HITACHI
Inspire the Next

Tuning JVM Garbage Collection for Pentaho

This page intentionally left blank.

Contents

This page intentionally left blank.

Overview

The purpose of this document is to highlight the available options within the Oracle Java Development Kit (JDK) Java Virtual Machine (JVM) to improve overall speed and performance, particularly with garbage collection.

Our intended audience is Pentaho administrators, or anyone with a background in Java who is interested in maximizing VM speed and performance.

The intention of this document is to speak about topics generally; however, these are the specific versions covered here:

Software	Version(s)
Pentaho	6.x, 7.x, 8.x

The Components Reference in Pentaho Documentation has a complete list of supported software and hardware.

Before You Begin

Before beginning, use the following information to prepare for the procedures described in the main section of the document.

This document assumes that you have knowledge of Pentaho and Java and that you have already installed JDK on your system.

Use Case: Memory Slowdown

Janice is a Pentaho administrator who wants to improve speed and performance in her installation.

Janice decides to optimize JVM garbage collection.

Java Descriptions

This section provides brief descriptions of each JVM compartment, and JVM's garbage collection program. You will be given a breakdown of each compartment, how they are used, and what they are used for.

You can find details on these topics in the following sections:

What is JVM Garbage Collection?

The garbage collector is a program which runs on the JVM and eliminates objects out of the memory that are no longer being used by a Java application. It is a form of automatic memory management.

More information about JVM and JVM Garbage Collection can be found on Oracle's JVM documentation.

JVM Compartments

Oracle's documentation provides details on each compartment of the JVM, its uses, and its purposes. The following list provides quick access to each compartment description:

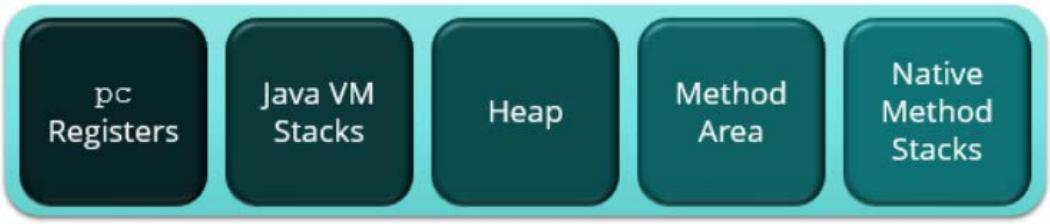

Figure 1: JVM Compartments

Process-Dependent Memory Parameters

This section provides information on JVM heap space, permanent generation memory, and settings. You will learn how to control the amount of memory that has been distributed to your JVM, and how to store class definitions with their appropriate size. Topics covered in this section are as follows:

- JVM Heap Space
- JVM PermSize
- Sample JVM Settings

JVM Heap Space (-Xms and -Xmx)

The `-Xms` and `-Xmx` control the amount of heap memory your application's JVM is allocated. Keep in mind that the heap memory, as defined by `-Xms` and `-Xmx`, is not the total memory used by a JVM. The total memory includes heap, permanent generation memory, thread stacks, and shared libraries.

If you have a machine with 96GB of memory, and you want to use all of it for processing KJB/KTR, run multiple instances of Kitchen, Pan, or Carte with a maximum of 24GB instead of using the total physical RAM for a single instance.

You should also leave at least 33% additional memory for general operating systems and other applications. Keep in mind that Pentaho products are multi-threaded, so if there are a lot of threads, Pentaho can consume considerable memory when you have many steps in the transformation.

JVM PermSize (Permanent Generation Memory)

This information applies only to Pentaho 7.0 and earlier, and potentially 7.1, as this section does not apply to Java version 8 and up.

The permanent generation memory is used to store class definitions. Because Pentaho applications typically load a lot of class definitions dynamically, it is best to increase this value. A sufficient size would be 256MB, and 512MB to 1024MB for server JVMs.

Setting `-XX:PermSize` and `-XX:MaxPermSize` will typically give better performance. Oracle's help center and Java SE documentation have more information on this topic.

Sample JVM Settings

Java applications usually require JVM tuning. The following sample settings should be considered for server applications.

Keep in mind that the stack size is in KB, not MB or GB. If the stack size gets too large and you launch threads, it may cause significant problems. Stack size should be as small as you can make it, while still having everything work well.

Table 1: Java Virtual Machine Settings

Property	Definition
-server	Selects server application runtime optimizations. The directory server will take longer to start, but will be more aggressively optimized for higher throughput.
-Xms24G	Sets the initial and minimum Java heap size.
-Xmx24G	Sets the maximum Java heap size, which can vary depending on the operating system you are running.
-Xss256k	Sets the thread stack size. Thread stacks are memory areas allocated for each Java thread for their internal use. This is where the thread stores its local execution state.
-DJava.rmi.server.hostname=<external IP>	The value of this property represents the hostname string that should be associated with remote stubs for locally created remote objects, to allow clients to invoke methods on the remote object. The default value of this property is the IP address of the local host, in dotted-quad format.
The values below are applicable as of this writing and regularly change. See the Java documentation for the most current recommendations.	
-Xxn512k	Explicitly defines the size of the young generation.
-XX:PermSize=256m	This property value may need to change based on your implementation. `PermSize` is additional to the `-Xmx` value set by the user on the JVM options. `MaxPermSize` allows the JVM to grown the `PermSize` to the specified amount.
-XX:MaxPermSize=1024m **JDK 1.7 Only (deprecated in JDK 1.8)**	Size of the permanent generation (5.0 and newer: 64-bit VMs are scaled 30% larger; 1.4 `amd64`: 96m; 1.3.1 `-client`: 32m). This is applicable only for Pentaho versions prior to 8.0.
-XX:+ExplicitGC InvokesConcurrent	Enables the concurrent marking task within CMS collector to perform in parallel with multiple processors, which reduces the duration and enables better support applications with larger number of threads and high object allocation rates, particularly on large multiprocessor machines.
-XX:+ScavengeBefore FullGC	Attempts to free up memory by scavenging youngest generation before doing a full garbage collection.
-XX+CMSScavenge BeforeRemark	Attempts scavenge before the concurrent mark sweep (CMS) remark step. This helps keep the remark phase short.
-XX:+UseConcMark SweepGC	This old generation collector does most of the work in the background without stopping application threads.
-XX:+CMSParallel RemarkEnabled	Allows remarking to be done in parallel to program execution; this is great for multi core servers.

Property	Definition
-XX:+UseCMS Initiating OccupancyOnly	Indicates all concurrent garbage collection CMS cycles should start based on value of `-XX:CMSInitiatingOccupancyFraction`. Generally, it is advisable to use both `-XX:CMSInitiatingOccupancyFraction=<percent>` and `-XX:+UseCMSInitiatingOccupancyOnly`
-XX:CMSInitiating OccupancyFraction =<percent>	The percentage of CMS generation occupancy necessary to start a CMS collection cycle. A negative value means that `CMSTriggerRatio` is used.
-XX:+UseConcMark SweepGC	Sets the garbage collector policy to the concurrent (low pause time) garbage collector (also known as CMS).
-XX:+CMS IncrementalMode	Enables the incremental mode; works only with `-XX:+UseConcMarkSweepGC`.
-XX:+CMSIncremental Pacing	Enables automatic adjustment of the incremental mode duty cycle based on statistics collected while the JVM is running.
-XX:+CMSClass UnloadingEnabled	Tells JVM to unload classes which are not needed anymore by the running application
-XX:+UseParNewGC	Enables multi-threaded young generation collection.
-XX:+Disable ExplicitGC	Disables calls to `System.gc()` that would be enabled by default (`-XX:-DisableExplicitGC`). Note that the JVM still performs garbage collection when necessary.
-XX:NewRatio=2	Ratio of old generation size to young generation size. For example, a value of 2 means the maximum size of the old generation will be twice the maximum size of the young. In other words, the young generation can get up to 1/3 of the heap.
-XX:SurvivorRate =<ratio>	Ratio of the survivor space relative to the even size, calculated using this formula: *survivor ratio*= $\left(\frac{\text{young size}}{\text{survivor size}}\right)$-2
-XX:+Tiered Compilation	Introduced in Java SE 7, this brings client startup speeds to the server VM. Normally, a server VM uses the interpreter to collect profiling information about methods that are fed into the compiler. In the tiered scheme, in addition to the interpreter, the client compiler is used to generate compiled versions of methods that collect profiling information about themselves. Because the compiled code is substantially faster than the interpreter, the program executes with greater performance during the profiling phase. In many cases, a startup that is even faster than with the client VM can be achieved, because the final code produced by the server compiler may already be available during the early stages of application initialization. The tiered scheme can also achieve better peak performance than a regular server VM, because the faster profiling phase allows a longer period of profiling, which may yield better optimization.
-XX:+UseCompressed 0ops	This is on by default, unless `-Xms` is less than 32GB, in which case it turns off if not specified.

JVM Tools (Diagnostic, Monitoring, and Troubleshooting)

This section provides information on ways to gather what is going on within the JVM, which can point out areas needing attention.

Key Points

Being ready to find and react to problems when they come up requires:

- Knowing how to get logging turned on when needed, as detail logging should not be turned on during regular operation due to the possibility of the creation of a large-size file.
- Having visual JVM monitoring installed and ready to be used when needed, since using this visual tool allows you see threads, memory usage, and stats to quickly point to areas needing attention.
- Viewing and monitoring your environment during good times, so you have benchmarks for comparison.

JVM Verbose Logging

You can instruct the JVM to log certain details by adding arguments to the Java exec command line. For example:

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -
Xloggc:{full_path}/{file_name}.log
```

Table 2: JVM Verbose Logging

Verbose Command	Description
`-verbose:class`	Displays classes loaded by JVM, which can be helpful for knowing all classes being loaded in a particular scenario.
`-verbose:gc`	Shows details of garbage collection and of young, old, used, and total memory consumption. When combined with `-XX:+PrintGCTimeStamps -XX:+PrintGCDetails`, more rich details are added on the lines within the logs.
`-verbose:ini`	Displays the Java native methods when they are registered in the application.

VisualVM

VisualVM is a free visual tool that can allow you to see detailed information about Java applications while they are running on a JVM. VisualVM organizes JVM data that is retrieved by the Java Development Kit (JDK) tools and presents the information in a way that allows data on multiple Java applications to be quickly viewed—both local applications and applications that are running on remote hosts.

VisualVM features include:

- Display local and remote Java processes
- Display process configuration and environment
- Monitor process performance and memory
- Visualize process threads
- Profile performance and memory usage
- Take and display thread dumps
- Take and browse heap dumps
- Analyze core dumps
- Analyze applications offline

Related Information

Here are some links to information that you may find helpful while using this best practices document:

- Java Garbage Collection Basics
- Java HotSpot Virtual Machine Performance Enhancements
- Oracle Java Virtual Machine Specification
- Oracle: Tuning JVM Options
- Pentaho Components Reference
- VisualVM

Finalization Checklist

This checklist is designed to be added to any implemented project that uses this collection of best practices, to verify that all items have been considered and reviews have been performed.

Name of the Project:_____

Date of the Review:_____

Name of the Reviewer:_____

Item	Response	Comments
Did you run more than one instance of Kitchen, Pan, or Carte when working with JVM heap space?	YES_____ NO_____	
Did you increase the JVM PermSize value so that it is able to dynamically load many class definitions?	YES_____ NO_____	
Did you tune the Java applications required for the JVM?	YES_____ NO_____	
Did you refer to each of the key points put in place to find and react to problems that may have come up?	YES_____ NO_____	