GC Intelligence Report

gc-virtual-threads.log

Duration: 2 min 12 sec 338 ms

Recommendations

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

- **585 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%.

- **20.0 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.**

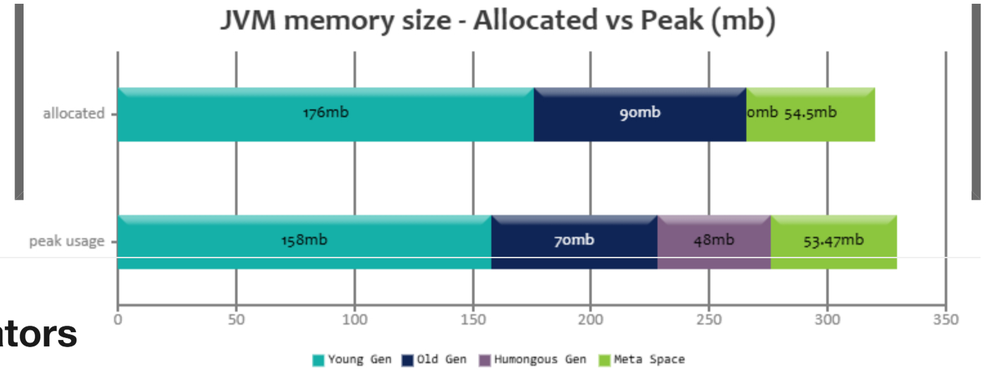
- This application is using the G1 GC algorithm. If you are looking to tune G1 GC performance even further, here are the **important G1 GC algorithm related JVM arguments**

JVM memory size

(To learn about JVM Memory, click here)

Generation	Allocated	Peak
Young Generation	176 mb	158 mb
Old Generation	90 mb	70 mb

Humongous	n/a	48 mb
Meta Space	54.5 mb	53.47 mb
Young + Old + Meta space	320.5 mb	295.47 mb



Key Performance Indicators

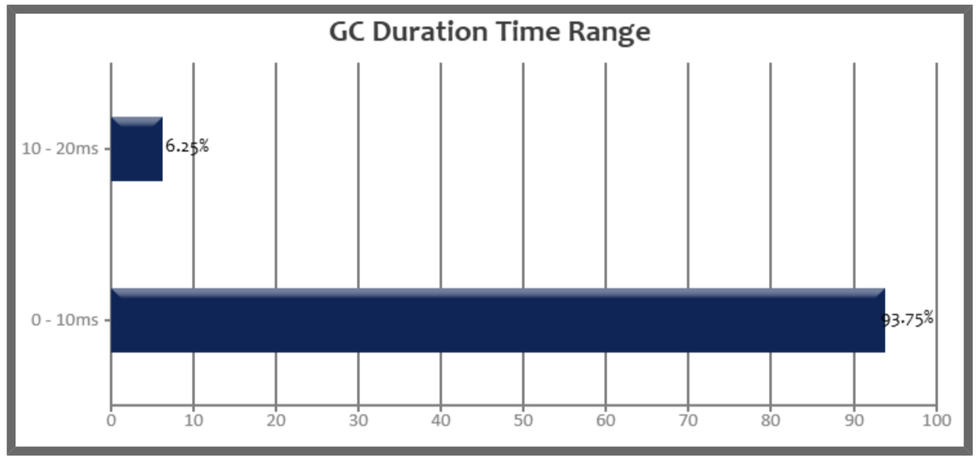
(Important section of the report. To learn more about KPIs, click here)

1. **Throughput : 99.424%**
2. **CPU Time : 850 ms**
3. **Latency:**

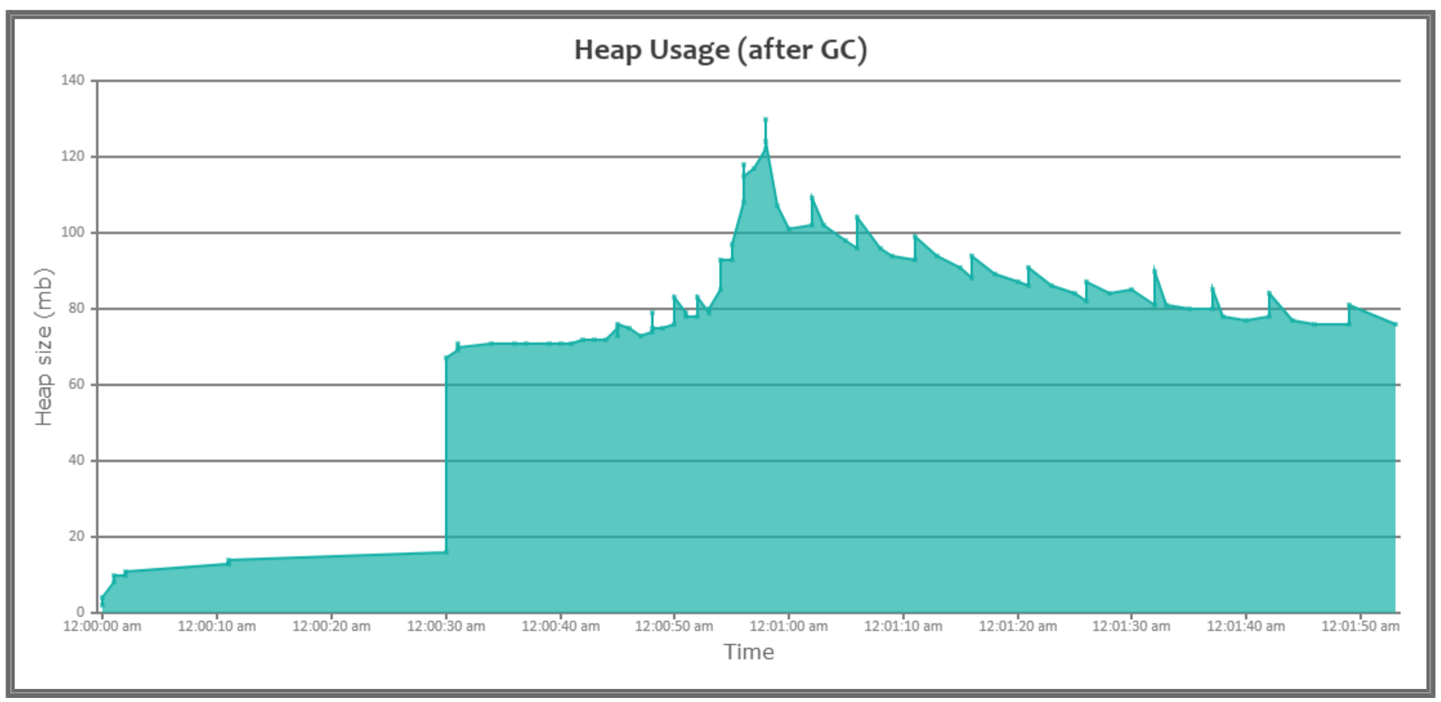
Avg Pause GC Time	**6.80 ms**
Max Pause GC Time	**20.0 ms**

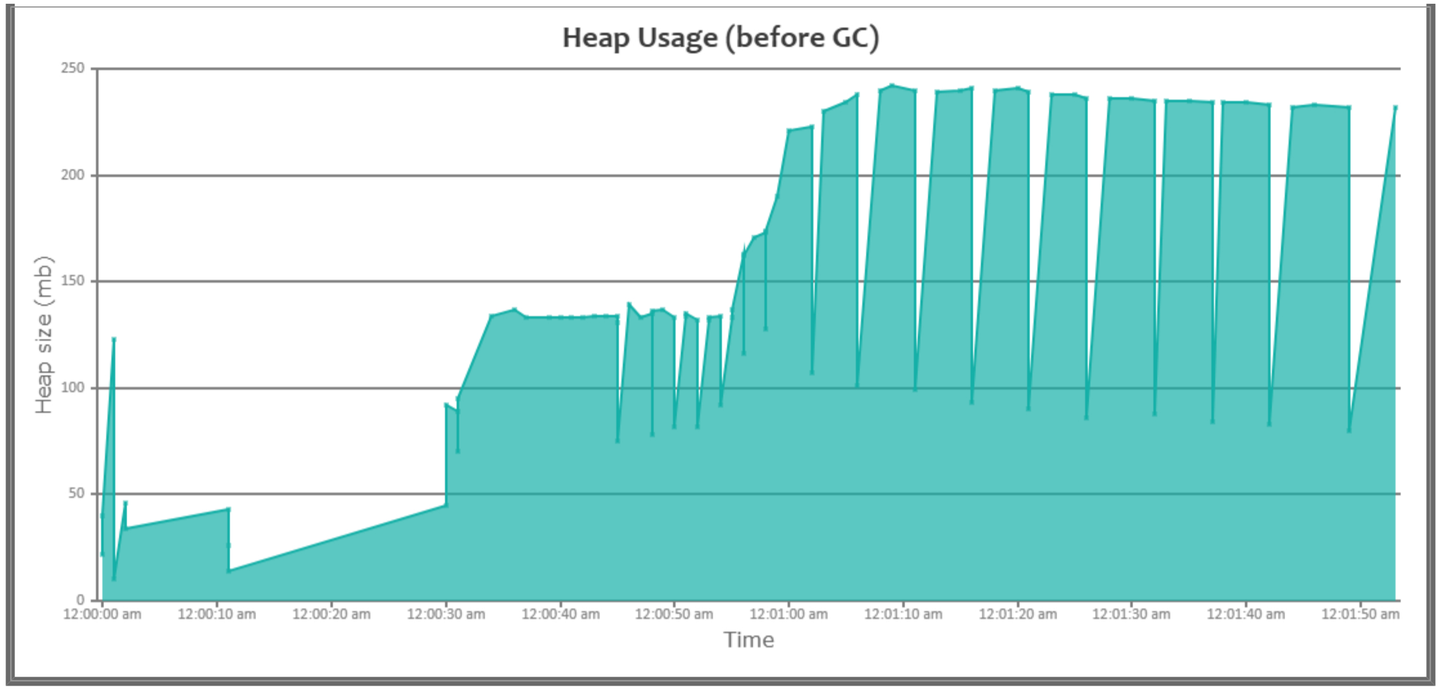
GC Pause Duration Time Range :

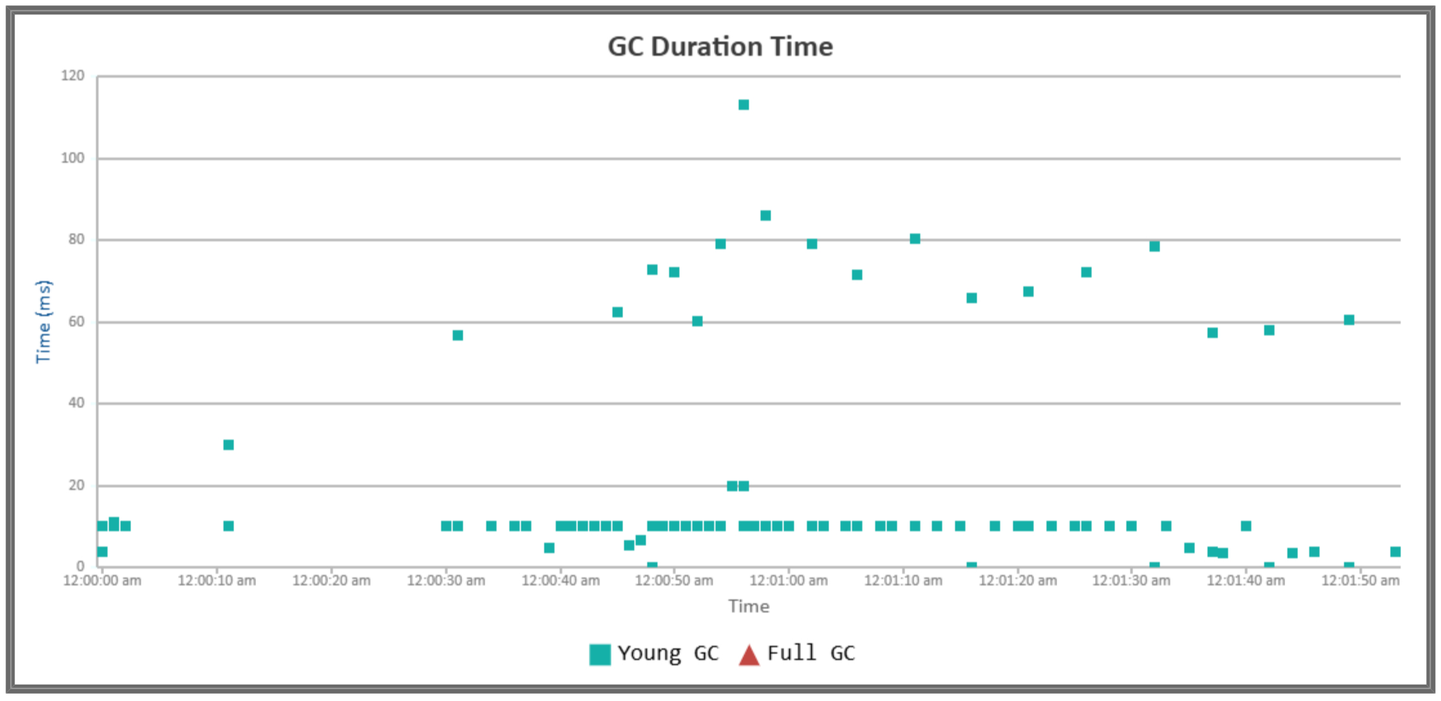
Duration (ms) 10 ms Change	No. of GCs	Percentage
0 - 10	105	93.75%
10 - 20	7	6.25%

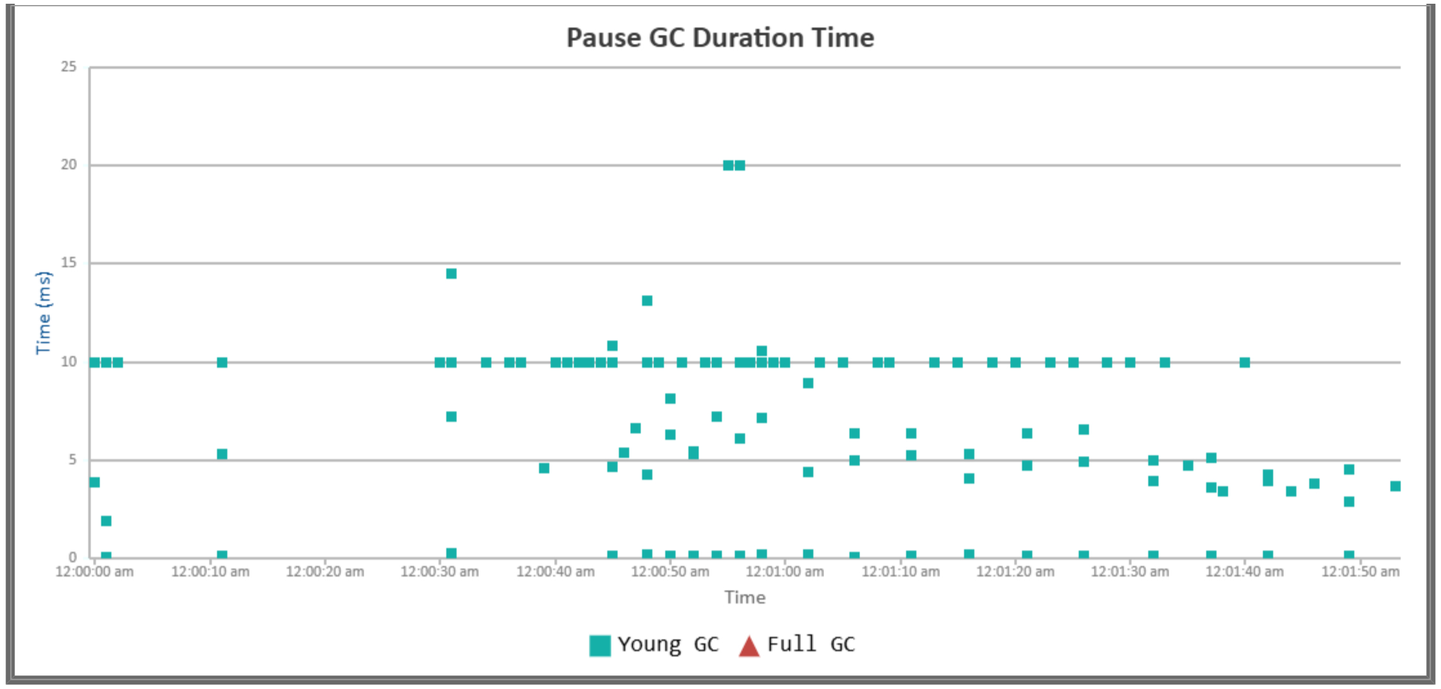


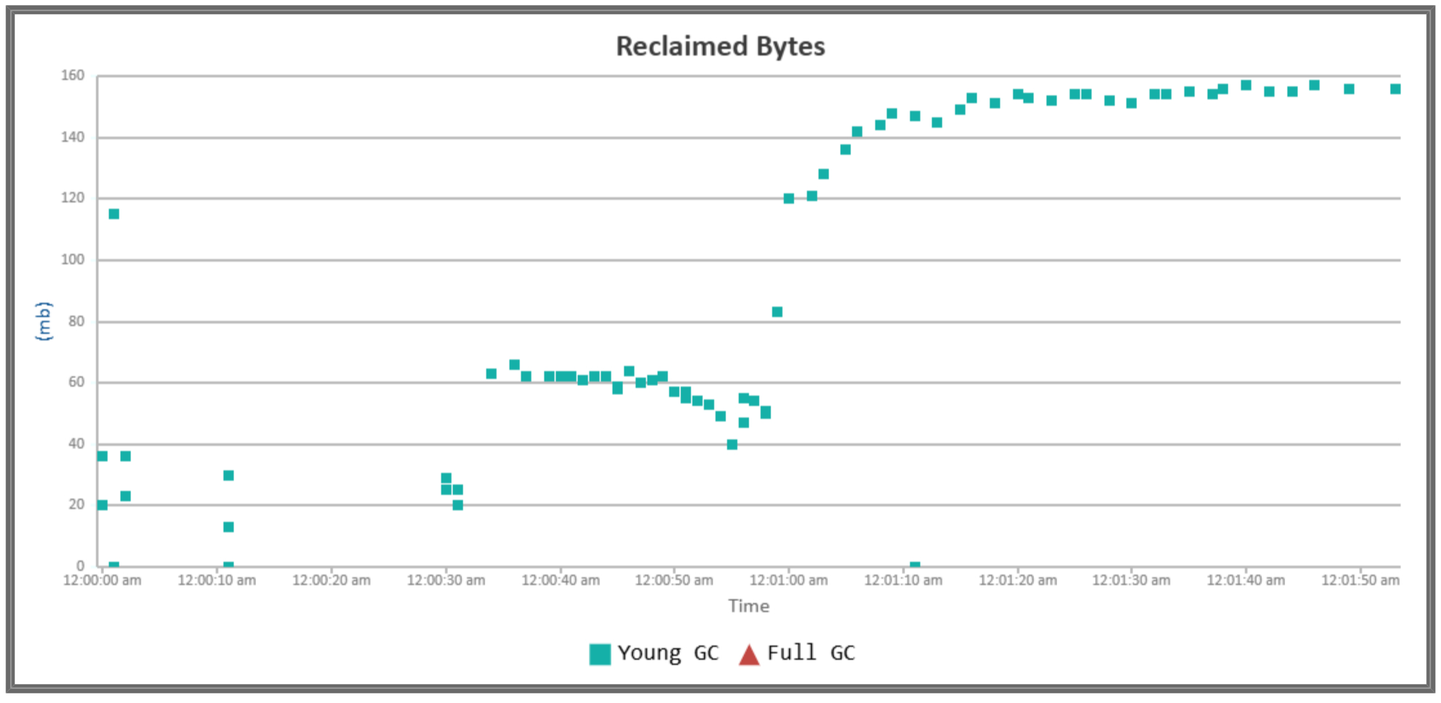
Interactive Graphs (How to zoom graphs?)

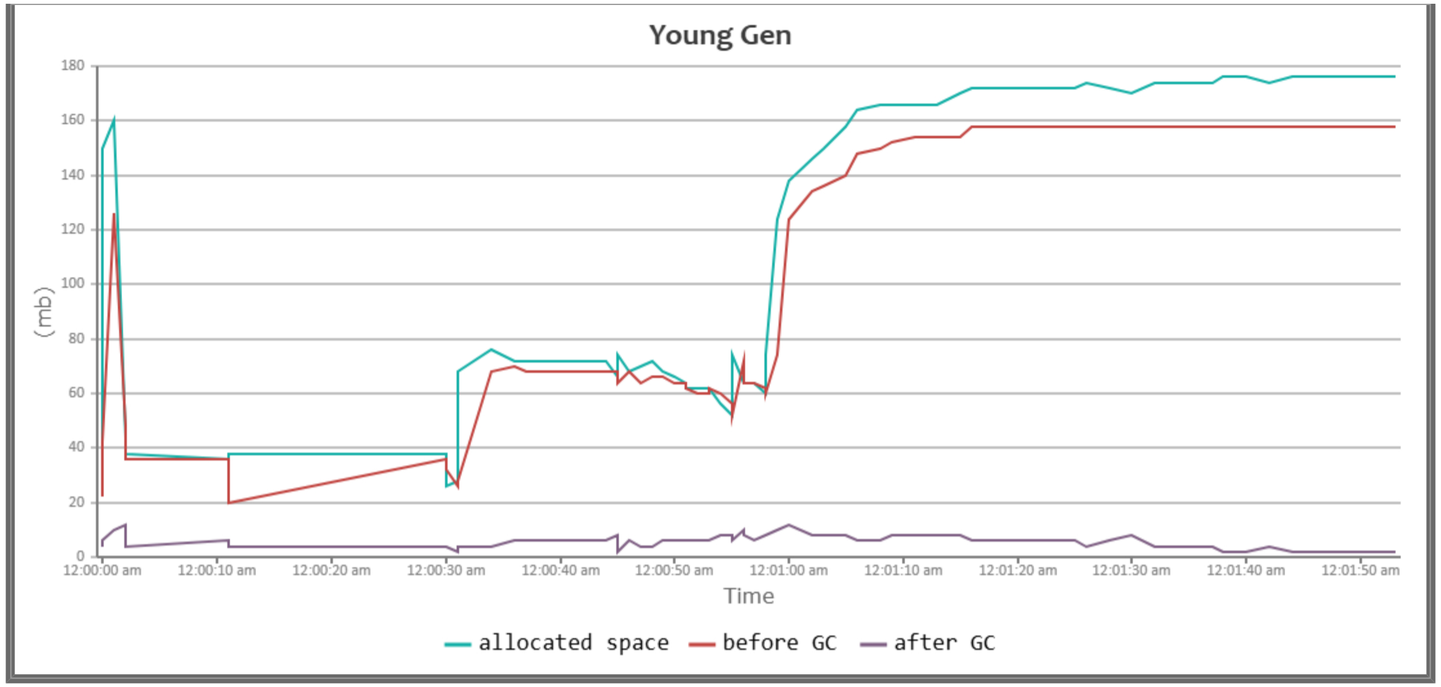


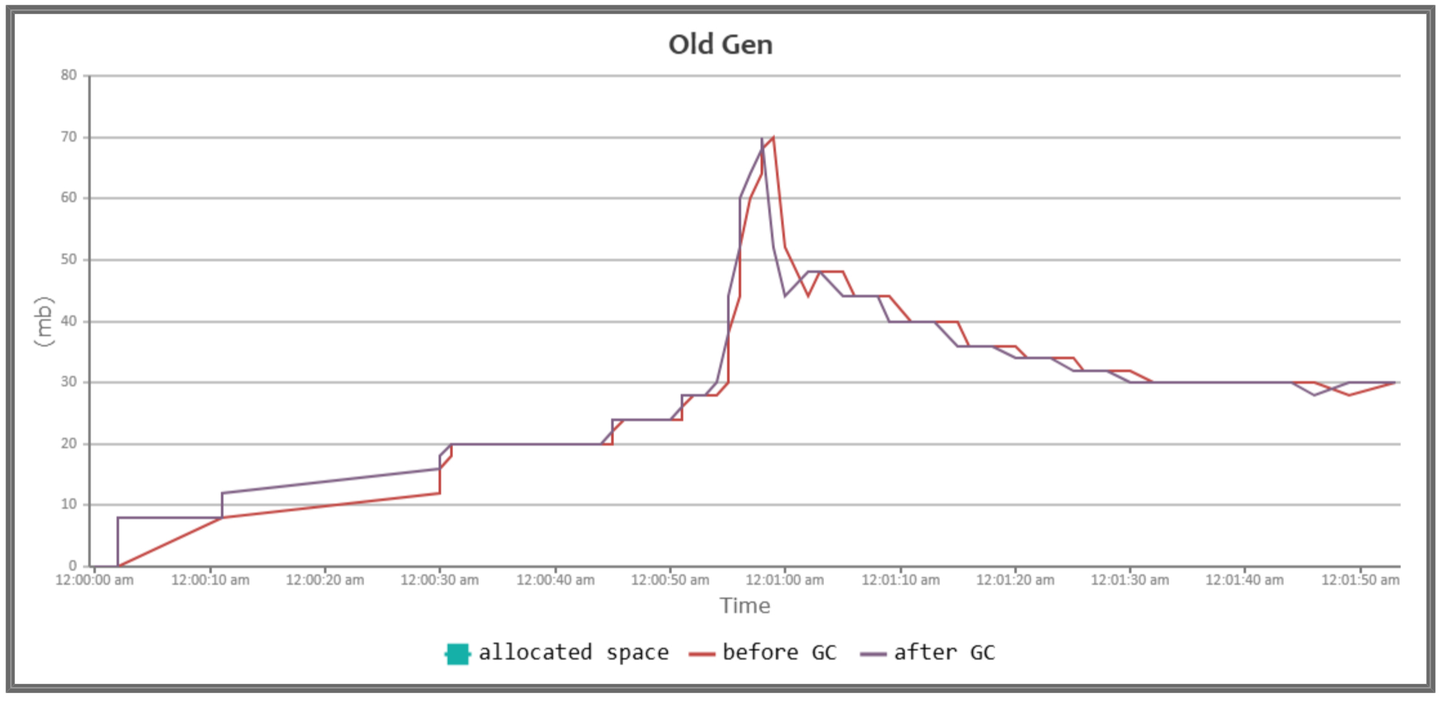

Heap Usage (before GC)
Heap size (mb)
250
200
150
100
50
0
12:00:00 am
12:00:10 am
12:00:20 am
12:00:30 am
12:00:40 am
12:00:50 am
12:01:00 am
12:01:10 am
12:01:20 am
12:01:30 am
12:01:40 am
12:01:50 am
Time

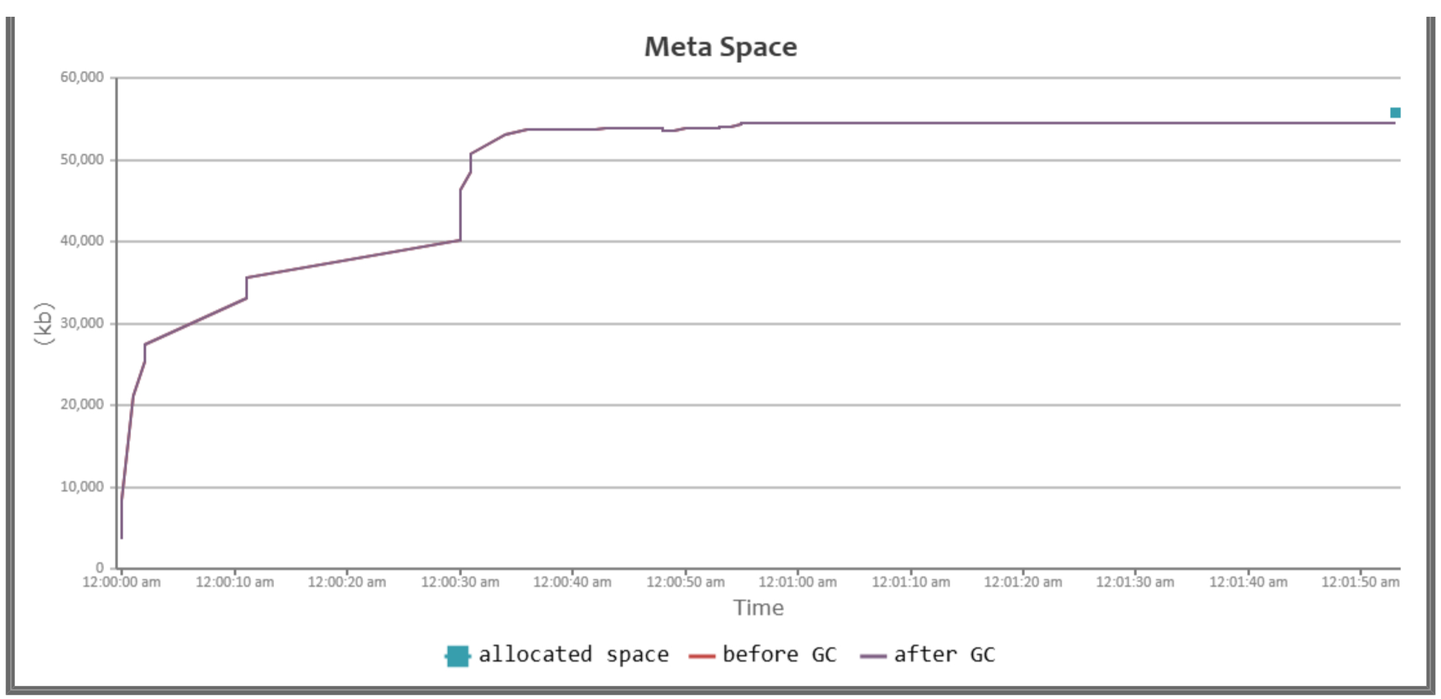

GC Duration Time
Time (ms)
120
100
80
60
40
20
0
12:00:00 am
12:00:10 am
12:00:20 am
12:00:30 am
12:00:40 am
12:00:50 am
12:01:00 am
12:01:10 am
12:01:20 am
12:01:30 am
12:01:40 am
12:01:50 am
Time
Young GC
Full GC

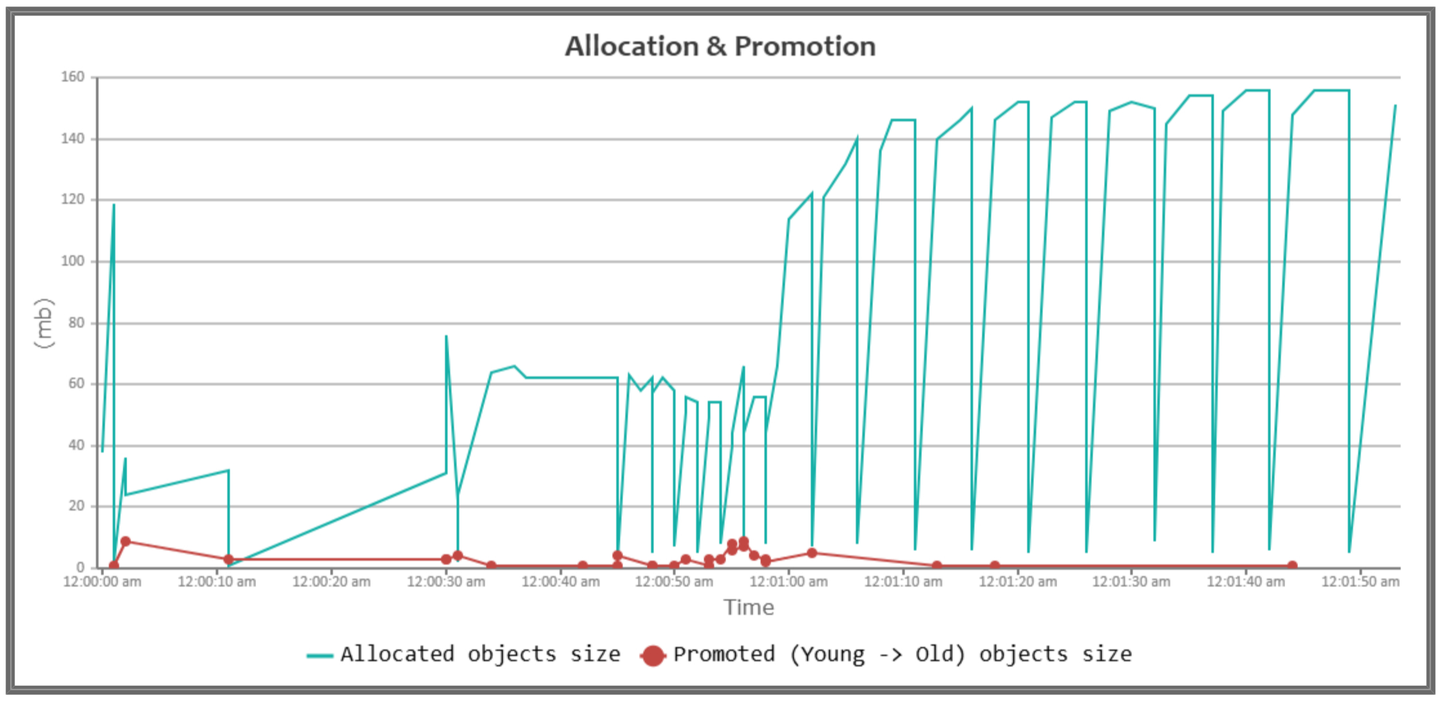

Pause GC Duration Time
Time (ms)
25
20
15
10
5
0
12:00:00 am
12:00:10 am
12:00:20 am
12:00:30 am
12:00:40 am
12:00:50 am
12:01:00 am
12:01:10 am
12:01:20 am
12:01:30 am
12:01:40 am
12:01:50 am
Time
Young GC
Full GC


Reclaimed Bytes
(mb)
160
140
120
100
80
60
40
20
0
12:00:00 am
12:00:10 am
12:00:20 am
12:00:30 am
12:00:40 am
12:00:50 am
12:01:00 am
12:01:10 am
12:01:20 am
12:01:30 am
12:01:40 am
12:01:50 am
Time
Young GC
Full GC


Young Gen
180
160
140
120
100
80
60
40
20
0
(mb)
12:00:00 am
12:00:10 am
12:00:20 am
12:00:30 am
12:00:40 am
12:00:50 am
12:01:00 am
12:01:10 am
12:01:20 am
12:01:30 am
12:01:40 am
12:01:50 am
Time
allocated space
before GC
after GC


Old Gen
80
70
60
50
40
30
20
10
0
(mb)
12:00:00 am
12:00:10 am
12:00:20 am
12:00:30 am
12:00:40 am
12:00:50 am
12:01:00 am
12:01:10 am
12:01:20 am
12:01:30 am
12:01:40 am
12:01:50 am
Time
allocated space
before GC
after GC

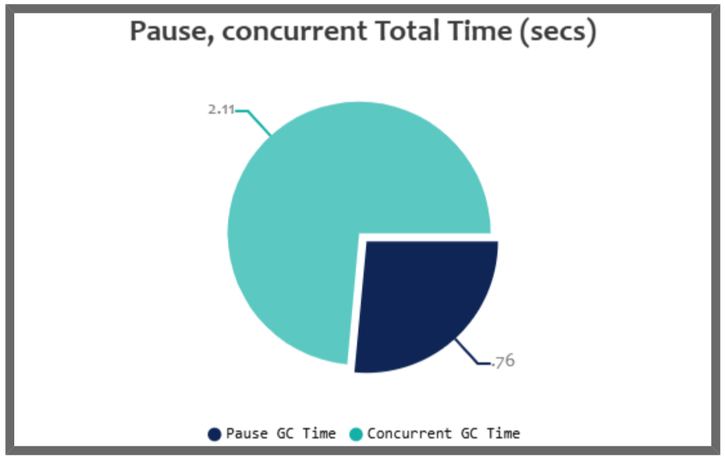


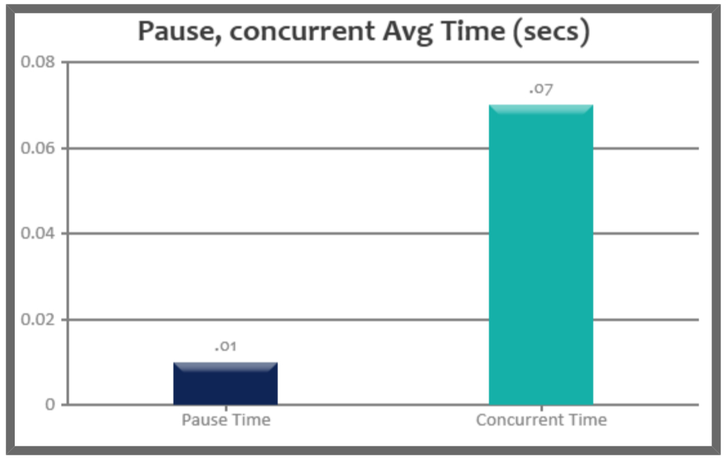


G1 Collection Phases Statistics

	Young GC	Concurrent Marking	Root Region Scanning	Remark	Cleanup
Total Time	1 sec 977 ms	1 sec 848 ms	162 ms	124 ms	2.78 ms
Avg Time	21.5 ms	92.4 ms	8.08 ms	6.20 ms	0.139 ms
Std Dev Time	25.8 ms	28.6 ms	4.55 ms	3.05 ms	0.0419 ms
Min Time	0	3.00 ms	2.53 ms	1.92 ms	0.0530 ms
Max Time	113 ms	151 ms	17.7 ms	14.5 ms	0.264 ms
Interval Time	1 sec 247 ms	5 sec 673 ms	5 sec 673 ms	5 sec 673 ms	5 sec 673 ms
Count	92	20	20	20	20

G1 GC Time





Pause Time

Total Time	762 ms
Avg Time	6.80 ms
Std Dev Time	4.42 ms
Min Time	0.0530 ms
Max Time	20.0 ms

Concurrent Time

Total Time	2 sec 114 ms
Avg Time	70.5 ms
Std Dev Time	49.6 ms
Min Time	10.0 ms
Max Time	169 ms

Object Stats

Total created bytes	6.54 gb
Total promoted bytes	90 mb
Avg creation rate	50.6 mb/sec

CPU Stats

(To learn more about CPU stats, click here)

CPU Time:	850 ms
User Time:	790 ms
Sys Time:	60.0 ms

Avg promotion rate	696 kb/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)

Not Reported in the log.

Allocation stall metrics

(To learn more about Allocation Stall, click here)

Not Reported in the log.

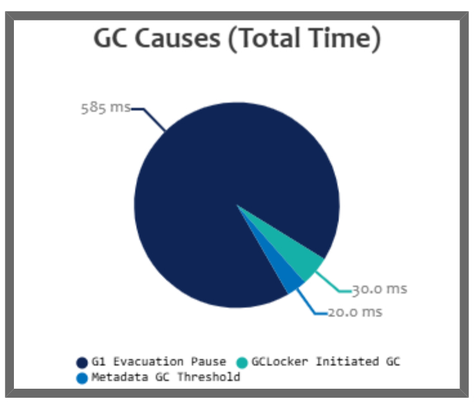
String Deduplication Metrics

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	67	8.73 ms	20.0 ms	585 ms
GCLocker Initiated GC	3	10.0 ms	10.0 ms	30.0 ms
Metadata GC Threshold	2	10.0 ms	10.0 ms	20.0 ms



Tenuring Summary

Not reported in the log.

JVM Arguments

(To learn about JVM Arguments, click here)

Not reported in the log.