Arun Kejariwal

CS229
Project Report

Abstract
Elasticity of cloud assists in achieving availability of web applications by scaling up a cluster as incoming traffic increases and keep operational costs under check by scaling down a cluster when incoming traffic decreases. The elasticity of cloud makes performance modeling and analysis non-trivial. In the context of Java-based web applications, a key aspect is the performance of the garbage collector (GC). Existing tools for analyzing the performance of a GC are tailored for a single Java process, hence not suitable for comparative garbage collector performance analysis across all the nodes in a cluster in the cloud. The variation in GC performance stems from a multitude of sources, for example (but not limited to) multi-tenancy, non-uniform arrival traffic and other systemic factors. The objective of the project is to employ unsupervised learning techniques to detect bad nodes (where, for example, Full GC time on a bad node would be much higher than the healthy nodes).

Introduction
Netflix migrated its infrastructure from a traditional data center to AWS EC2 to leverage, amongst other benefits, elasticity and support for multiple availability zones. Migration to the cloud eliminated the cycles spent on hardware procurement, datacenter maintenance and resulted in higher development agility. Having said that, the elasticity of AWS – a cluster can be scaled up or down – makes performance characterization and tuning of, in the current context, Java-based applications in the cloud non-trivial. Tuning the Java HotSpot virtual machine garbage collection (GC) is key to performance optimization of web applications. For instance, Zhao et al. (OOPSLA'09) demonstrated that GC has material impact on the performance and scalability of Java applications on multi-core platforms. At Netflix, and in general, the following factors, in part, mandate the need for a tool for advanced analysis and tuning GC performance in the cloud.

- **Performance evaluation of a new JVM:** Recently, Java 7 was announced. Several new optimizations such as Compressed Oops, Escape Analysis are supported in Java 7. The new optimizations can potentially impact GC performance. For example, escape analysis can result in elimination of certain object allocations. Figure 1 shows the time series of heap usage, in JConsole, of one of the Netflix applications. The time series does not throw light on the distribution of heap usage that is needed to characterize the garbage collector performance.

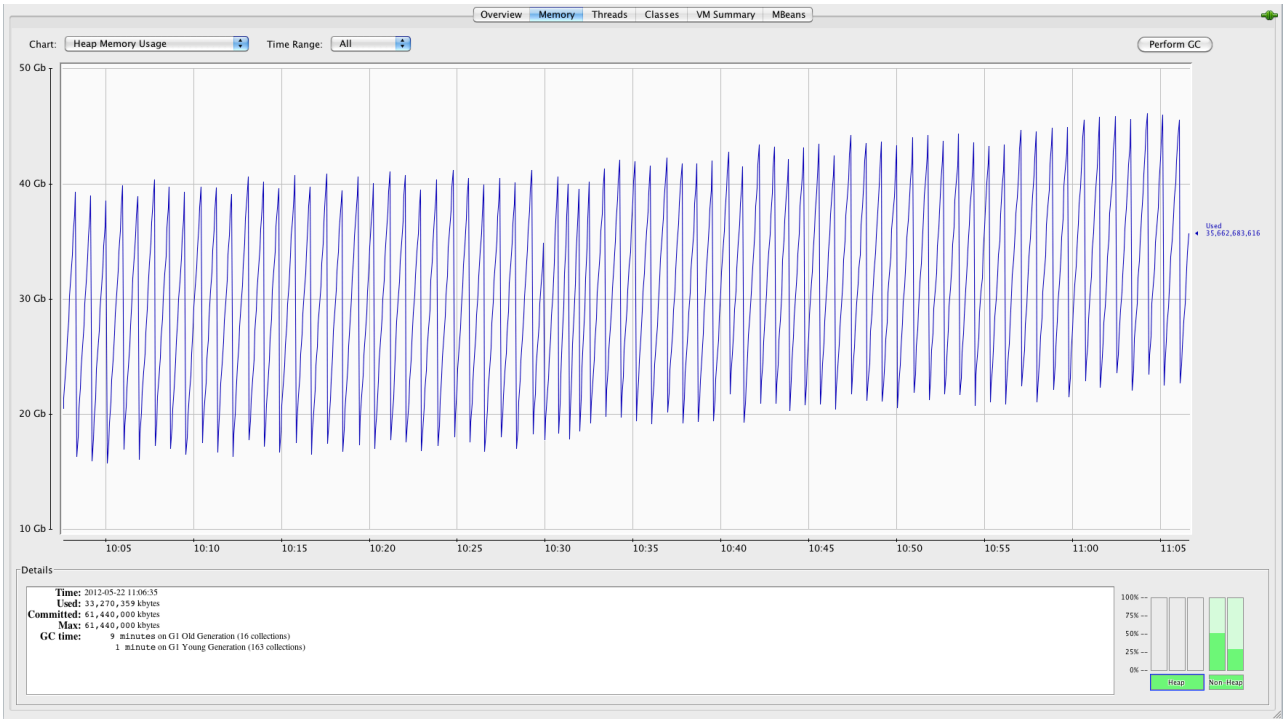

Figure 1

- **Application-driven selection of garbage collector type:** Compare the performance of the G1 GC, first introduced in Java 1.6.0 14 and the concurrent-mark-and-sweep CMS (default in Java 7) GC.

The focus of this project is to detect outlier – with respect to GC performance – nodes in a given cluster.

Brief Background

In this section, we present an overview of the JVM heap layout and the associated terminology. Figure 2 depicts the layout of the HotSpot VM heap. From the Figure we note that Java heap consists of three spaces: young generation (YG), old generation (OG) and the permanent generation.

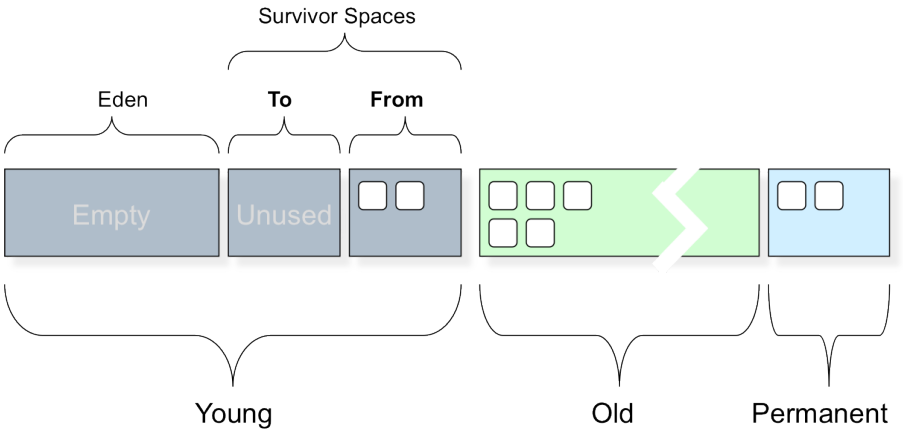

Figure 2

In Java, objects are first allocated to the young generation space. Objects are promoted to the old generation whose age – the age of an object is the number of minor GCs it has survived – is more than the internally determined (by the VM) threshold.[1] VM, Java class metadata and interned Strings and class static variables are stored in the permanent generation.

Input Data

Figure 3 presents a bar plot of the size of GC logs for the various nodes (34 in total) in the cluster chosen for analysis. Note that the chosen cluster corresponds to a production cluster for a Netflix application.

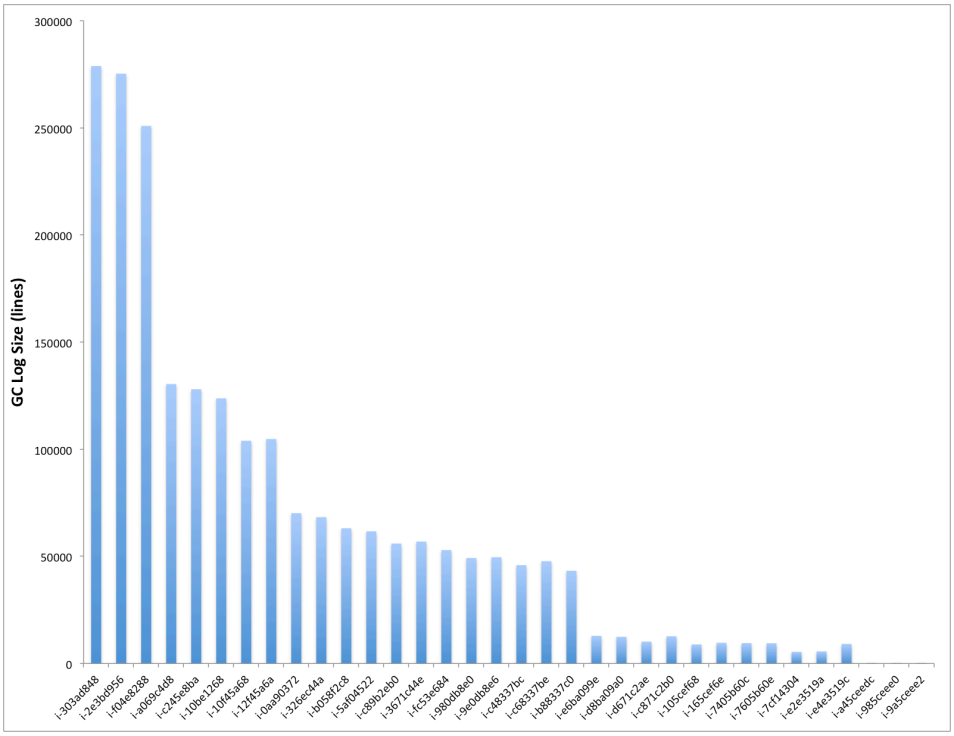

Figure 3

[1] The HotSpot VM command line option, -XX:MaxTenuringThreshold=<n>, can be used to set the threshold for tenuring objects to the old generation.

The GC log corresponding to node i-985ceee0 has only 8 lines as the node came online close to when the data was captured. Computation of, say, standard deviation promotion size for the node results in an invalid value. Hence, the node is excluded from outlier detection analysis.

Data Pre-processing

The format of a GC log varies from one type of GC to another. Further, given a GC type, the format of the GC log varies based on the JVM flags specified at run time. The Netflix application chosen uses CMS (concurrent-mark-sweep) GC. The GC Log Analysis Engine developed for this project supports various GC log formats corresponding to the different JVM options. The following shows an excerpt of a GC log from one of the nodes. Lines highlighted in blue corresponding to GC events at different time stamps.

Example GC log excerpt

```
2012-07-08T16:17:22.101+0000: 81.638: [GC 81.638: [ParNew
Desired survivor size 214728704 bytes, new threshold 1 (max 3)
- age   1:  322981576 bytes,  322981576 total
- age   2:   38071600 bytes,  361053176 total
- age   3:   38318016 bytes,  399371192 total
: 3619504K->419392K(3774912K), 0.8012030 secs] 3619504K->493482K(26843584K), 0.8013840 secs]
[Times: user=2.73 sys=0.24, real=0.80 secs]

2012-07-08T16:17:40.956+0000: 100.493: [GC 100.493: [ParNew
Desired survivor size 214728704 bytes, new threshold 1 (max 3)
- age   1:  415393472 bytes,  415393472 total
: 3774912K->419392K(3774912K), 1.3058180 secs] 3849002K->922326K(26843584K), 1.3060310 secs]
[Times: user=4.49 sys=0.38, real=1.30 secs]

2012-07-08T16:17:46.956+0000: 106.494: [GC 106.494: [ParNew
Desired survivor size 214728704 bytes, new threshold 1 (max 3)
- age   1:  429456064 bytes,  429456064 total
: 3496225K->419392K(3774912K), 1.9057210 secs] 3999160K->1694379K(26843584K), 1.9059240 secs]
[Times: user=5.93 sys=0.48, real=1.90 secs]

2012-07-08T16:17:48.879+0000: 108.416: [GC [1 CMS-initial-mark: 1274987K(23068672K)]
1768525K(26843584K), 0.2872400 secs] [Times: user=0.28 sys=0.01, real=0.29 secs]

2012-07-08T16:17:49.167+0000: 108.704: [CMS-concurrent-mark-start]

2012-07-08T16:17:51.163+0000: 110.700: [CMS-concurrent-mark: 1.985/1.997 secs] [Times:
user=4.22 sys=0.15, real=1.99 secs]

2012-07-08T16:17:51.163+0000: 110.701: [CMS-concurrent-preclean-start]

2012-07-08T16:17:51.264+0000: 110.801: [CMS-concurrent-preclean: 0.099/0.100 secs] [Times:
user=0.20 sys=0.00, real=0.10 secs]

2012-07-08T16:17:51.264+0000: 110.801: [CMS-concurrent-abortable-preclean-start]
```

The numbers before and after the arrow of `3619504K->419392K` indicate the size of the young generation before and after garbage collection, respectively. After minor collections the size includes some objects that are garbage (no longer alive) but that cannot be reclaimed. These objects are either contained in the tenured generation, or referenced from the tenured or permanent generations. The numbers before and after the arrow of `3619504K->493482K` indicate the size of the heap before and after garbage collection.

References mentioned in the last section throw further light on the how to read the other details of a CMS GC log.

Mining GC Logs

Time Series Analysis

Owing to agile product development (which was one of the factors to move to the cloud from the datacenter), new applications and features are pushed to the cloud with high frequency. This, at times, results in introduction of performance regressions in GC performance. One of the ways to detect the time and extent of such regression is analysis

of time series of the various GC metrics such as young generation usage over time. Multi-tenancy and hardware failure[2] can lead to a degradation in performance to end-users due to service unavailability and can potentially result in losses to the business – in both revenue as well as long term reputation. Thus, detecting "bad" nodes in a cluster, ASG in the current context, is key to maintain low latency and high throughput. Comparative time series analysis of GC performance across the entire cluster is one of the ways to detect "bad" nodes. The data/plots presented in this section were generated using R.

Heap Usage Summary

Heap usage may across nodes for a multitude of reasons. For example, owing to non-perfect load balancer, higher number of requests may be directed to a particular node that would result in higher heap usage and consequently increase GC pressure. As a first cut, we plotted the time series of heap usage for all the nodes to obtain a birds eye view of the entire cluster. Any obvious outlier can be detected from such a plot. Figure 4 presents the time series of heap usage of the chosen cluster.

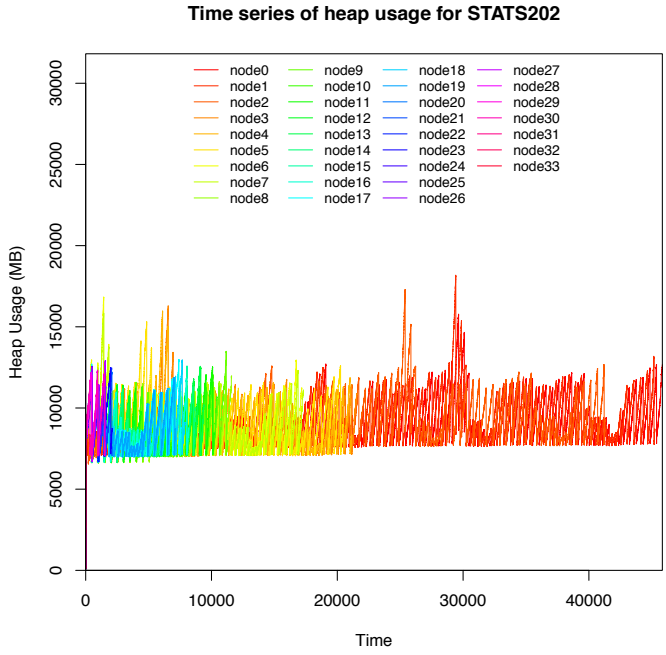

Figure 4

From the time series above we note that there are no obvious outliers. Next, for each time series we compute the first and second moment – mean and standard deviation respectively. The data is presented below:

```
Mean heap usage on node node0:  8916.806, standard deviation:  1287.865
Mean heap usage on node node1:  8930.43, standard deviation:  1342.247
Mean heap usage on node node2:  9024.683, standard deviation:  1320.615
Mean heap usage on node node3:  8833.356, standard deviation:  1400.383
Mean heap usage on node node4:  8886.79, standard deviation:  1432.032
Mean heap usage on node node5:  8864.896, standard deviation:  1415.554
Mean heap usage on node node6:  8826.984, standard deviation:  1460.153
Mean heap usage on node node7:  8837.163, standard deviation:  1429.953
Mean heap usage on node node8:  8587.584, standard deviation:  1215.003
Mean heap usage on node node9:  8639.791, standard deviation:  1250.882
Mean heap usage on node node10:  8597.872, standard deviation:  1211.660
Mean heap usage on node node11:  8670.956, standard deviation:  1271.779
Mean heap usage on node node12:  8545.246, standard deviation:  1226.987
Mean heap usage on node node13:  8611.459, standard deviation:  1263.144
Mean heap usage on node node14:  8534.202, standard deviation:  1177.827
Mean heap usage on node node15:  8592.963, standard deviation:  1247.732
Mean heap usage on node node16:  8535.877, standard deviation:  1228.79
Mean heap usage on node node17:  8640.466, standard deviation:  1330.246
Mean heap usage on node node18:  8662.608, standard deviation:  1339.855
Mean heap usage on node node19:  8529.378, standard deviation:  1253.819
```

[2] Hoelzle and Barroso point out that hardware failure in the cloud is more of a norm than exception. Dai et al. list the various types of failures in the cloud. Vishwanath and Nagappan reported that 8% of all servers can expect to see at least 1 hardware incident in a given year and that this number is higher for machines with lots of hard disks.

```
Mean heap usage on node node20:  9341.118, standard deviation:  1535.381
Mean heap usage on node node21:  9233.24, standard deviation:  1491.748
Mean heap usage on node node22:  9097.511, standard deviation:  1396.751
Mean heap usage on node node23:  9308.15, standard deviation:  1538.446
Mean heap usage on node node24:  9403.009, standard deviation:  1624.591
Mean heap usage on node node25:  9413.621, standard deviation:  1744.138
Mean heap usage on node node26:  9844.594, standard deviation:  1650.387
Mean heap usage on node node27:  9634.47, standard deviation:  1635.731
Mean heap usage on node node28:  9285.548, standard deviation:  1572.141
Mean heap usage on node node29:  9518.678, standard deviation:  1709.409
Mean heap usage on node node30:  9284.458, standard deviation:  1591.032
Mean heap usage on node node31:  5948.449, standard deviation:  2847.418
Mean heap usage on node node32:  163.081, standard deviation:  72.18429
Mean heap usage on node node33:  4137.322, standard deviation:  2463.644
```

For best performance, one like to minimize both the mean heap usage (μ_H) and the standard deviation (σ_H). In other words, one would like to maximize the following metric:

$$\frac{1}{\mu * \sigma}$$

The metric is similar to Sharpe Ratio (used in finance) but in the current case the mean (μ_H) is the denominator as we would like to have lower mean heap usage.[3]

Anomaly Detection

We computed the above metric for each node and plotted a scatter plot using R. Figure 4 shows the scatter plot for the metric.

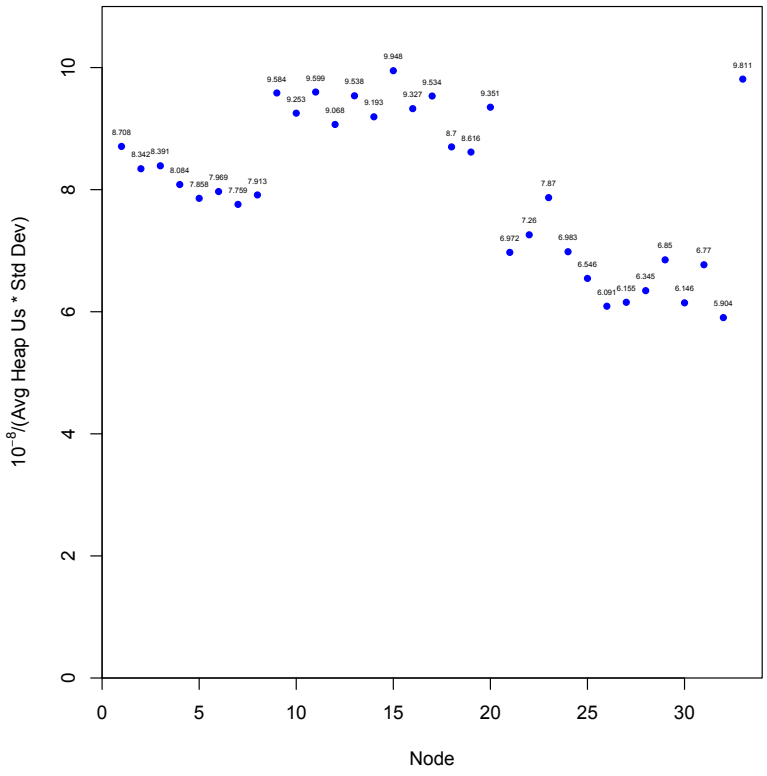

Figure 4

Next, we used k-Means to determine outlier nodes. We tried different values of k=2, 3 and narrowed down to k=3 as it clustered the nodes with low values of the metric. A scatter plot with clustering-based color coding is shown in Figure 5.

[3] A high heap usage increases memory pressure which in turn adversely impacts performance.

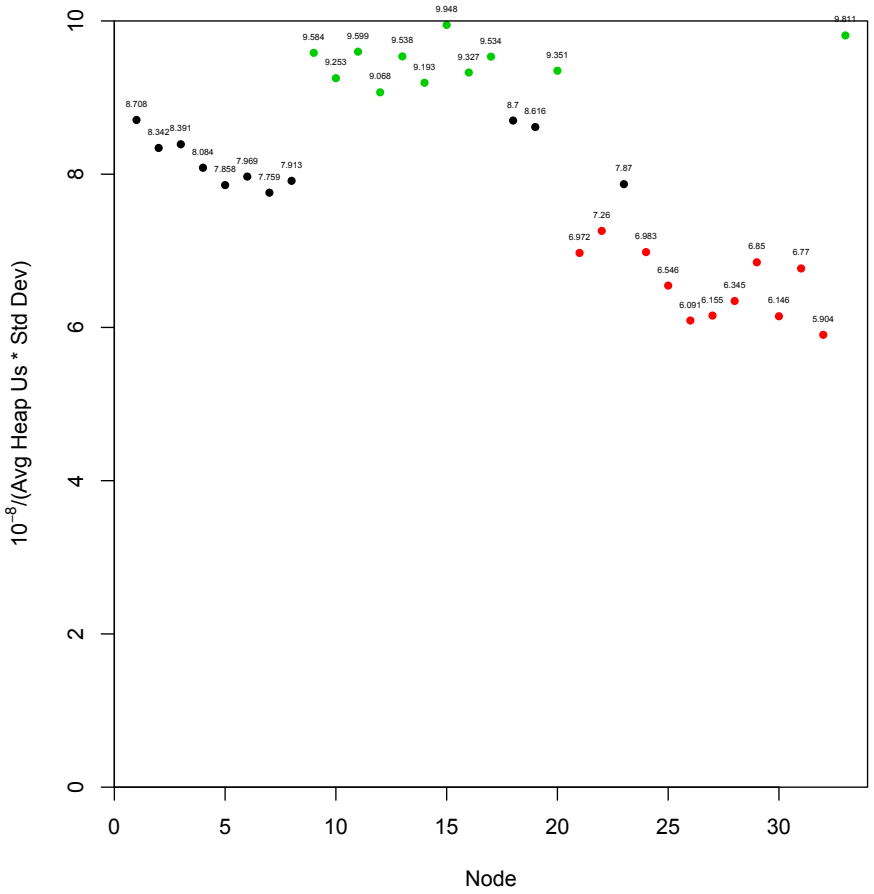

Figure 5

From Figure 5 we see that there are set of 11 nodes which have a low (relative to others) of the metric.

Young Gen Usage Summary

Akin to overall heap usage, we extracted the time series for Young Gen for each node and computed the mean and standard deviation for each time series. The data is presented below:

```
Mean young gen usage on node node0:  87.85087, standard deviation:  33.87566
Mean young gen usage on node node1:  88.55817, standard deviation:  35.00981
Mean young gen usage on node node2:  86.2207, standard deviation:  33.92038
Mean young gen usage on node node3:  81.56372, standard deviation:  31.03004
Mean young gen usage on node node4:  81.66883, standard deviation:  30.97657
Mean young gen usage on node node5:  82.80407, standard deviation:  32.66848
Mean young gen usage on node node6:  83.08853, standard deviation:  34.00983
Mean young gen usage on node node7:  82.5501, standard deviation:  33.82947
Mean young gen usage on node node8:  80.79707, standard deviation:  32.88013
Mean young gen usage on node node9:  79.69276, standard deviation:  33.63487
Mean young gen usage on node node10:  80.77414, standard deviation:  32.8249
Mean young gen usage on node node11:  81.31262, standard deviation:  34.40717
Mean young gen usage on node node12:  80.96931, standard deviation:  34.66557
Mean young gen usage on node node13:  81.65107, standard deviation:  35.32639
Mean young gen usage on node node14:  81.55016, standard deviation:  34.75346
Mean young gen usage on node node15:  82.01223, standard deviation:  35.92627
Mean young gen usage on node node16:  81.87328, standard deviation:  36.21742
Mean young gen usage on node node17:  83.48216, standard deviation:  38.25221
Mean young gen usage on node node18:  82.14804, standard deviation:  37.48508
Mean young gen usage on node node19:  81.81286, standard deviation:  35.99382
Mean young gen usage on node node20:  86.59663, standard deviation:  52.3834
Mean young gen usage on node node21:  87.40986, standard deviation:  54.7278
Mean young gen usage on node node22:  91.46672, standard deviation:  57.99277
Mean young gen usage on node node23:  89.55589, standard deviation:  56.67445
Mean young gen usage on node node24:  91.18868, standard deviation:  57.93368
Mean young gen usage on node node25:  90.65398, standard deviation:  60.29953
```

```
Mean young gen usage on node node26:  92.8079, standard deviation:  63.1866
Mean young gen usage on node node27:  93.70674, standard deviation:  64.72424
Mean young gen usage on node node28:  96.10727, standard deviation:  66.66045
Mean young gen usage on node node29:  100.1759, standard deviation:  71.53554
Mean young gen usage on node node30:  94.00995, standard deviation:  62.71223
Mean young gen usage on node node31:  304.8011, standard deviation:  111.7066
Mean young gen usage on node node32:  163.081, standard deviation:  72.18429
Mean young gen usage on node node33:  369.6325, standard deviation:  81.46092
```

Anomaly Detection

We computed the above metric for each node and plotted a scatter plot using R. Figure 4 shows the scatter plot for the metric.

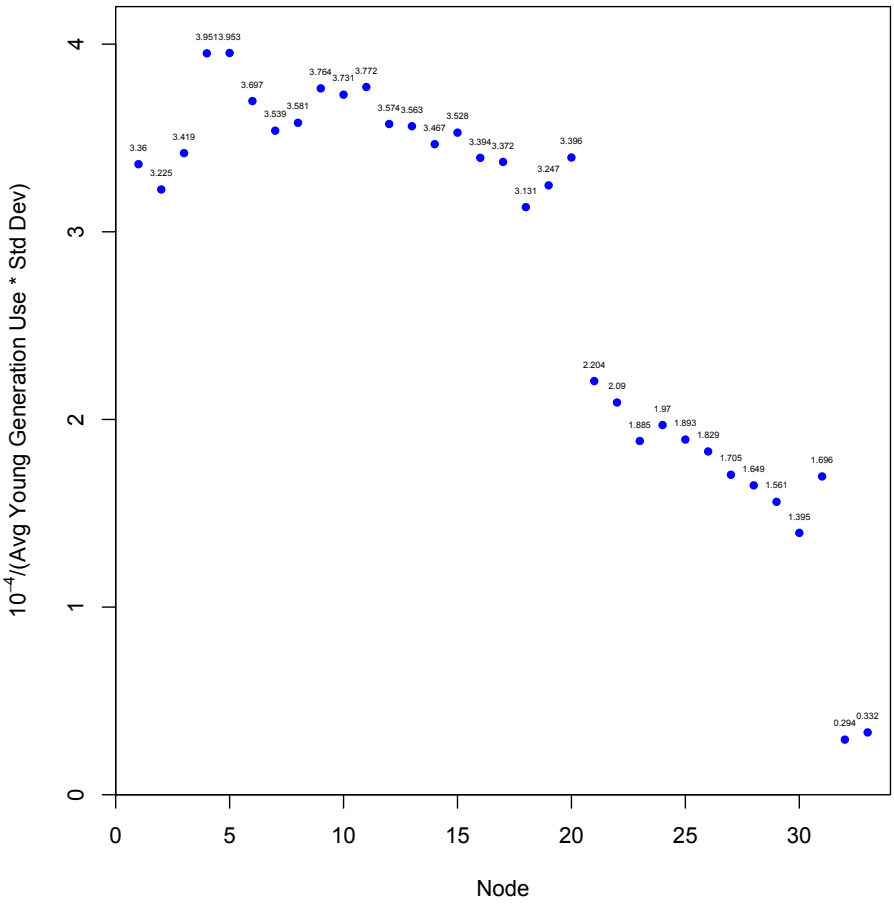

Figure 6

Next, we used k-Means to determine outlier nodes. Akin to anomaly detection based on heap usage, for anomaly detection based on young gen usage, we tried different values of k=2, 3 and narrowed down to k=3. A scatter plot with clustering-based color coding is shown in Figure 7.

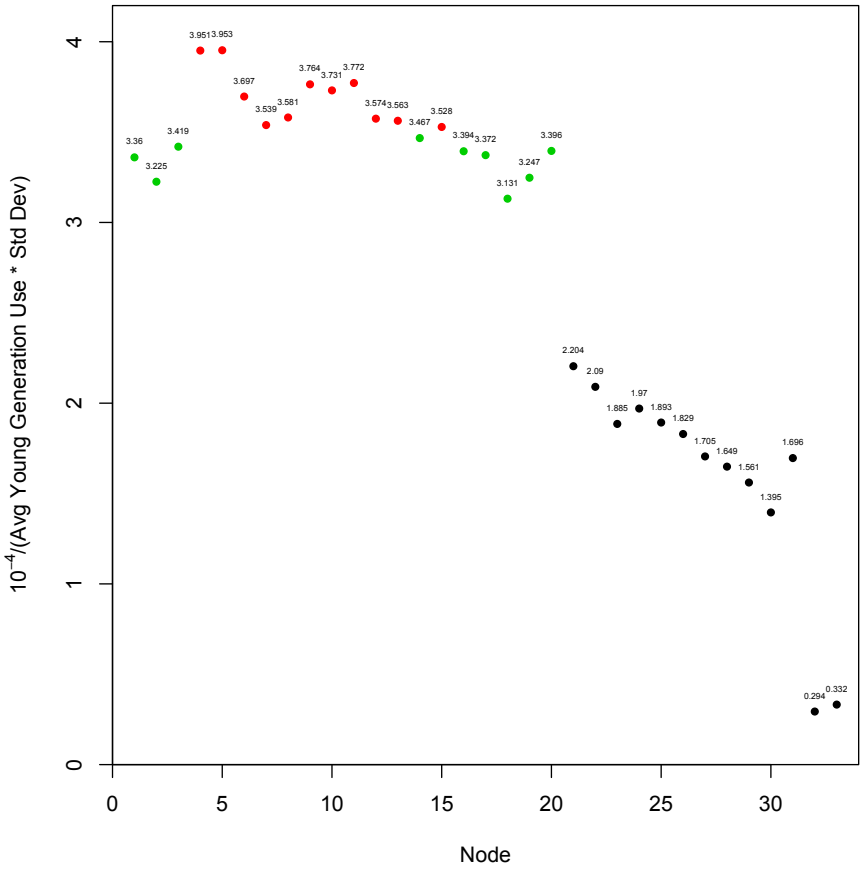

Figure 7

Unlike clustering for heap usage, from Figure 7 we note that k=3 is probably not the ideal choice as node 32 and nodes 33 form a "natural" separate cluster. Based on Figure 7, nodes 32 and 33 can be regarded as outliers. However, based on Figure 5, node 33 is not an outlier. As a first approximation, one can potentially take an intersection of the two outlier sets (obtained based clustering of overall heap usage and young gen usage). Consequently, node 32 would be considered as an outlier in the current context.

References

[1] http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#cms.measurements
[2] https://blogs.oracle.com/poonam/entry/understanding_cms_gc_logs
[3] http://blog.ragozin.info/2011/10/java-cg-hotspots-cms-and-heap.html
[4] https://blogs.oracle.com/poonam/entry/understanding_g1_gc_logs
[5] https://blogs.oracle.com/jonthecollector/
[6] http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html#0.0.0.Measurement%7Coutline