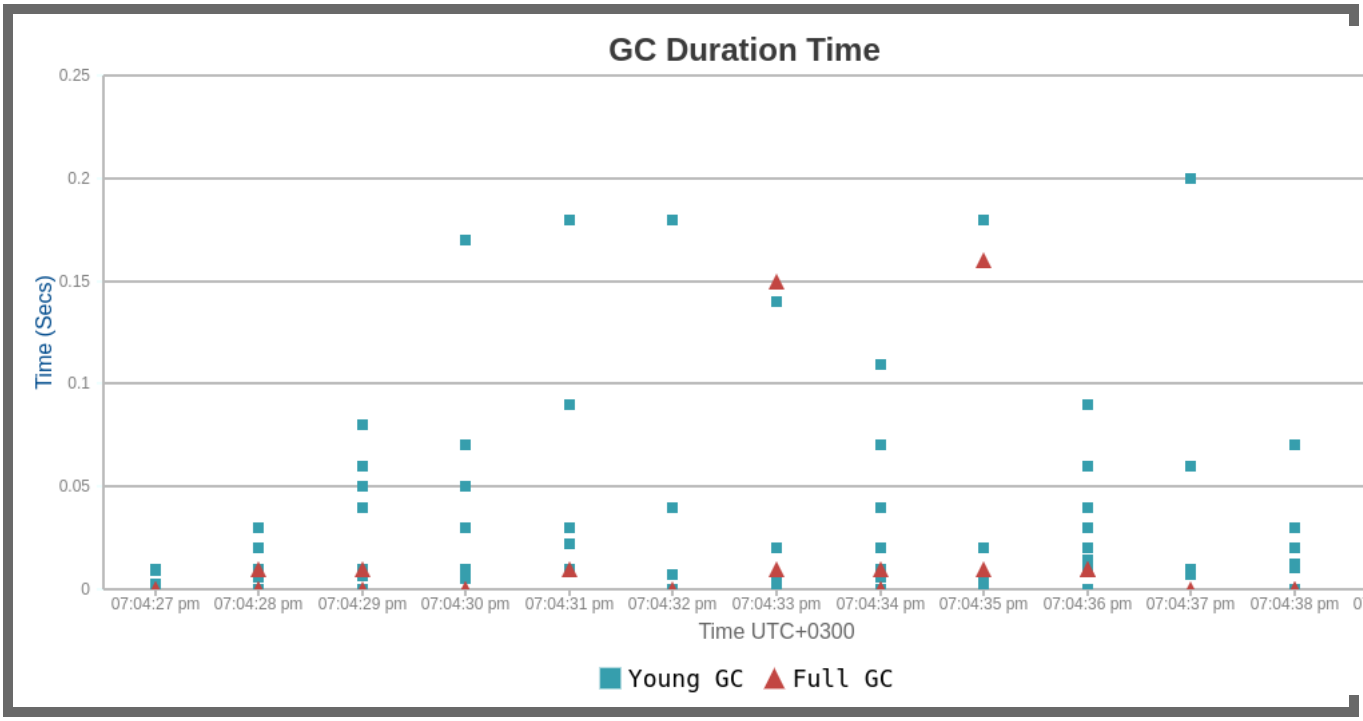


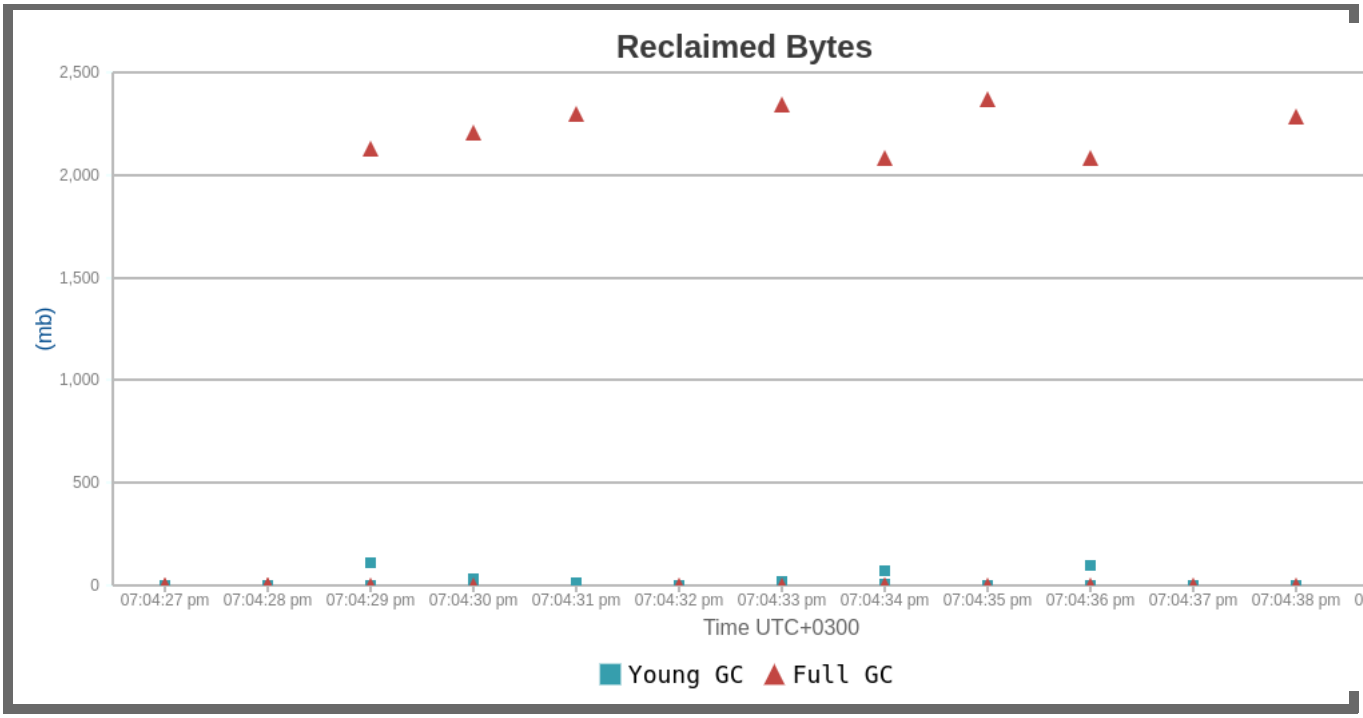
Interactive Graphs

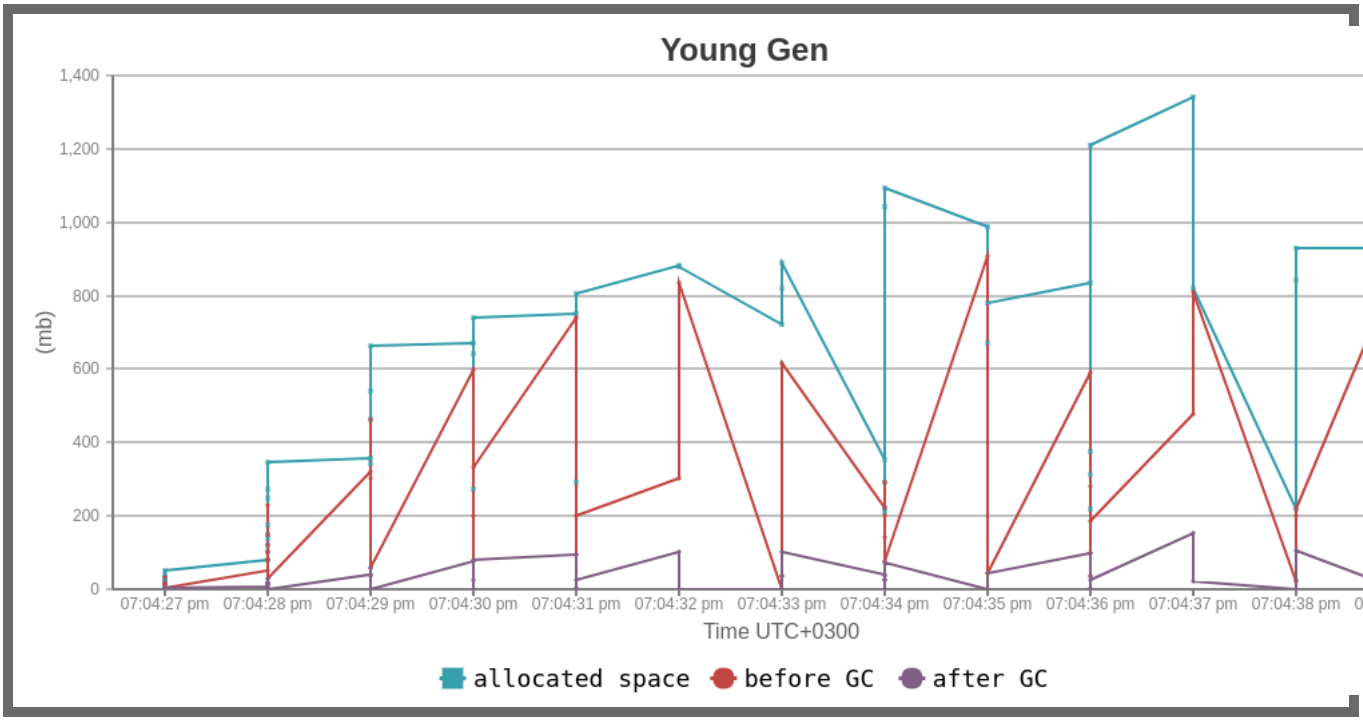
(All graphs are zoomable)



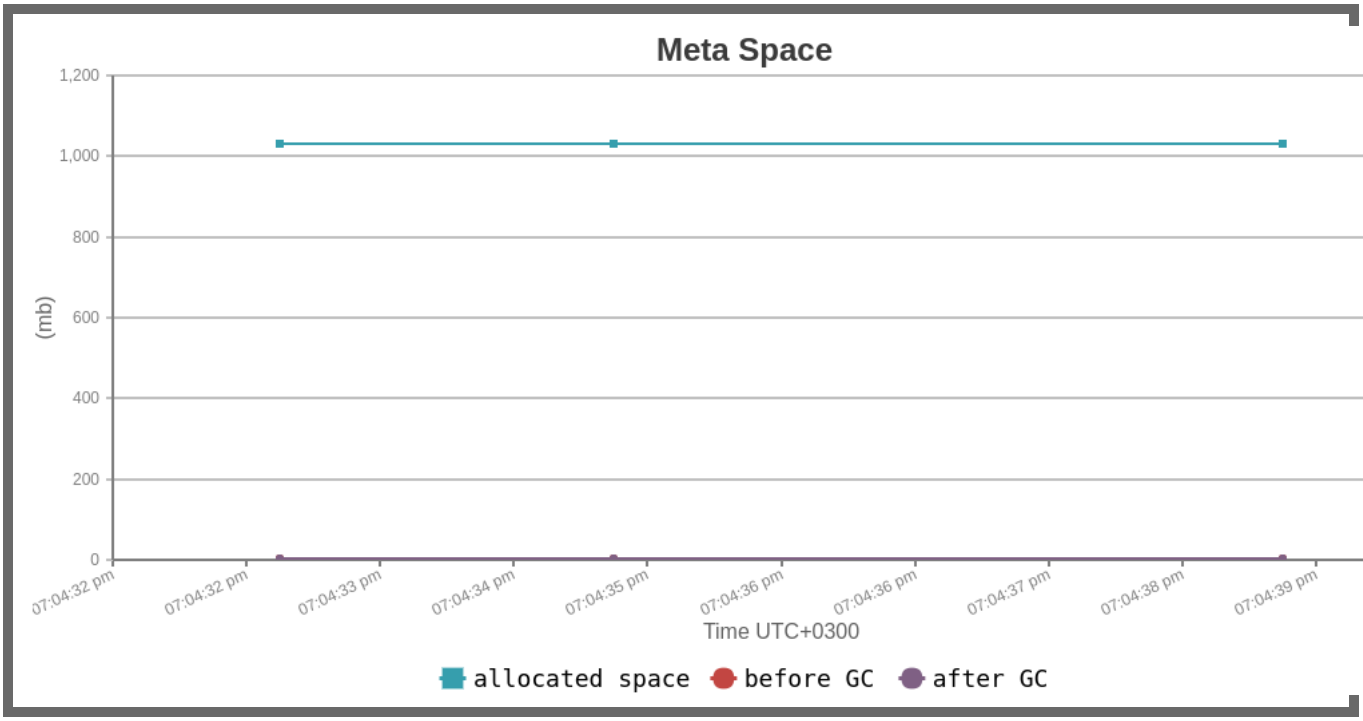

Heap Usage (before GC)
Heap size (mb)
3,500
3,000
2,500
2,000
1,500
1,000
500
0
07:04:27 pm
07:04:28 pm
07:04:29 pm
07:04:30 pm
07:04:31 pm
07:04:32 pm
07:04:33 pm
07:04:34 pm
07:04:35 pm
07:04:36 pm
07:04:37 pm
07:04:38 pm
Time UTC+0300

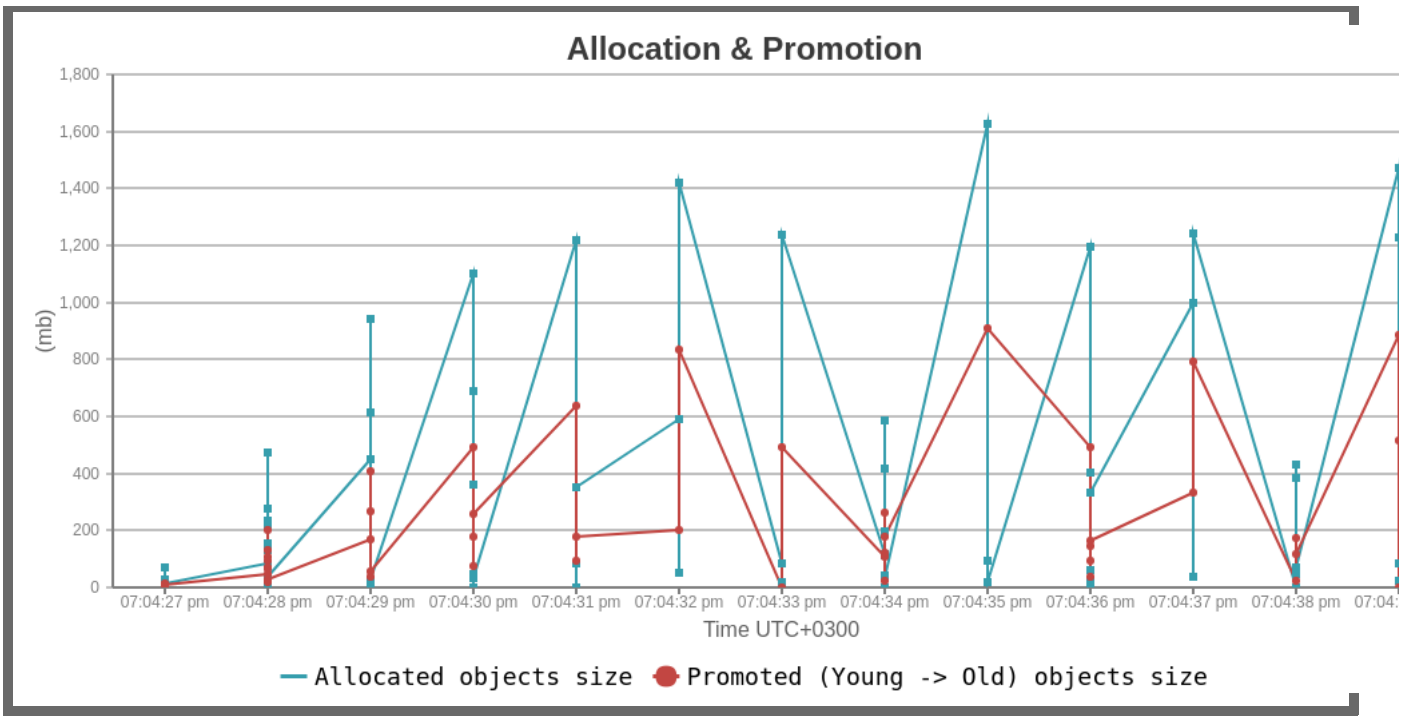

GC Duration Time
Time (Secs)
0.25
0.2
0.15
0.1
0.05
0
07:04:27 pm
07:04:28 pm
07:04:29 pm
07:04:30 pm
07:04:31 pm
07:04:32 pm
07:04:33 pm
07:04:34 pm
07:04:35 pm
07:04:36 pm
07:04:37 pm
07:04:38 pm
Time UTC+0300
Young GC
Full GC


Reclaimed Bytes
2,500
2,000
1,500
1,000
500
0
(mb)
07:04:27 pm
07:04:28 pm
07:04:29 pm
07:04:30 pm
07:04:31 pm
07:04:32 pm
07:04:33 pm
07:04:34 pm
07:04:35 pm
07:04:36 pm
07:04:37 pm
07:04:38 pm
Time UTC+0300
Young GC
Full GC

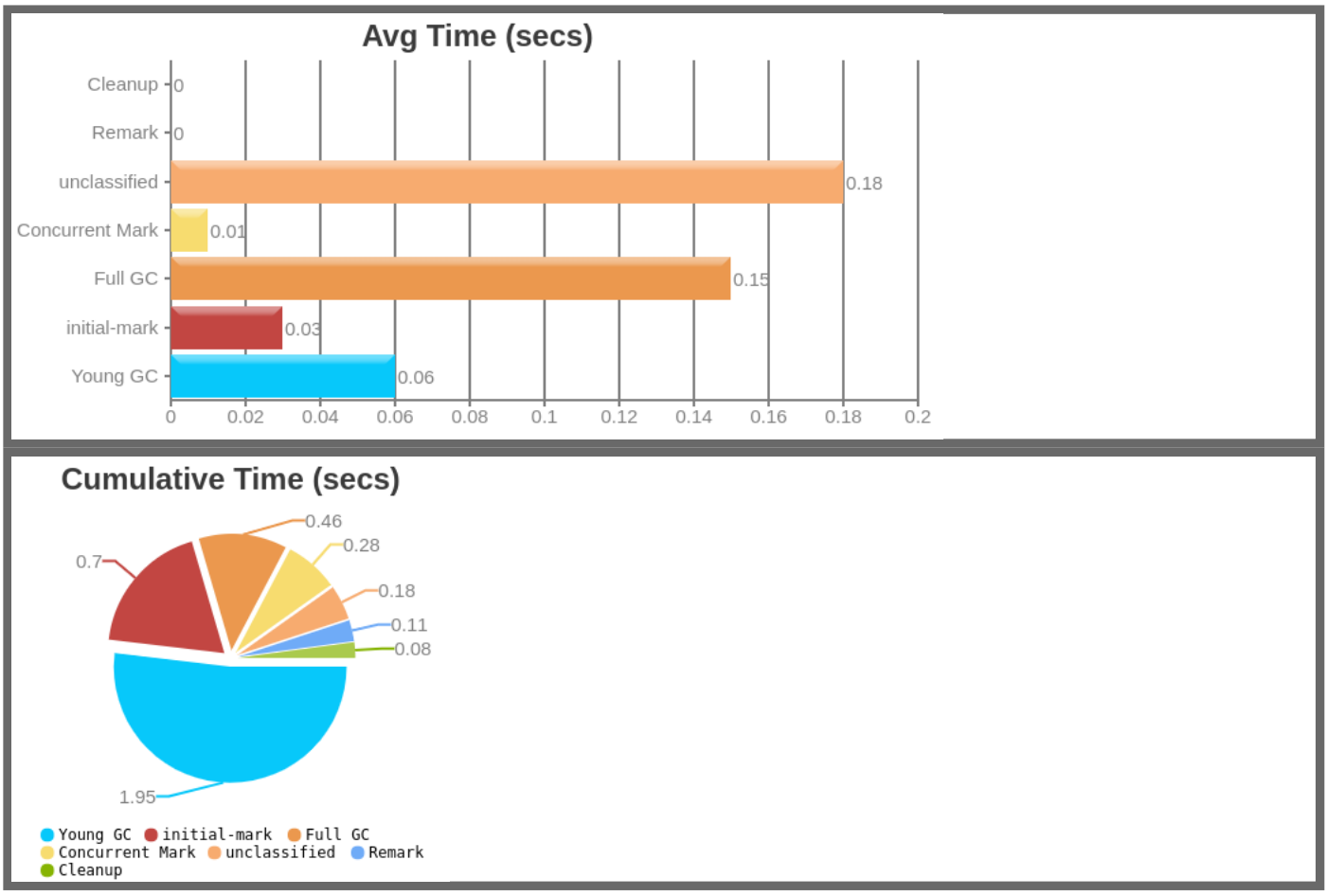

Young Gen
1,400
1,200
1,000
800
600
400
200
0
(mb)
07:04:27 pm
07:04:28 pm
07:04:29 pm
07:04:30 pm
07:04:31 pm
07:04:32 pm
07:04:33 pm
07:04:34 pm
07:04:35 pm
07:04:36 pm
07:04:37 pm
07:04:38 pm
Time UTC+0300
allocated space
before GC
after GC


Old Gen
3,500
3,000
2,500
2,000
1,500
1,000
500
0
(mb)
07:04:27 pm
07:04:28 pm
07:04:29 pm
07:04:30 pm
07:04:31 pm
07:04:32 pm
07:04:33 pm
07:04:34 pm
07:04:35 pm
07:04:36 pm
07:04:37 pm
07:04:38 pm
Time UTC+0300
allocated space
before GC
after GC


Meta Space
1,200
1,000
800
600
400
200
0
(mb)
07:04:32 pm
07:04:32 pm
07:04:33 pm
07:04:34 pm
07:04:35 pm
07:04:36 pm
07:04:36 pm
07:04:37 pm
07:04:38 pm
07:04:39 pm
Time UTC+0300
allocated space
before GC
after GC



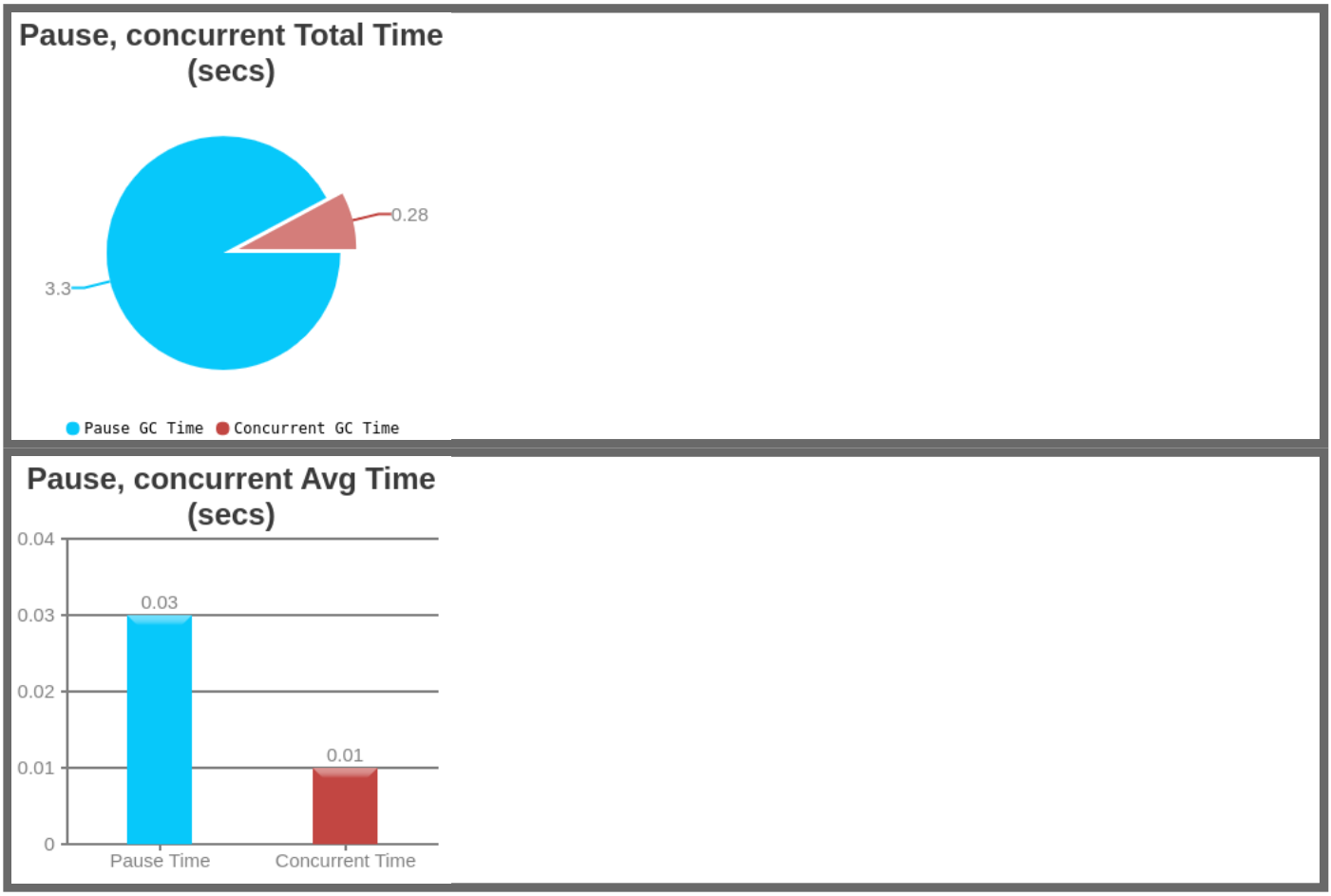
G1 Collection Phases Statistics



	Young GC	initial-mark	Full GC	Concurrent Mark	unclassified	Remark	Cleanup	Total
Count	30	22	3	22	1	22	22	122
Total GC Time	1 sec 950 ms	700 ms	460 ms	282 ms	180 ms	110 ms	80 ms	3 sec 762 ms

Avg GC Time	65 ms	32 ms	153 ms	13 ms	180 ms	5 ms	4 ms	31 ms
Avg Time std dev	66 ms	28 ms	5 ms	21 ms	0	5 ms	5 ms	50 ms
Min/Max Time	0 / 200 ms	0 / 90 ms	0 / 160 ms	0 / 110 ms	0 / 180 ms	0 / 10 ms	0 / 10 ms	0 / 200 ms
Avg Interval Time	415 ms	556 ms	3 sec 117 ms	547 ms	n/a	547 ms	547 ms	560 ms

G1 GC Time



Pause Time

Total Time	3 sec 300 ms
Avg Time	33 ms
Std Dev Time	51 ms
Min Time	0
Max Time	200 ms

Concurrent Time

Total Time	282 ms
Avg Time	13 ms
Std Dev Time	21 ms
Min Time	2 ms

Max Time	110 ms

Object Stats

(These are perfect micro-metrics (https://blog.gceasy.io/2017/05/30/improving-your-performance-reports/) to include in your performance reports)

Total created bytes	21.98 gb
Total promoted bytes	10.32 gb
Avg creation rate	1.82 gb/sec
Avg promotion rate	877.08 mb/sec

Memory Leak

No major memory leaks.

(**Note:** there are 8 flavours of OutOfMemoryErrors (https://tier1app.files.wordpress.com/2014/12/outofmemoryerror2.pdf). With GC Logs you can diagnose only 5 flavours of them(Java heap space, GC overhead limit exceeded, Requested array size exceeds VM limit, Permgen space, Metaspace). So in other words, your application could be still suffering from memory leaks, but need other tools to diagnose them, not just GC Logs.)

Consecutive Full GC

None.

Long Pause

None.

Safe Point Duration

(To learn more about SafePoint duration, click here (https://blog.gceasy.io/2016/12/22/total-time-for-which-application-threads-were-stopped/))

Not Reported in the log.

GC Causes

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
G1 Evacuation Pause	21	68 ms	195 ms	1 sec 418 ms	33.57%
G1 Humongous Allocation	25	50 ms	185 ms	1 sec 255 ms	29.71%
Evacuation Failure	7	157 ms	195 ms	1 sec 98 ms	25.98%
Full GC - Allocation Failure	3	151 ms	152 ms	454 ms	10.74%
Total	56	n/a	n/a	4 sec 224 ms	100.0%