**Heap Space**:

Looking at the heap memory graph in visual VM, “used memory” increases continuously. Even if major GCs are run, very less memory is freed.

***Reason:*** There are high numbers of objects created and probably not being de-referenced even after its use. E.g. keeping objects in session unnecessarily. Change in java version will not have any effect on these kinds of objects. To identify exact reason, we may need to take it by each scenario and check it out be looking at the code.

***Resolution:*** If the reason is the one mentioned above, we may require major code changes in each module where we need to explicitly de-reference objects after its use. This will ensure that whenever GCs are run, it is fruitful and it may also lead to use of mainly minor GCs only.

***Efforts:*** Very high. Not feasible in this release

***Improvements:*** Medium to high.

**GC Activity**:

We observed GC activity in both 1.5 & 1.6 runs, but found similar observation in both versions. For first half of the script execution, minor GCs are performed most of the time. For the second half of the script execution, major GCs are performed every-time, **no minor GCs are performed (in second half).**

***Observation:***

*Average time taken to run 1 minor GC: 80-100ms*

*Average time taken to run 1 major GC: 4 seconds*

*Frequency of GC runs*

*Minor GC(first half of the script in java 1.5) : approx. every 1 second*

*Major GC(first half of the script in java 1.5) : approx. every minute*

*Minor GC(first half of the script in java 1.6) : approx. every 5 seconds*

*Major GC(second half of the script in java 1.5) : approx. after every 2 seconds*

*Major GC(second half of the script in java 1.6) : approx. after every 4 seconds*

***Reason:*** Initially, there is high amount of free memory available in the heap, so minor GCs are run in the smaller (typically 1/4th) portion (temporary or young generation space) of the heap memory and is very fast. There are more amount of objects that lives long, so objects keep being copied from temporary to permanent space (part of heap only). Which eventually gets loaded more and more, which required major GC(responsible to clean up permanent or old generation space).

***Resolution:***

* Please refer 1-6-parameter-scenarios.docx.

***Efforts:*** High

***Improvements:*** Low to medium

**Log Level**:

* There are many unnecessary INFO level logs present in the code. If we remove such unnecessary logs, we can see major performance improvements.

***Resolution:*** For this release we may go ahead by changing log level to WARNING.

***Efforts:*** Very Low

***Improvements:*** Very High

***Risk:*** In case, we have any issue in production and we need to use info level logs, we may need to reproduce the issue with INFO log level. In most of the cases, we need WARNING level logs are sufficient for knowing the errors.

**Memory Usage:**

***String & Char[]:*** 33% of the total heap space is used by String & char[].

***Resolution:*** We can reduce this by using -XX:+UseCompressedStrings parameter. This parameter uses byte[] to construct String instead of char[] wherever possible.

***Efforts:*** Low

***Improvements:*** Can’t say. R&D is required to check that whether above change is effective without any code changes or not.

***Net.sourceforge.jtds.jdbc.ColInfo:*** 13% of the total heap space is used.

***Byte[]:*** 9% of the total heap space is used.

***HashMap:*** 8% of total heap space is used.

For last three classes, we can’t do much just by tweaking parameters, and doesn’t give any hint as of now that what could be going wrong there or whether anything is going wrong or not.