GC Intelligence Report

g1gc_1.log

Duration: 44 min 47 sec 405 ms

System Time greater than User Time

WARNING

In **7 GC event's**, 'sys' time is greater than 'usr' time. It's not a healthy sign. Read our recommendations to **reduce sys time**

Timestamp	User Time (secs)	Sys Time (secs)	Real Time (secs)
2025-10-13T17:29:55.496	0.11	0.32	0.01
2025-10-13T17:29:55.784	0.26	0.35	0.01
2025-10-13T17:30:11.770	0.95	1.18	0.05
2025-10-13T17:30:15.229	1.45	1.87	0.08
2025-10-13T17:30:21.132	1.68	2.09	0.08
2025-10-13T17:30:25.885	1.99	2.38	0.12
2025-10-13T17:31:02.955	2.29	2.47	0.16

Recommendations

(**CAUTION:** Please do thorough testing before implementing below recommendations.)

- **32 sec 802 ms** of GC pause time is triggered by **'G1 Evacuation Pause'** event. This GC is triggered when copying live objects from one set of regions to another set of regions. When Young generation regions are only copied then Young GC is triggered. When both Young + Tenured regions are copied, Mixed GC is triggered..

  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. 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.

  5. 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. Note: G1 GC caps this value at 50%.

- **3 sec 248 ms** of GC pause time is triggered by **'G1 Humongous Allocation'** event. 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.

- **170 ms** of GC pause time is triggered by **'Metadata GC Threshold'** event. This type of GC event is triggered under two circumstances:

- **172 ms** of GC pause time is triggered by '**Metadata GC Threshold**' event. This type of GC event is triggered under two circumstances:
  1. Configured metaspace size is too small than the actual requirement
  2. There is a classloader leak (very unlikely, but possible).

  Solution:
  You may consider setting '-XX:MaxMetaspaceSize' to a higher value. If this property is not present already please configure it. Setting these arguments to a higher value will reduce 'Metadata GC Threshold' frequency. If you still continue to see 'Metadata GC Threshold' event reported, then you need to inspect metaspace contents. Learn how to inspect metaspace contents from **this article.**

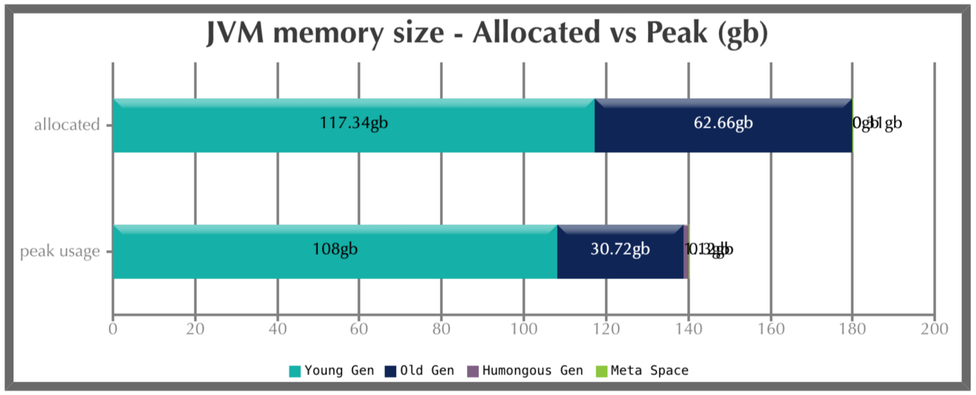
- It looks like you are using G1 GC algorithm. If you are running on Java 8 update 20 and above, you may consider passing **-XX:+UseStringDeduplication** to your application. It will remove duplicate strings in your application and has potential to improve overall application's performance. You can learn more about this property in **this article.**

- This application is using the G1 GC algorithm. If you are looking to tune G1 GC performance even further, here are the **important G1 GC algorithm related JVM arguments**

JVM memory size

(To learn about JVM Memory, click here)

Generation	Allocated	Peak
Young Generation	117.34 gb	108 gb
Old Generation	62.66 gb	30.72 gb
Humongous	n/a	1.12 gb
Meta Space	317.19 mb	311.34 mb
Young + Old + Meta space	180.31 gb	138.37 gb



Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here)

1. **Throughput : 98.369%**

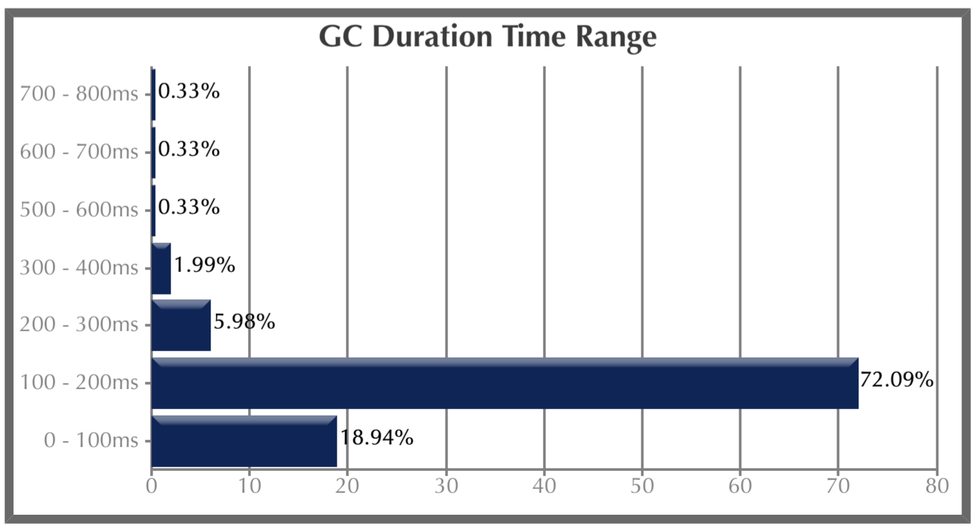
2. **CPU Time : 51 min 57 sec 970 ms**

3. **Latency:**

Avg Pause GC Time	**146 ms**
Max Pause GC Time	**710 ms**

GC Pause Duration Time Range :

Duration (ms) 100 ms Change	No. of GCs	Percentage
0 - 100	57	18.94%
100 - 200	217	72.09%



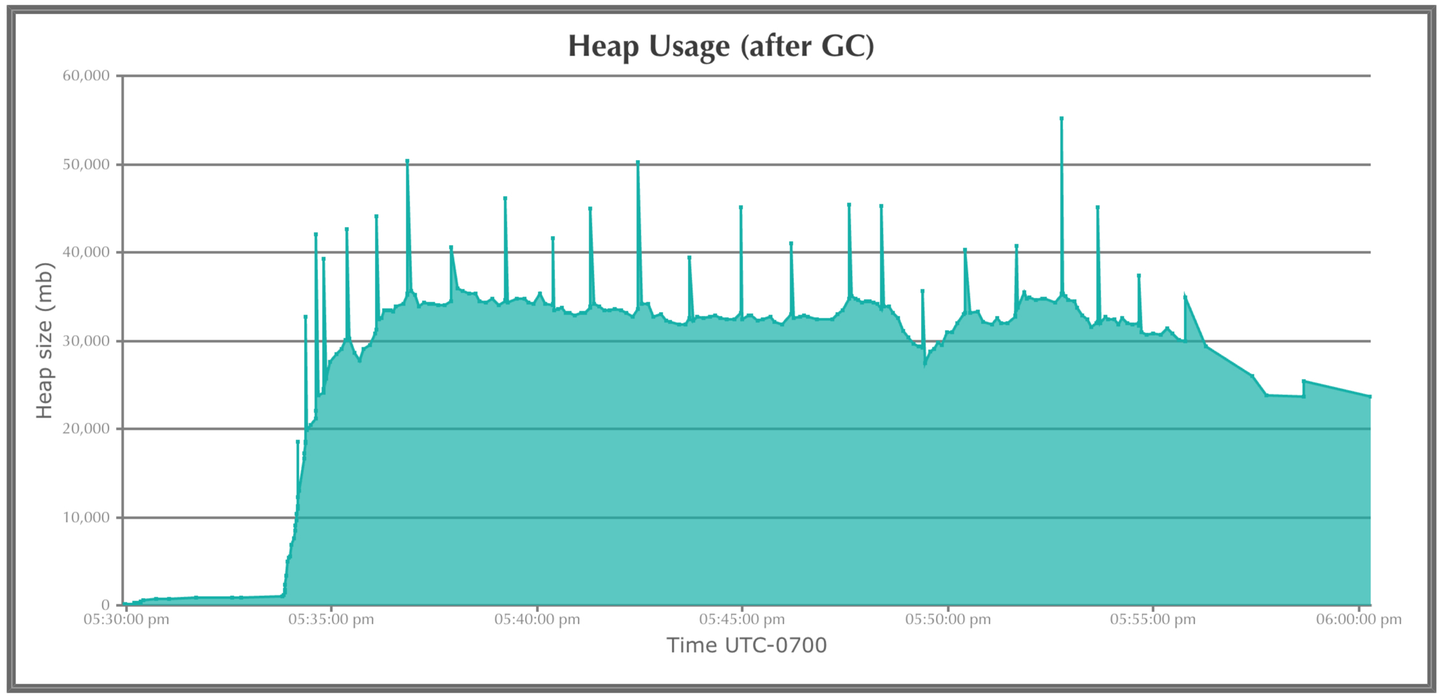
200 - 300	18	5.98%
300 - 400	6	1.99%
500 - 600	1	0.33%
600 - 700	1	0.33%
700 - 800	1	0.33%

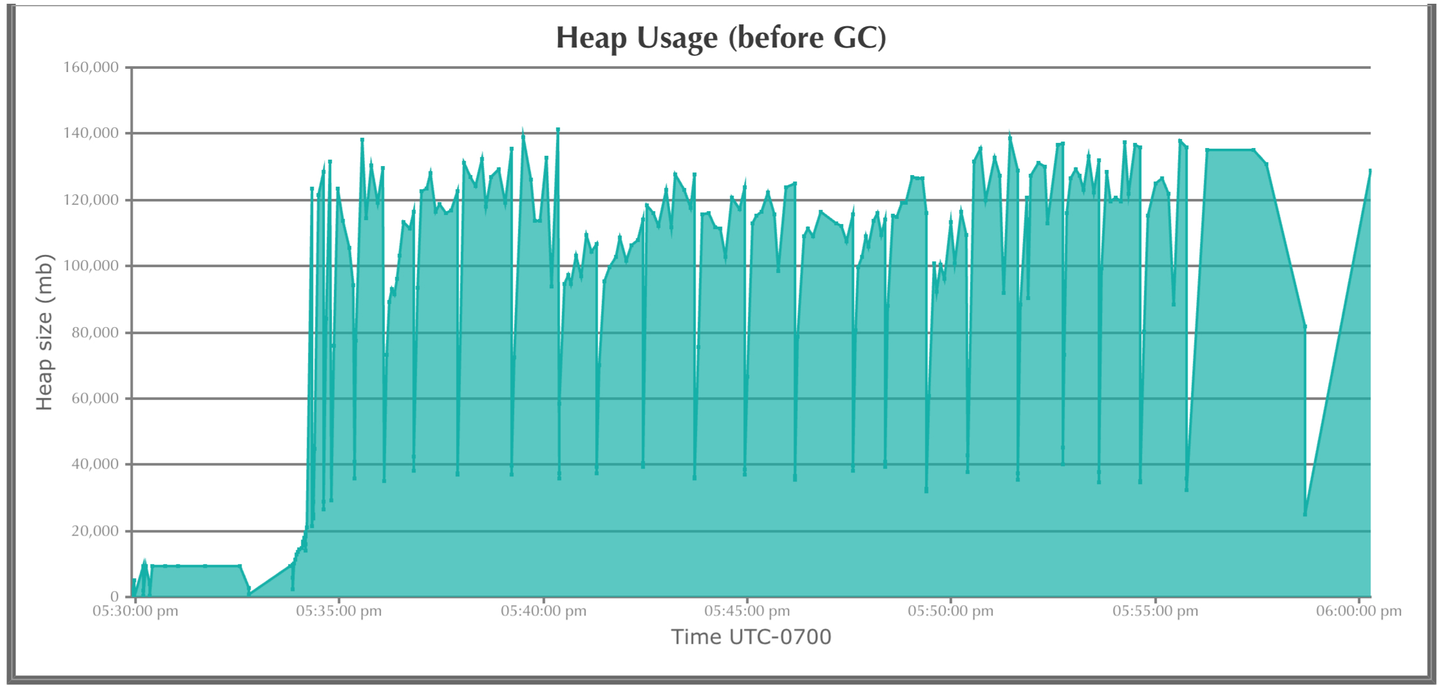
Analyze Specific Time Periods (Beta)

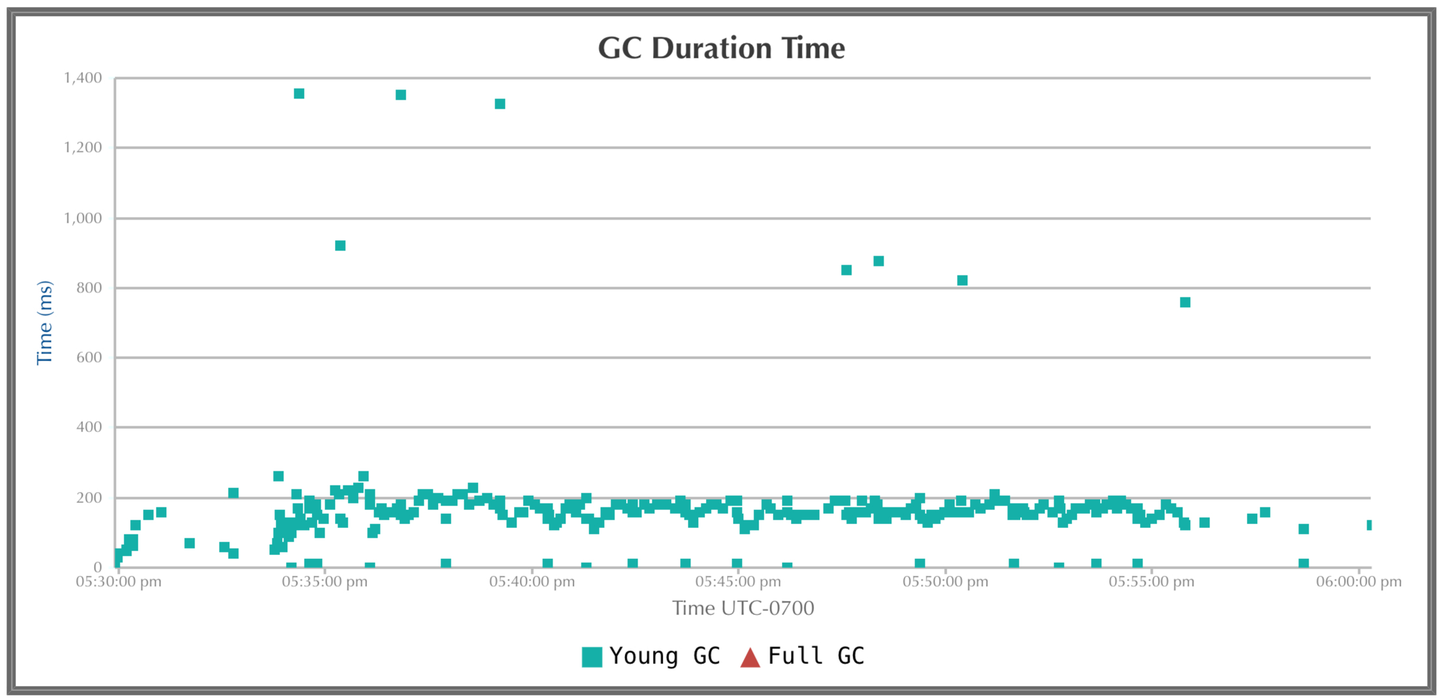
Select time range

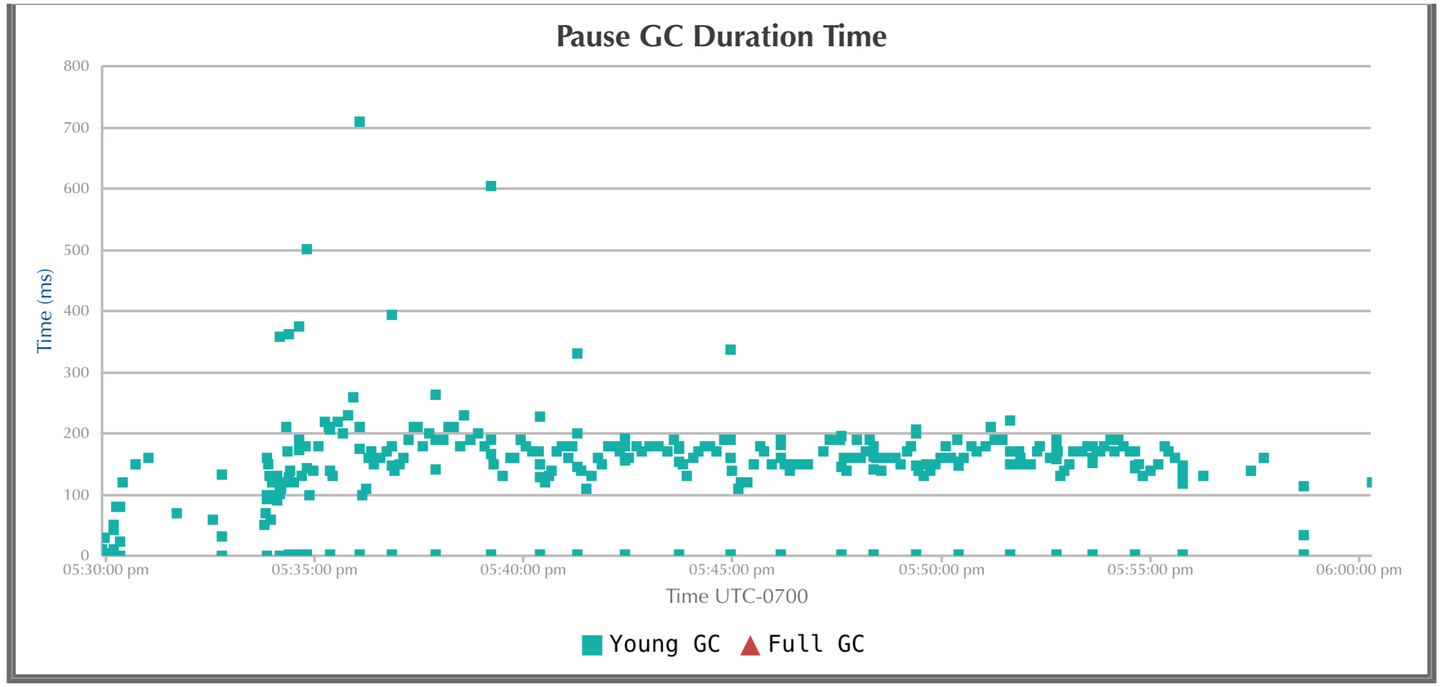
Reset

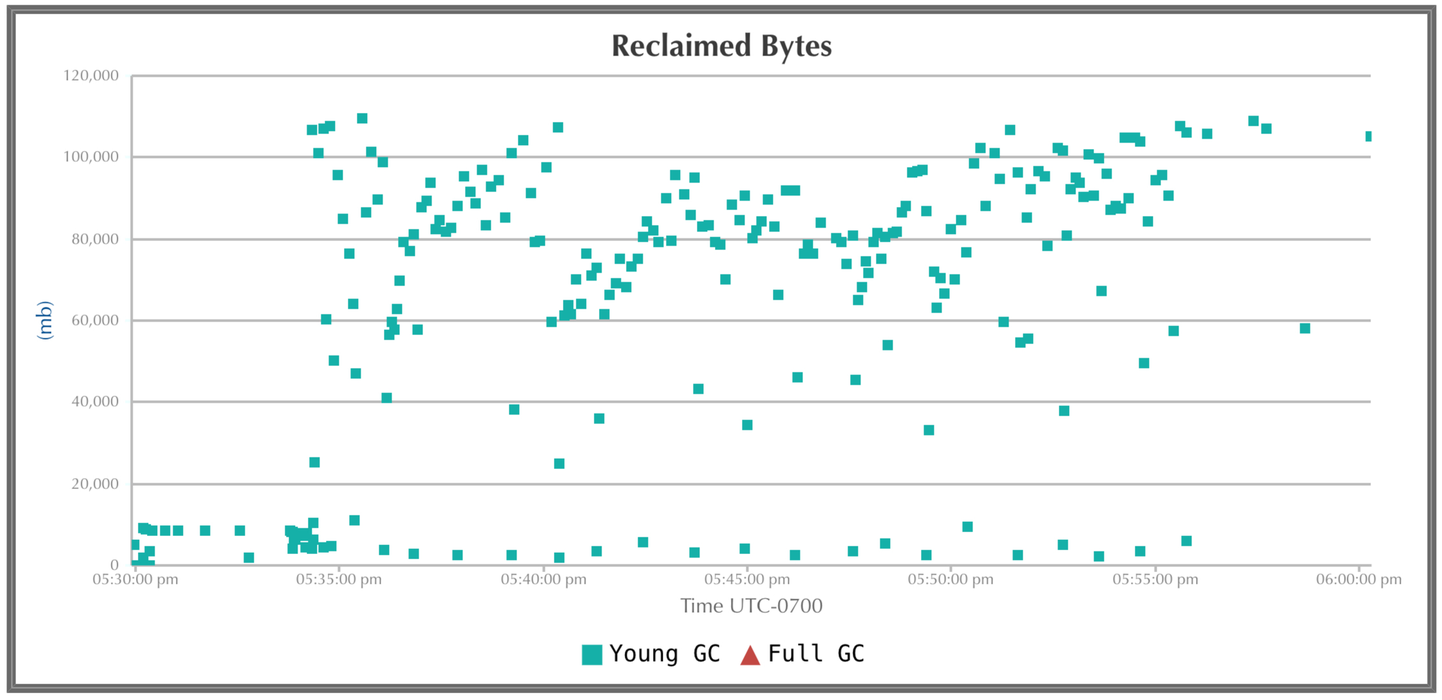
Interactive Graphs (How to zoom graphs?)

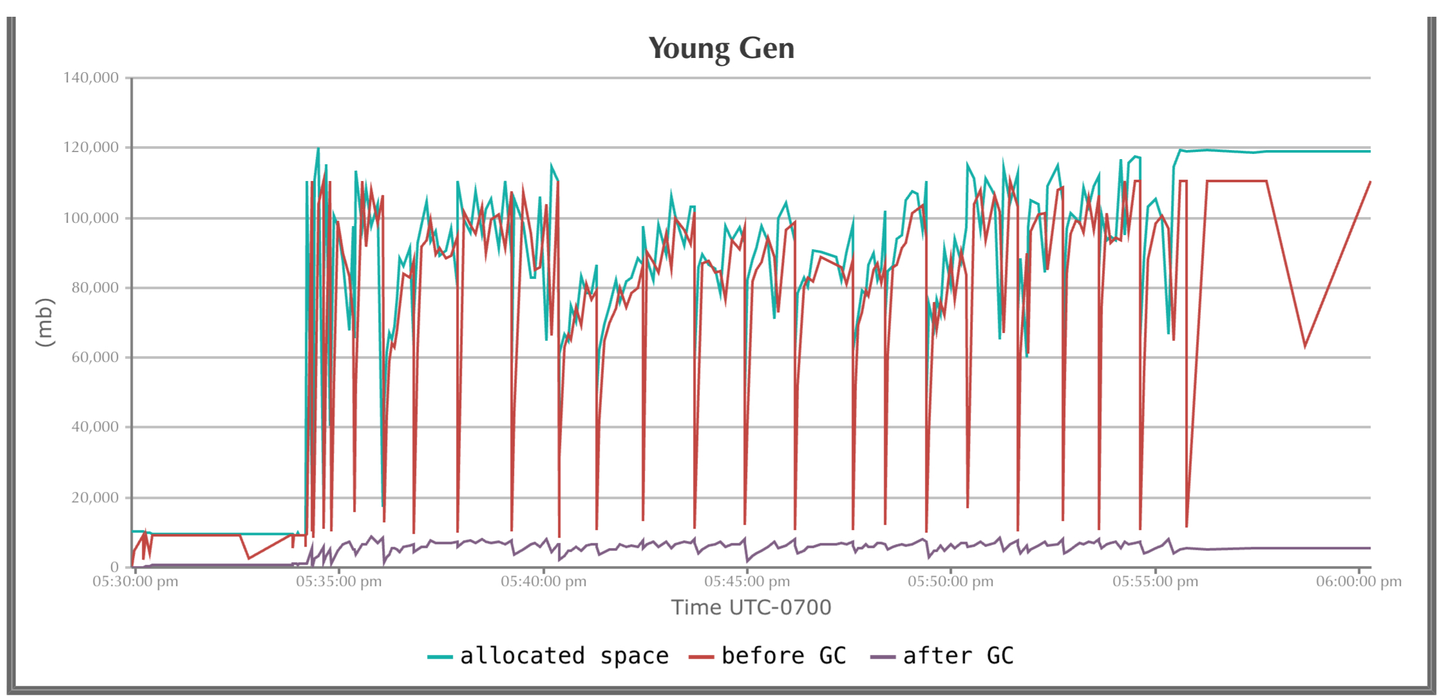


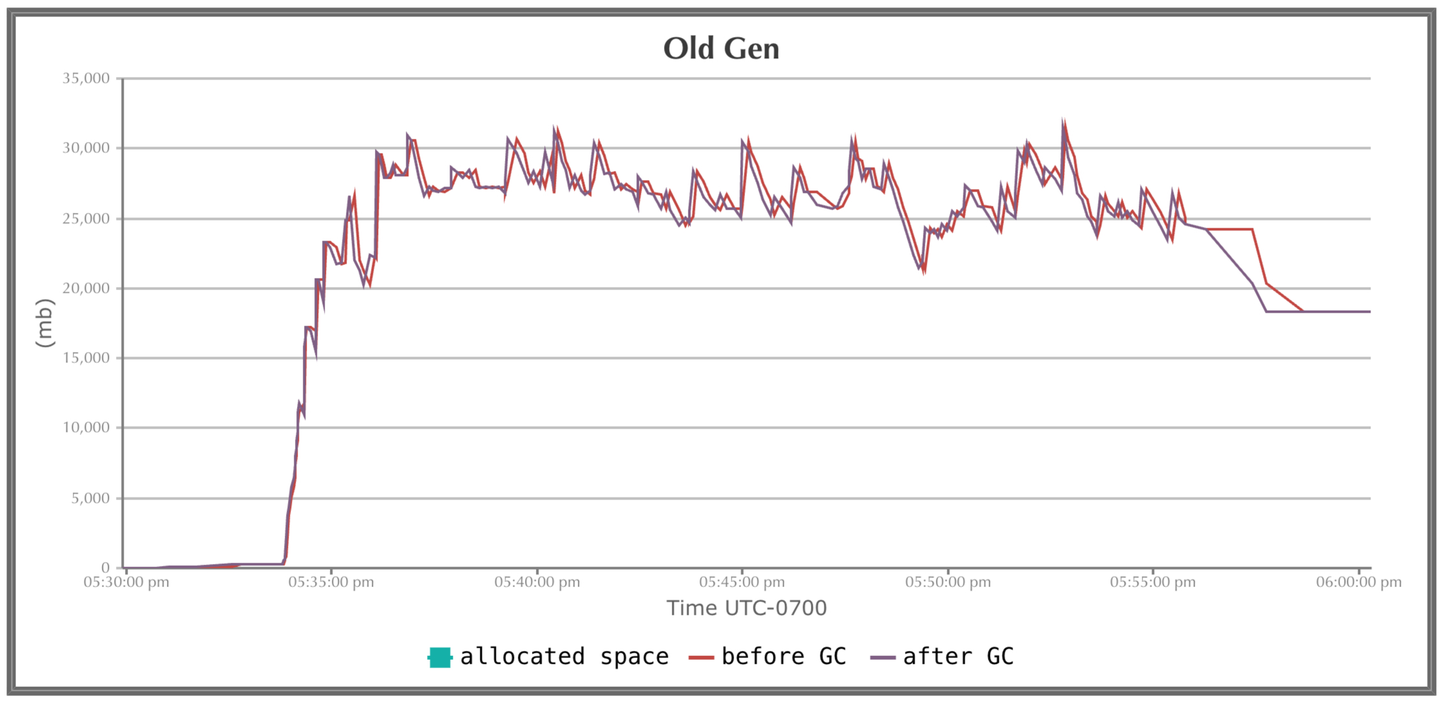

Heap Usage (before GC)
160,000
140,000
120,000
100,000
80,000
60,000
40,000
20,000
0
Heap size (mb)
05:30:00 pm
05:35:00 pm
05:40:00 pm
05:45:00 pm
05:50:00 pm
05:55:00 pm
06:00:00 pm
Time UTC-0700

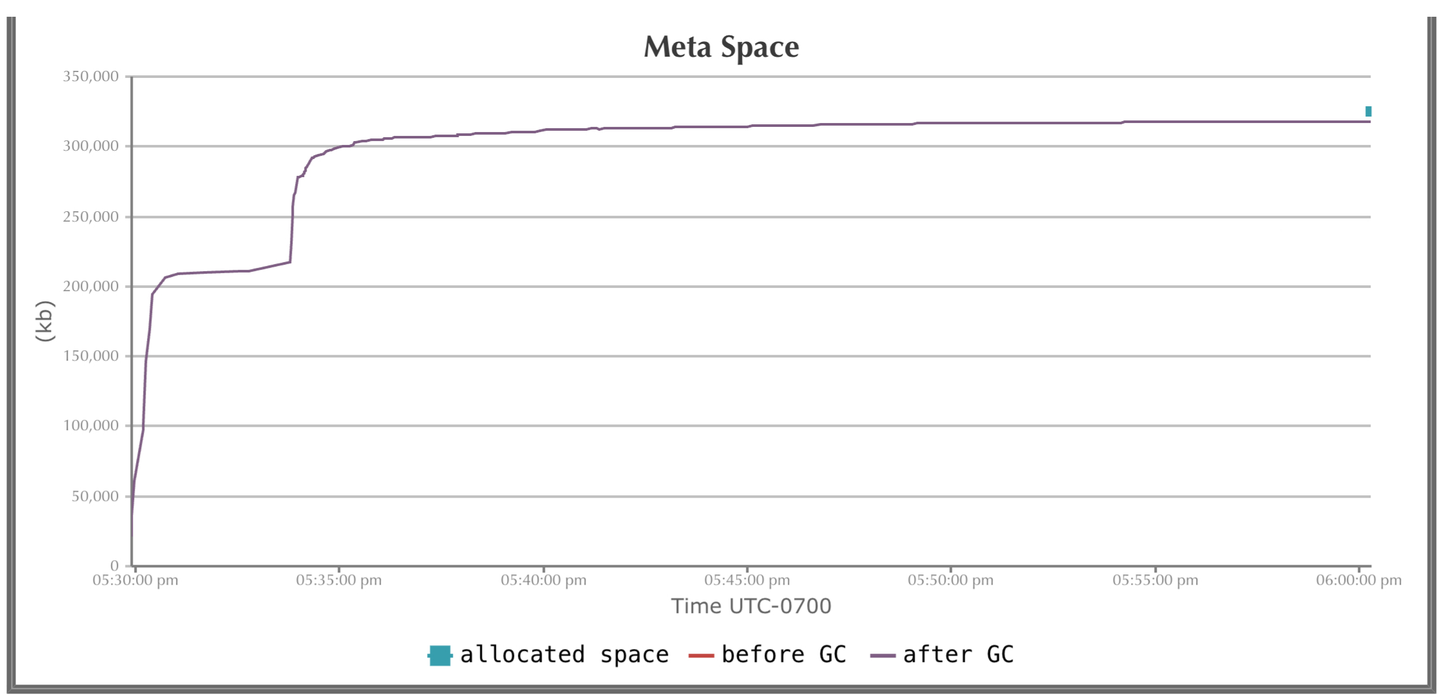

GC Duration Time
1,400
1,200
1,000
800
600
400
200
0
Time (ms)
05:30:00 pm
05:35:00 pm
05:40:00 pm
05:45:00 pm
05:50:00 pm
05:55:00 pm
06:00:00 pm
Time UTC-0700
Young GC
Full GC

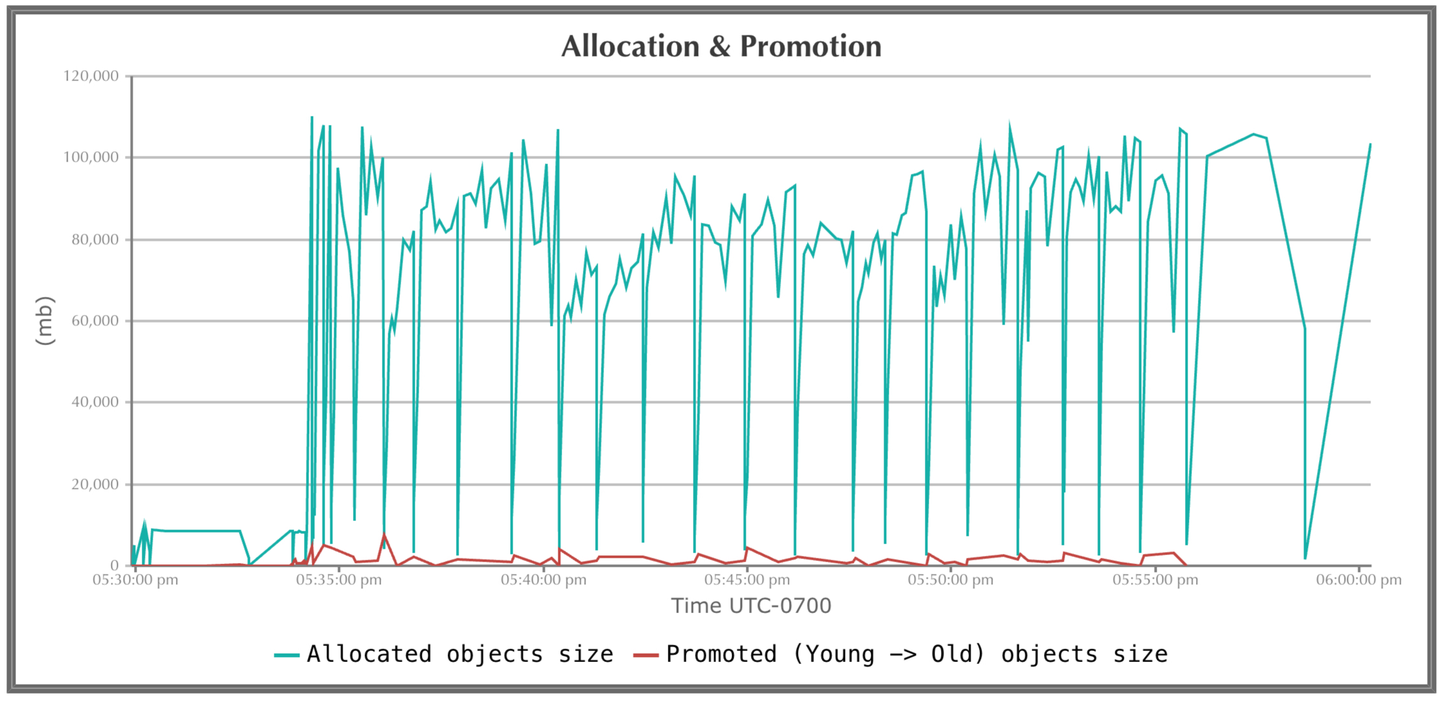

Pause GC Duration Time
800
700
600
500
400
300
200
100
0
Time (ms)
05:30:00 pm
05:35:00 pm
05:40:00 pm
05:45:00 pm
05:50:00 pm
05:55:00 pm
06:00:00 pm
Time UTC-0700
Young GC
Full GC


Reclaimed Bytes
120,000
100,000
80,000
60,000
40,000
20,000
0
(mb)
05:30:00 pm
05:35:00 pm
05:40:00 pm
05:45:00 pm
05:50:00 pm
05:55:00 pm
06:00:00 pm
Time UTC-0700
Young GC
Full GC


Young Gen
140,000
120,000
100,000
80,000
60,000
40,000
20,000
0
(mb)
05:30:00 pm
05:35:00 pm
05:40:00 pm
05:45:00 pm
05:50:00 pm
05:55:00 pm
06:00:00 pm
Time UTC-0700
allocated space
before GC
after GC


Old Gen
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
(mb)
05:30:00 pm
05:35:00 pm
05:40:00 pm
05:45:00 pm
05:50:00 pm
05:55:00 pm
06:00:00 pm
Time UTC-0700
allocated space
before GC
after GC

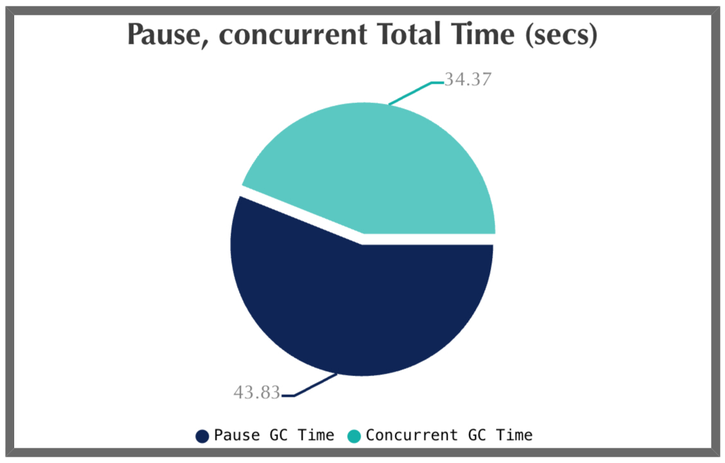


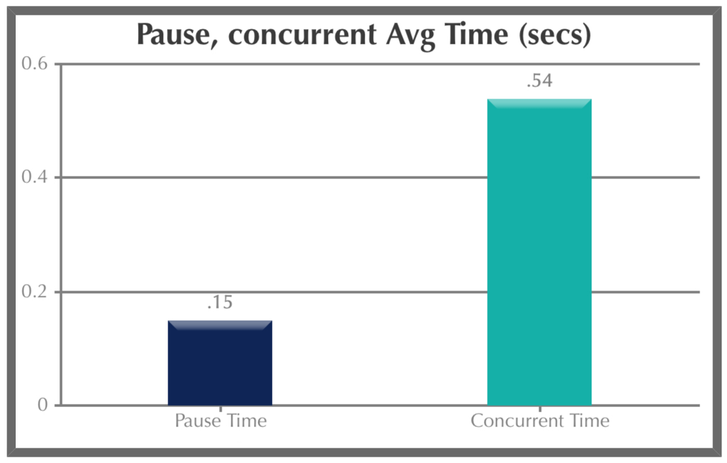


G1 Collection Phases Statistics

	Young GC	Concurrent Marking	Remark	Root Region Scanning	Cleanup
Total Time	45 sec 819 ms	28 sec 312 ms	7 sec 119 ms	2 sec 24 ms	47.1 ms
Avg Time	170 ms	885 ms	222 ms	63.3 ms	1.47 ms
Std Dev Time	165 ms	489 ms	167 ms	36.6 ms	0.748 ms
Min Time	0	34.6 ms	2.00 ms	2.28 ms	0.00800 ms
Max Time	1 sec 354 ms	1 sec 694 ms	710 ms	126 ms	2.11 ms
Interval Time	6 sec 803 ms	55 sec 659 ms	55 sec 659 ms	55 sec 659 ms	55 sec 659 ms
Count	269	32	32	32	32

G1 GC Time





Pause Time

Total GC Pause Time	43 sec 832 ms
GC Pause Events	301.0
Avg GC Pause Time	146 ms
Std Dev GC Pause Time	83.5 ms
Min GC Pause Time	0.00800 ms
Max GC Pause Time	710 ms

Concurrent Time

Total Concurrent GC Time	34 sec 366 ms
Concurrent GC Events	64.0
Avg Concurrent GC Time	537 ms
Std Dev Concurrent GC Time	551 ms
Min Concurrent GC Time	10.0 ms
Max Concurrent GC Time	1 sec 794 ms

Object Stats

Total created bytes	14.24 tb
Total promoted bytes	105.94 gb
Avg creation rate	5.42 gb/sec

CPU Stats

(To learn more about CPU stats, click here)

CPU Time:	51 min 57 sec 970 ms
User Time:	51 min 8 sec 400 ms
Sys Time:	49 sec 570 ms

Avg promotion rate	40.37 mb/sec

Memory Leak

No major memory leaks.
(Note: there are 8 flavours of OutOfMemoryErrors. 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)

	Total Time	Avg Time	% of total duration
Total time for which app threads were stopped	47.728 secs	0.016 secs	1.776 %
Time taken to stop app threads	6.108 secs	0.002 secs	0.227 %

Threads Affected By Allocation Stalls

(To learn more about Allocation Stall, click here)

Not Reported in the log.

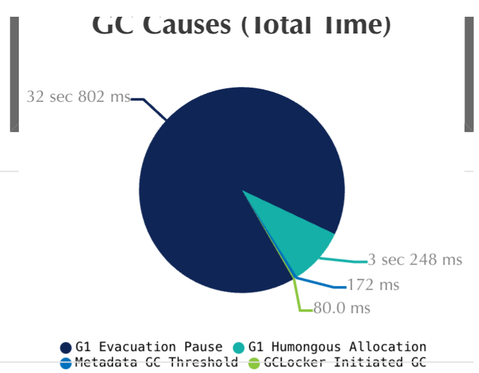
String Deduplication Metrics

Not Reported in the log.

GC Causes

(What events caused GCs & how much time they consumed?)

Cause	Count	Avg Time	Max Time	Total Time
G1 Evacuation Pause	205	160 ms	260 ms	32 sec 802 ms
G1 Humongous Allocation	22	148 ms	176 ms	3 sec 248 ms
Metadata GC Threshold	5	34.4 ms	80.3 ms	172 ms
GCLocker Initiated GC	1	80.0 ms	80.0 ms	80.0 ms



Tenuring Summary

Not reported in the log.

JVM Arguments

(To learn about JVM Arguments, click here)

Not reported in the log.

Export Data

Normalized GC Data