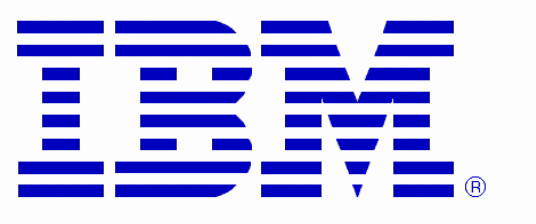


WebSphere Application Server Troubleshooting and Performance Lab on Docker

Authors

- Kevin Grigorenko (kevin.grigorenko@us.ibm.com)

Version History

- V2 (April 30th, 2019): Convert to Docker and modernize.
- V1 (December 14th, 2016): First version on VMWare.

Contents

1 Introduction

WebSphere Application Server[1] (WAS) is a platform for serving Java based applications. WAS comes in two product forms[2]:

1. Traditional WAS[3] (colloquially: tWAS): Released in 1998 and still fully supported.
2. WAS Liberty[4] (colloquially: Liberty or WebSphere Liberty): Released in 2012 and designed for fast startup, composability, and the cloud. The commercial WAS Liberty product is built on top of the open source core of Liberty called OpenLiberty[5].

The two products share some source code but differ in significant ways.

Both Traditional WAS and WAS Liberty come in different flavors including *Base* and *Network Deployment (ND)* in which ND layers additional features such as advanced high availability on top of Base.

1.1 Lab

This lab assumes the installation and use of Docker to run the lab. For example, install Docker Desktop for Windows or Mac hosts:

- Windows "Requires Microsoft Windows 10 Professional or Enterprise 64-bit."
  - Download: https://hub.docker.com/editions/community/docker-ce-desktop-windows
  - For details, see https://docs.docker.com/docker-for-windows/install/
- Mac "Requires Apple Mac OS Sierra 10.12 or above"
  - Download: https://hub.docker.com/editions/community/docker-ce-desktop-mac
  - For details, see https://docs.docker.com/docker-for-mac/install/
- For a Linux host, simply install and start Docker (sudo systemctl start docker):
  - For an example, see https://docs.docker.com/install/linux/docker-ce/fedora/

This lab covers the major tools and techniques for troubleshooting and performance tuning for both Traditional WAS and WAS Liberty, in addition to specific tools for each. There is significant overlap because a lot of troubleshooting and tuning occurs at the operating system and Java levels, largely independent of WAS.

The lab Docker images come with Traditional WAS and WAS Liberty pre-installed so installation and configuration steps and problem determination are largely skipped.

Note that the way we are using Docker in these lab Docker images[6,7,8] is to run multiple services in the same container (e.g. Remote Desktop, VNC, Traditional WAS, WAS Liberty, a full GUI server, etc.) and although this approach is valid and supported[9], it is not generally recommended for production usage. In this case, Docker is used primarily for easy distribution and building of this lab.

[1] https://www.ibm.com/cloud/websphere-application-platform
[2] http://public.dhe.ibm.com/ibmdl/export/pub/software/websphere/wasdev/documentation/ChoosingTraditionalWASorLiberty-16.0.0.4.pdf
[3] https://www.ibm.com/support/knowledgecenter/en/SSAW57/mapfiles/product_welcome_wasnd.html
[4] https://www.ibm.com/support/knowledgecenter/en/SSAW57_liberty/as_ditamaps/was900_welcome_liberty_ndmp.html
[5] https://github.com/OpenLiberty/open-liberty
[6] https://github.com/kgibm/dockerdebug/blob/master/fedorawasdebug/Dockerfile
[7] https://github.com/kgibm/dockerdebug/blob/master/fedorajavadebug/Dockerfile
[8] https://github.com/kgibm/dockerdebug/blob/master/fedoradebug/Dockerfile
[9] https://docs.docker.com/config/containers/multi-service_container/

1.2 Operating System

This lab is built on top of Linux (specifically, Fedora Linux, which is the open source foundation of RHEL/CentOS). The concepts and techniques apply generally to other supported operating systems although details of other operating systems vary significantly and are covered elsewhere[10].

1.3 Java

Traditional WAS ships with a packaged IBM Java.

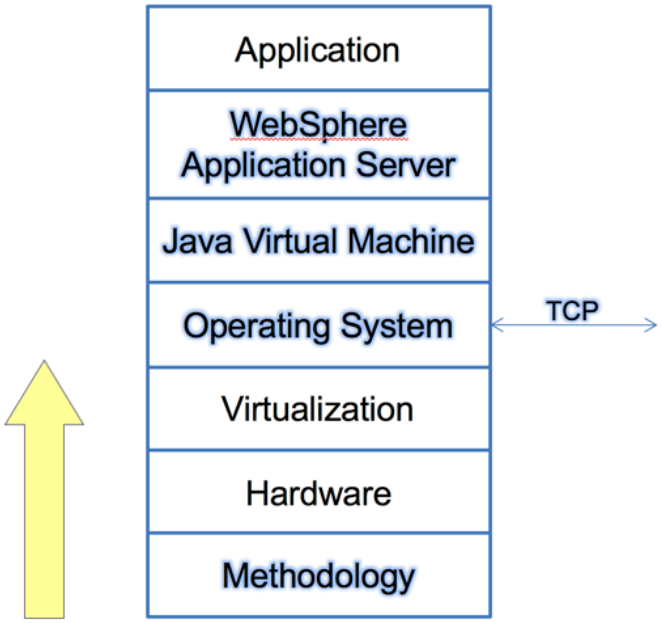
WAS Liberty supports any Java 8 or Java 11 compliant Java (with some minimum requirements[11]).

This lab uses IBM Java for both Traditional WAS and WAS Liberty. The concepts and techniques apply generally to other supported Java runtimes although details of other Java runtimes (e.g. HotSpot) vary significantly and are covered elsewhere[12].

The IBM Java virtual machine (named J9) has become largely open sourced into the OpenJ9 project[13]. OpenJ9 ships with OpenJDK through the AdoptOpenJDK project[14]. OpenJDK is somewhat different than the JDK that IBM Java uses. WAS Liberty >= 19.0.0.1 supports running with OpenJDK+OpenJ9 >= 11.0.2, although some tooling such as HealthCenter is not yet available in OpenJ9, so the focus of this lab continues to be IBM Java 8.

2 Core Concepts

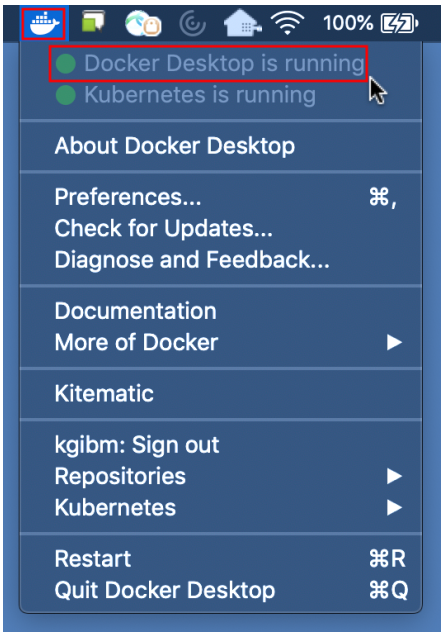
Problem determination and performance tuning are best done with all layers of the stack in mind. This lab will focus on the layers in bold below:



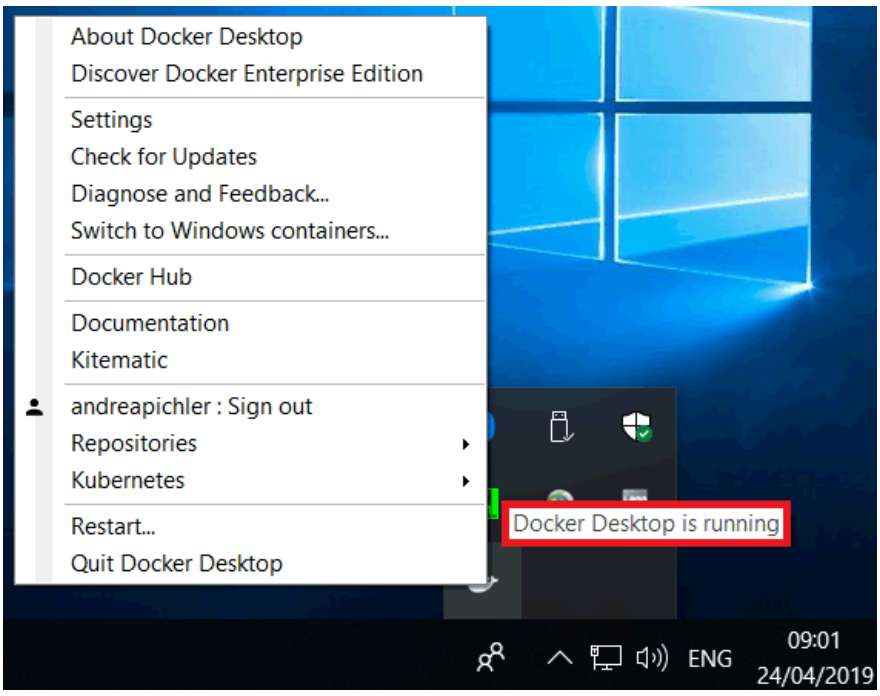
[10] https://publib.boulder.ibm.com/httpserv/cookbook/Operating_Systems.html
[11] https://www.ibm.com/support/knowledgecenter/SSAW57_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_restrict.html?view=kc#rwlp_restrict__rest13
[12] https://publib.boulder.ibm.com/httpserv/cookbook/Java.html
[13] https://github.com/eclipse/openj9
[14] https://adoptopenjdk.net/

3 Starting the Docker Container

1. Ensure that Docker is started. For example, start Docker Desktop and ensure it is running:
   macOS:



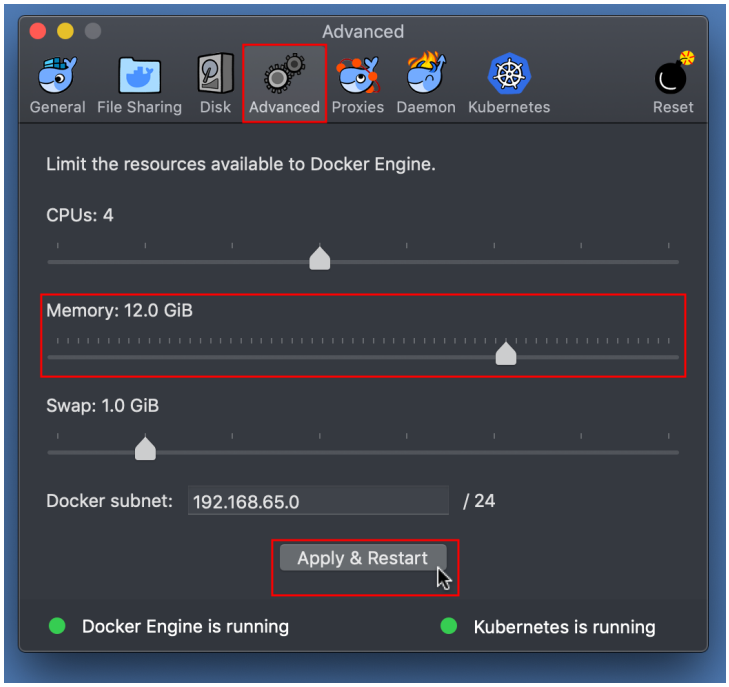
   Windows:



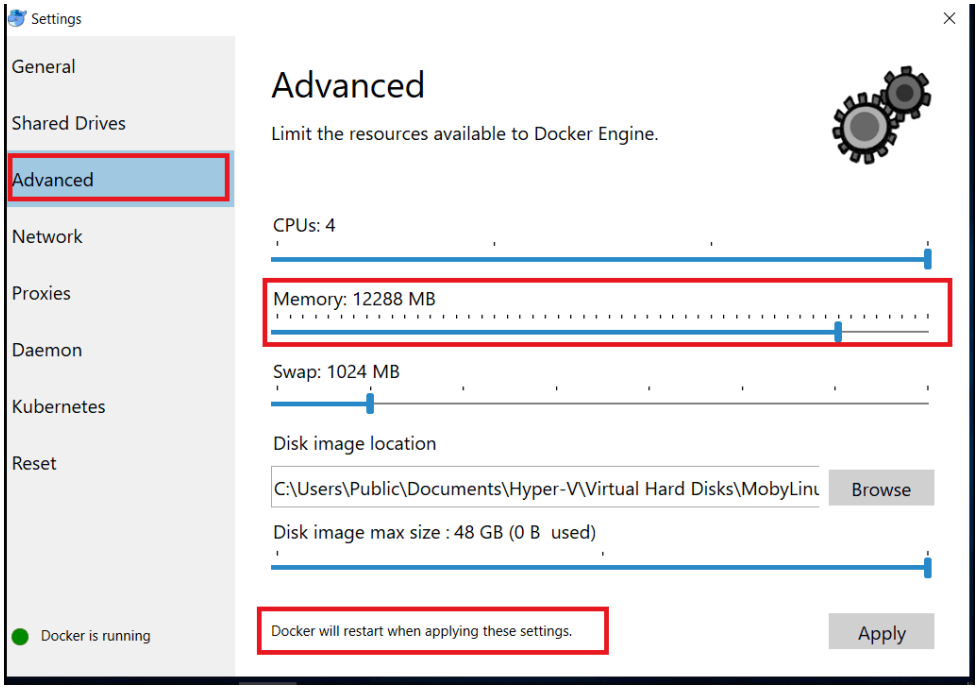
2. Ensure that Docker receives sufficient resources, particularly memory:
   a. Click the Docker Desktop icon and select “Preferences…” (on macOS) or “Settings” (on Windows)
   b. Select the Advanced tab.
   c. Increase Memory, ideally to at least 8GB.

d. Click Apply & Restart.

macOS:

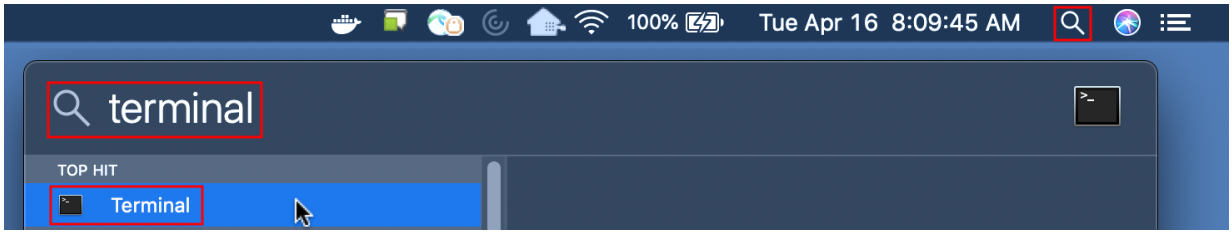


Windows:

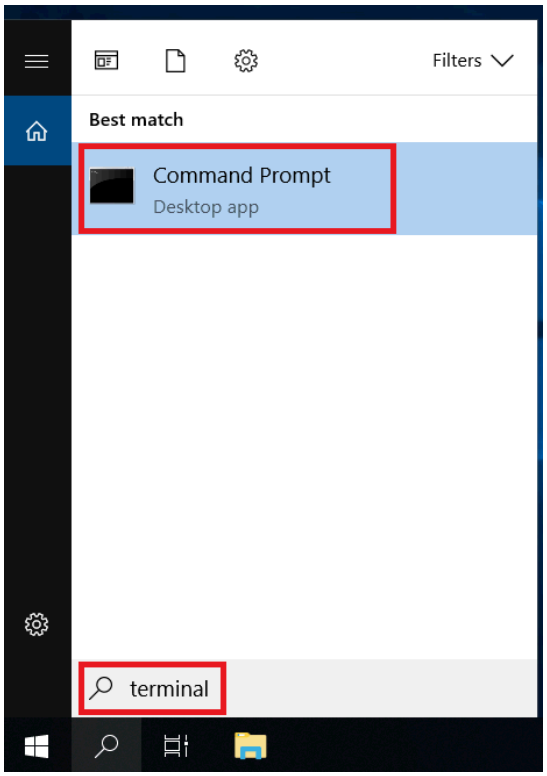


3. Open a terminal or command prompt:

   macOS:



   Windows:



4. Download the kgibm/fedorawasdebug image and related images:

```
docker pull kgibm/fedorawasdebug
```

   a. Note that these images are more than 10GB.

5. Run the kgibm/fedorawasdebug image as a container:

```
docker run --cap-add SYS_PTRACE --ulimit core=-1 --rm -p 9080:9080 -p 9443:9443 -p
9043:9043 -p 9081:9081 -p 9444:9444 -p 3389:3389 -p 5901:5901 -p 5902:5902 -p 22:22
-it kgibm/fedorawasdebug
```

6. Note that you may see various CNTR0019E/javax.ejb.NoSuchEJBException errors which are expected and will be resolved during the configuration phase of the lab.

7. Remote Desktop or VNC into the container:

   a. macOS VNC client:

      i. Open another tab in the terminal and run:

1. open vnc://localhost:5902
2. Password: **websphere**

b. Linux VNC client:

i. Open another tab in the terminal and run:

1. vncviewer localhost:5902
2. Password: **websphere**

c. Windows Remote Desktop client:

i. Open Remote Desktop

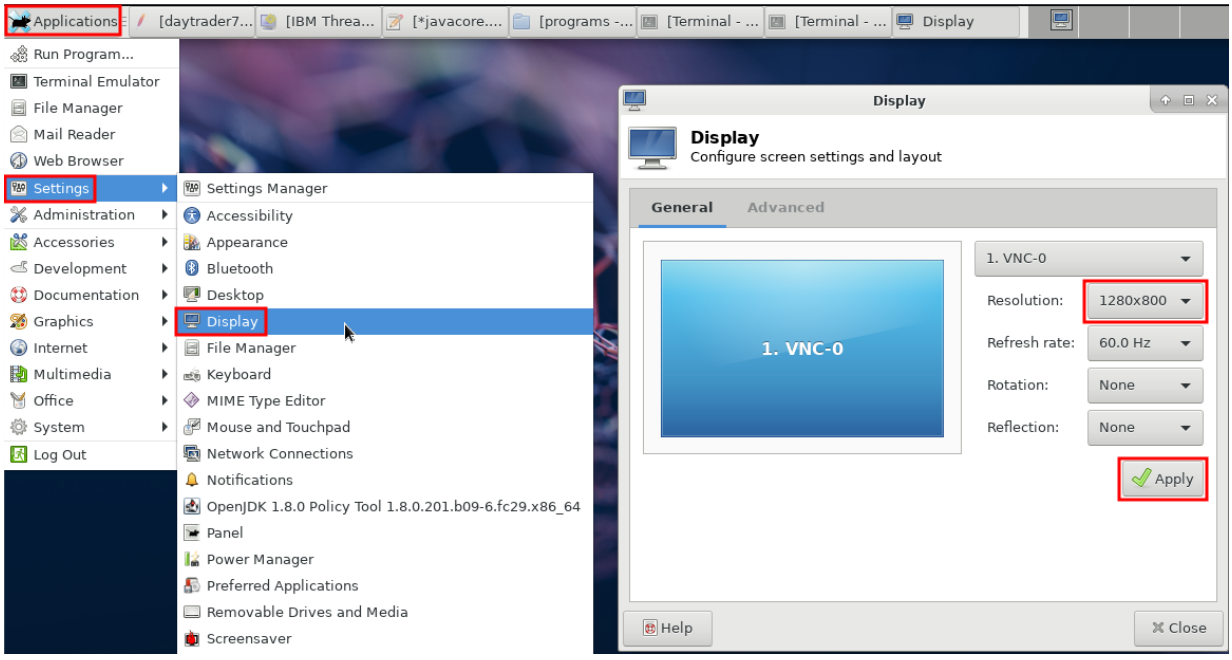
1. Connect to the host name **localhost**
2. User: **was**
3. Password: **websphere**

d. SSH:

i. If you want to simulate a production-like environment, you can instead SSH into the container (e.g. using terminal `ssh` or PuTTY):

1. ssh was@localhost
2. Password: **websphere**

8. When using VNC, you may change the display resolution:



9. Test WAS Liberty by going to http://localhost:9080/ in your host browser or the remote desktop/VNC browser.
10. Test Traditional WAS by going to http://localhost:9081/swat/ in your host browser or in the remote desktop/VNC browser.

a. Note that Traditional WAS takes a few minutes to start up after the container has started.

b. Test the Traditional WAS Administrative Console by going to https://localhost:9043/ibm/console in your client browser or in the remote desktop/VNC browser.

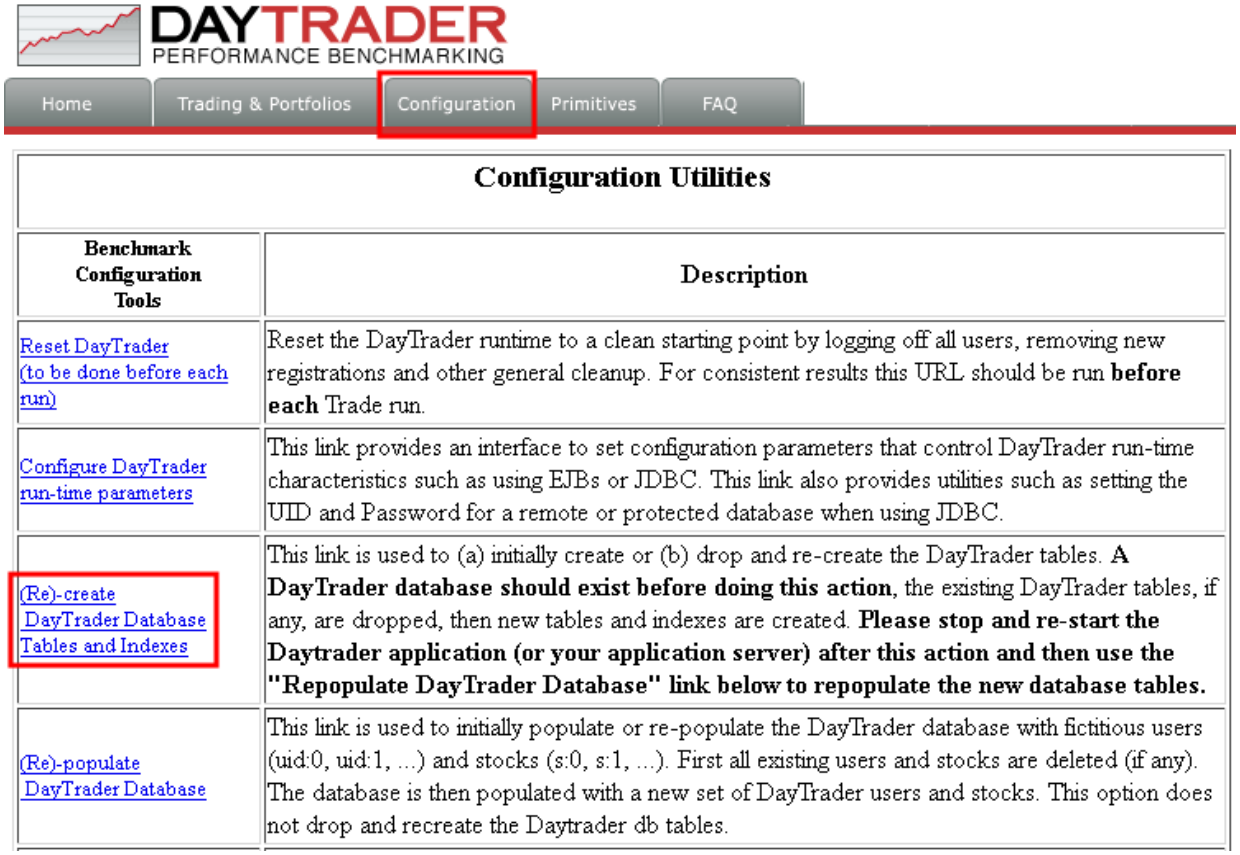
   i. User: **wsadmin**

   ii. Password: **websphere**

3.1 The DayTrader Sample Application

The DayTrader7 sample application[15] that we'll be using is pre-installed in the lab image but requires some initial preparation:

1. Open http://localhost:9080/daytrader/
2. Click the "Configuration" tab, and click "(Re)-create DayTrader Database Tables and Indexes":



[15] https://github.com/WASdev/sample.daytrader7

3. This will open a new tab and the database should be created with the following output:

TradeBuildDB: **** Database Product detected: Apache Derby ****

TradeBuildDB: **** The DDL file at path */dbscripts/derby/Table.ddl* will be used ****

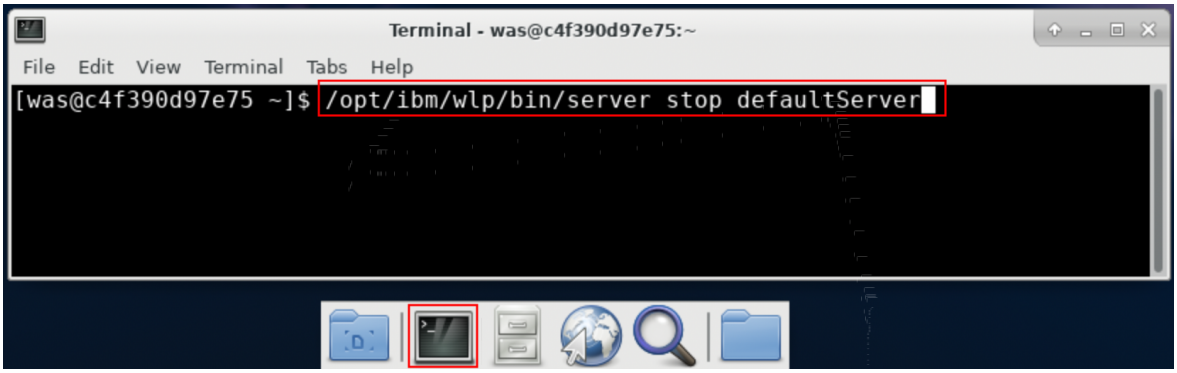
TradeBuildDB: Building DayTrader Database...
This operation will take several minutes. Please wait...
TradeBuildDB: **** Dropping and Recreating the DayTrader tables... ****

TradeBuildDB: **** DayTrader tables successfully created! ****

Please Stop and Re-start your Daytrader application (or your application server) and then use the "Repopulate Daytrader Database" link to populate your database.

4. The DayTrader application requires that the application is restarted after creating the database tables, so we will simply restart Liberty.
5. In the remote desktop or VNC viewer, open a terminal, type the following command to stop the Liberty server and press Enter:

```
/opt/ibm/wlp/bin/server stop defaultServer
```

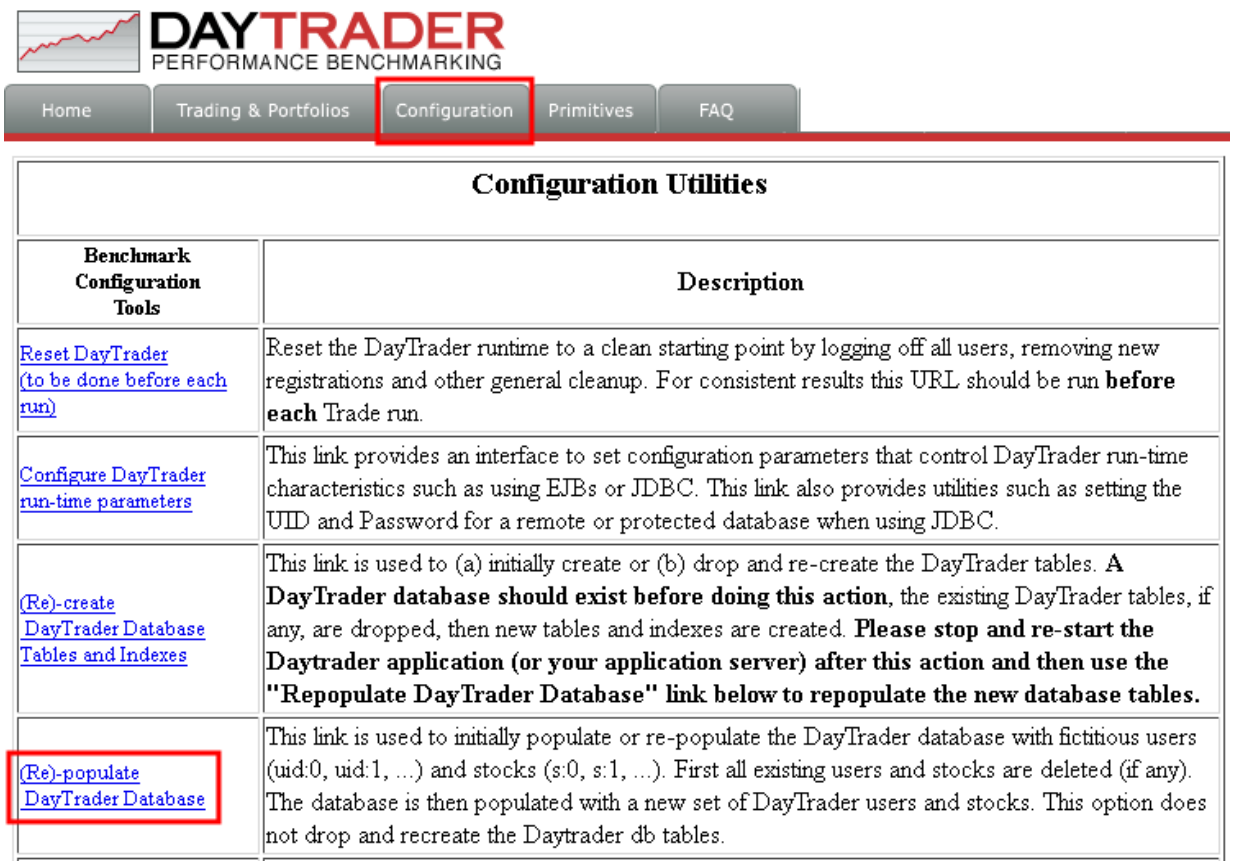


6. After the previous command completes, type the following command to start the Liberty server and press Enter:

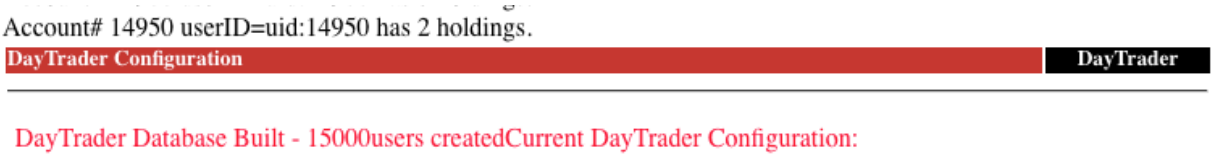
```
/opt/ibm/wlp/bin/server start defaultServer
```

7. Once the start command completes, refresh the DayTrader website at http://localhost:9080/daytrader/

8. Click "Configuration" and click "(Re)-populate DayTrader Database":



9. This will take 5-10 minutes depending on your computer and disk speeds. Keep scrolling to the bottom of the browser and wait until the bottom of the browser output shows "DayTrader Database Built":

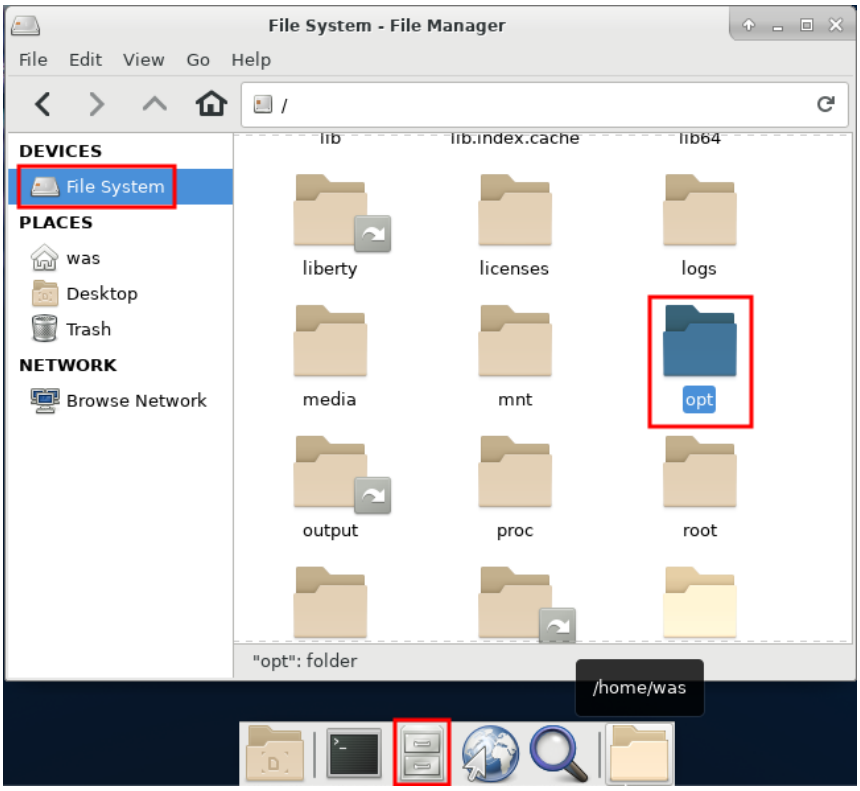


3.2 Apache Jmeter

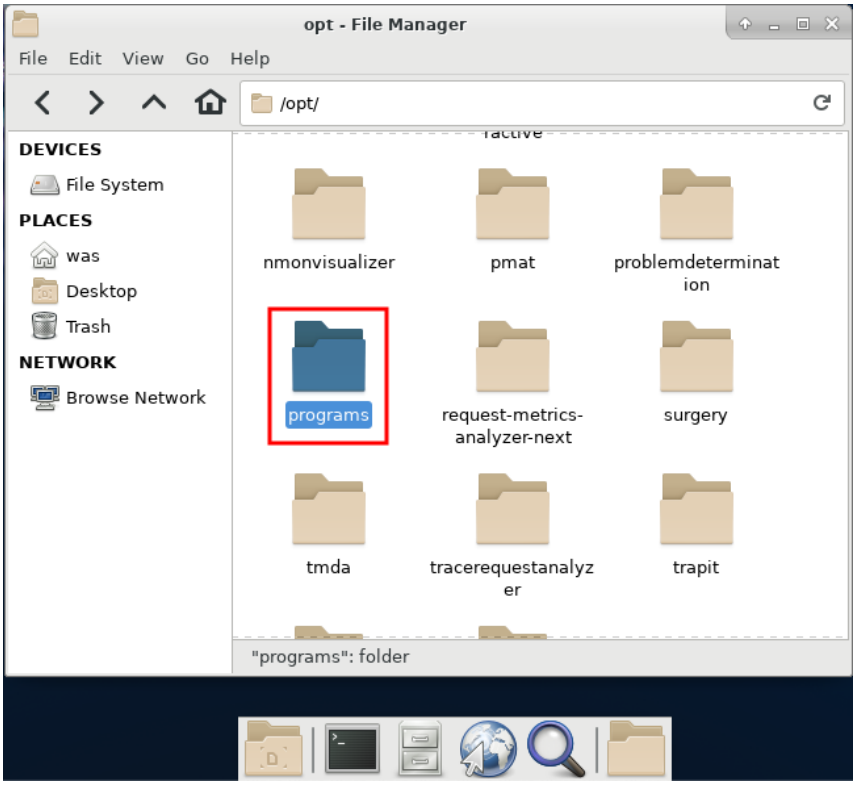
Apache JMeter[16] is a free tool that drives artificial, concurrent user load on a website. The tool is pre-installed in the lab image and we'll be using it to simulate website traffic to the DayTrader application.

[16] https://jmeter.apache.org/

1. Open File Manager and navigate to /opt:

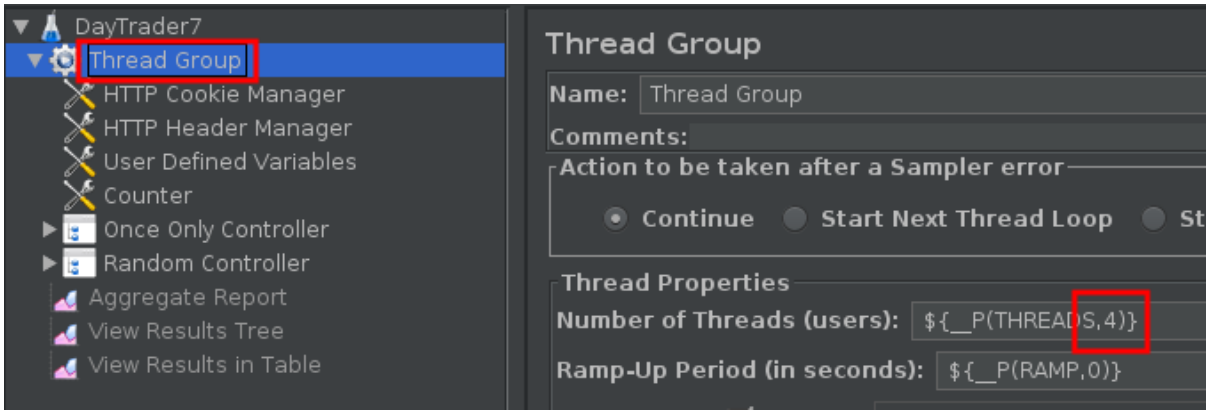


2. Navigate to /opt/programs/:

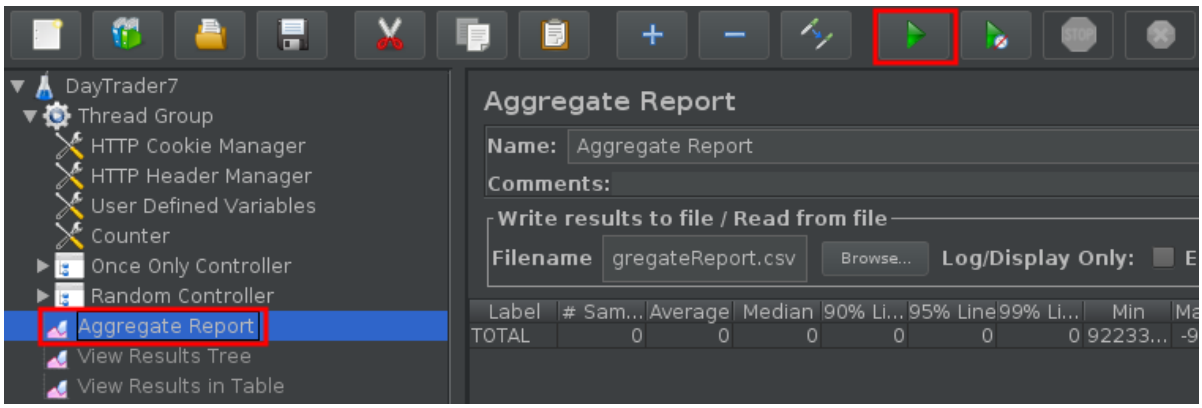


3. Double click on **JMeter**

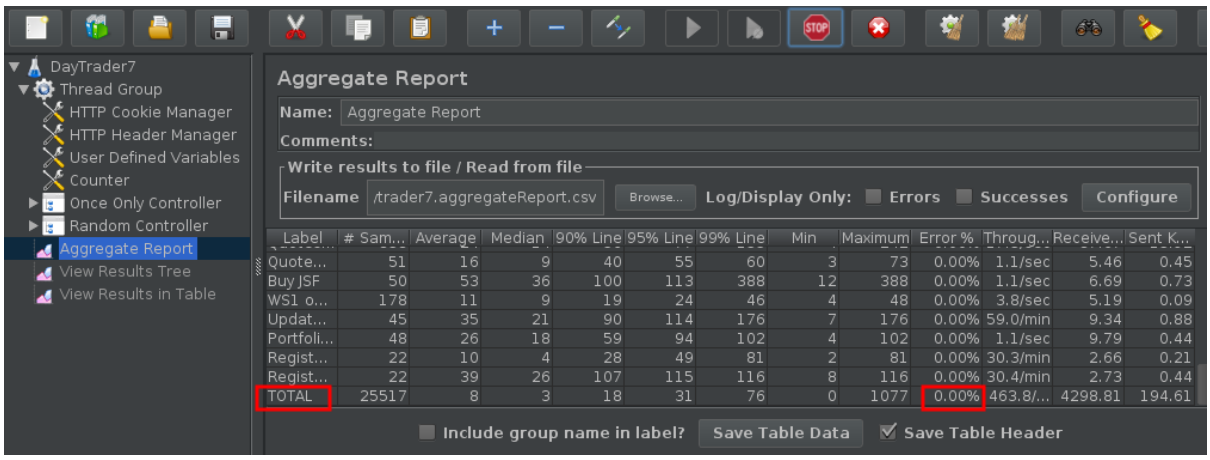
4. Click File → Open and select **/opt/daytrader7/jmeter_files/daytrader7.jmx**
5. By default, the script will execute 4 concurrent users. You may change this if you want (e.g. based on the number of CPUs available):



6. Click the green run button to start the stress test and click the "Aggregate Report" item to see the real-time results.



7. It will take some time for the responses to start coming back and for all of the pages to be exercised.
8. Ensure that the "Error %" value for the "TOTAL" row is always 0%. If there are any errors, review the logs.



9. Leave the test running for the remainder of the exercises, although you may use the Stop button

to reduce your CPU load and start the test again before each exercise.

4 Basics

First, we'll start with the basics that should be checked for most problems and performance issues:

1. Operating system CPU and memory usage
2. Thread dumps
3. Garbage Collection

5 WebSphere Linux Performance and Hang MustGather

IBM WebSphere Support provides a script called **linperf.sh** as part of the document, "MustGather: Performance, hang, or high CPU issues with WebSphere Application Server on Linux"[17] (similar scripts exist for other operating systems). This script should be pre-installed on all machines where you run WAS and it should be run when you have performance or hang issues and the resulting files should be uploaded when you open such a support case with IBM.

The linperf.sh script is pre-installed in the lab image at **/opt/linperf/linperf.sh**. In this exercise, you will run this script and analyze the output. The script demonstrates key Linux performance tools that are generally useful whether you decide to run this tool or use the commands individually.

First, let's discuss what this script does at a high level:

1. The script is executed with a set of process IDs (PIDs) of the suspect WAS processes.
2. The script gathers the output of the **netstat** command. This produces a snapshot of all active TCP and UDP network sockets.
3. The script gathers the output of the **top** command for the duration of the script (default 4 minutes). This produces periodic snapshots of a summary of system resources (CPU, memory, etc.) and the CPU usage details of the top *processes* using CPU.
4. The script gathers the output of the **top -H** command for each specified PID for the duration of the script. This produces periodic snapshots of a summary of system resources and the CPU usage details of the top *threads* using CPU in each PID.
5. The script gathers the output of the **vmstat** command for the duration of the script. This produces periodic snapshots of a summary of system resources. This is similar to the top command.
6. The script periodically requests a thread dump for each specified PID (default every 30 seconds). This produces detailed information on the Java process such as the threads and what they're doing.
7. The script gathers the output of the **ps** command for each specified PID on the same interval as the thread dumps. This produces detailed information on the command line of each PID and other resource utilization details. This is similar to the top command.

Now, let's run the script:

1. Make sure that the JMeter test is running.
2. Open a terminal on the lab image.
3. First, we'll need to find the PID(s) of WAS. There are a few ways to do this, and you only need to choose one method:
    a. Show all processes (**ps -elf**), search for the process using something unique in its command line (**grep defaultServer**), exclude the search command itself (**grep -v grep**), and then select the fourth column (in bold below):

```
$ ps -elf | grep defaultServer | grep -v grep
4 S was       1567     1 99  80   0 - 802601 -     19:26 pts/1    00:03:35 java -
javaagent:/opt/ibm/wlp/bin/tools/ws-javaagent.jar -Djava.awt.headless=true -
Xshareclasses:name=liberty,nonfatal,cacheDir=/output/.classCache/ -jar
/opt/ibm/wlp/bin/tools/ws-server.jar defaultServer
```

[17] https://www-01.ibm.com/support/docview.wss?uid=swg21115785

b. Search for the process using something unique in its command line using **pgrep -f**:

```
$ pgrep -f defaultServer
1567
```

4. Execute the **linperf.sh** command and pass the PID gathered above (replace 1567 with your PID):

```
$ /opt/linperf/linperf.sh 1567
Tue Apr 23 19:29:26 UTC 2019 MustGather>> linperf.sh script starting [...]
```

5. Wait for 4 minutes for the script to finish:

```
[...]
Tue Apr 23 19:33:33 UTC 2019 MustGather>> linperf.sh script complete.
Tue Apr 23 19:33:33 UTC 2019 MustGather>> Output files are contained within ----> 
linperf_RESULTS.tar.gz.   <----
Tue Apr 23 19:33:33 UTC 2019 MustGather>> The javacores that were created are NOT 
included in the linperf_RESULTS.tar.gz.
Tue Apr 23 19:33:33 UTC 2019 MustGather>> Check the <profile_root> for the 
javacores.
Tue Apr 23 19:33:33 UTC 2019 MustGather>> Be sure to submit linperf_RESULTS.tar.gz, 
the javacores, and the server logs as noted in the MustGather.
```

6. As mentioned at the end of the script output above, the resulting **linperf_RESULTS.tar.gz** does not include the thread dumps from WAS. Move them over to the current directory:

```
mv /opt/ibm/wlp/output/defaultServer/javacore.* .
```

At this point, if you were creating a support case, you would upload **linperf_RESULTS.tar.gz**, **javacore***, and all the WAS logs; however, instead, we will analyze the results to learn about these basic Linux performance tools:

1. Extract **linperf_RESULTS.tar.gz**:

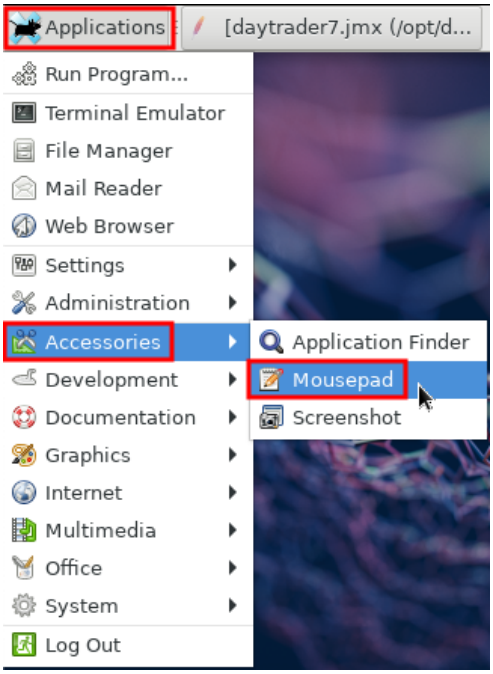
```
tar xzf linperf_RESULTS.tar.gz
```

2. This will produce various ***.out** files from the various Linux utilities.

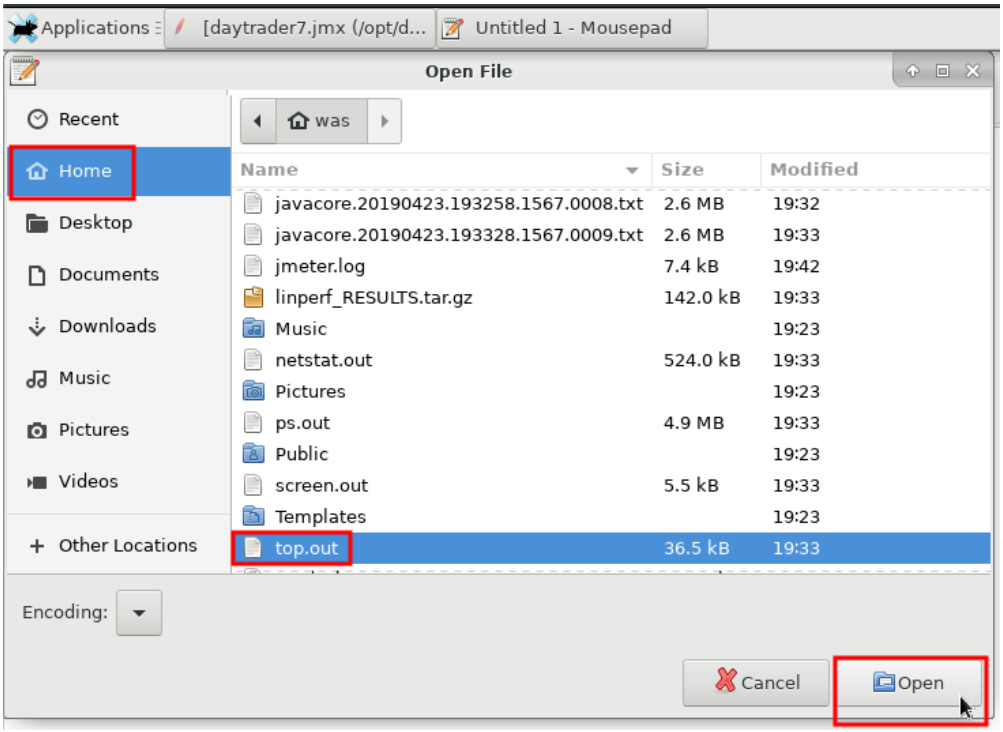
6 Linux top

top is one of the most basic Linux performance tools. Open **top.out** to review the output.

If you would like to open text files in the Linux container using a GUI tool, you may use a program such as **mousepad**:



Then click File > Open, and find the file where you ran **linperf.sh** such as in the Home directory:



There will be multiple sections of output, each prefixed with a timestamp which represents the previous interval (**linperf.sh** uses a default interval of 60 seconds). In the following example, the data represents CPU usage between 19:28:27 - 19:29:27. Review all intervals to understand CPU usage over time. For example, here is one interval:

```
Tue Apr 23 19:29:27 UTC 2019
top - 19:29:27 up  2:49,  1 user,  load average: 5.59, 2.41, 1.16
```

```
Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.7 us, 23.9 sy,  0.0 ni, 20.9 id,  1.5 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11993.4 total,    395.9 free,   1777.5 used,   9820.0 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1567 was       20   0 3216340 374372  36356 S 181.2   3.0   5:34.40 java -jav+
 1854 was       20   0 3701404 417256  24580 S  37.5   3.4   1:21.23 /usr/bin/+
  414 was       20   0  187948  19652  14296 S   6.2   0.2   0:00.22 xfsetting+
 2631 was       20   0   10676   4380   3820 R   6.2   0.0   0:00.02 top -bc -+
 2640 was       20   0   10676   4316   3756 S   6.2   0.0   0:00.01 top -bH -+
    1 root      20   0    3784   2956   2696 S   0.0   0.0   0:00.05 /bin/sh /+
    9 root      20   0   23916  21112   7532 S   0.0   0.2   0:00.31 /usr/bin/+
   13 root      20   0  151676   4188   3684 S   0.0   0.0   0:00.01 /usr/sbin+
   14 root      20   0    9264   5852   5184 S   0.0   0.0   0:00.01 /usr/sbin+
   15 root      20   0    6960   3636   3148 S   0.0   0.0   0:00.00 /usr/sbin+
```

One place to start is to check the server's RAM:

```
MiB Mem :  11993.4 total,    395.9 free,   1777.5 used,   9820.0 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.8 avail Mem
```

The values may be in bytes, KB, MB, or other formats depending on various settings.

The two values in bold are the important values:

1. The first bold value on the first line shows the total amount of RAM; in this example, about 11.9GB.
2. The second bold value on the second line shows the approximate amount of RAM that is available for applications if they need it (including readily reclaimable page cache and memory slabs); in this example, about 9.8GB. Notice that the actual amount of free RAM (first line, second column, in *italics*) is only about 395MB. Linux, like most other modern operating systems, is aggressive in using RAM for various caches, primarily the file cache, to improve disk I/O speeds; however, most of this memory is reclaimable if applications demand it. Note that Linux is particularly aggressive with its default **swappiness**[18] value and in some cases it will prefer to page out application pages instead of reclaiming file cache pages. Consider setting vm.swappiness=0 for production workloads that perform little file I/O and require most of the RAM.

Next, review the server's overall CPU usage:

```
Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.7 us, 23.9 sy,  0.0 ni, 20.9 id,  1.5 wa,  0.0 hi,  0.0 si,  0.0 st
```

The value in bold is the important value. **"id"** represents the percent of time during the interval that all CPUs were idle. It is better to look at **idle%** instead of **user%**, **system%**, etc. because this ensures that you quickly capture all potential users of CPU (including I/O wait, **"nice"**d processes[19], and hypervisor stealing[20]). Subtract the **"id"** number from 100 to get the approximate total CPU usage; in this example, (100 - 29.9) ~= 70.1%.

Next, top prints a sorted list of the highest CPU-using processes:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1567 was       20   0 3216340 374372  36356 S 181.2   3.0   5:34.40 java -jav+
```

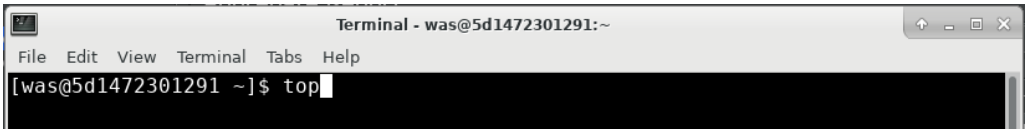
[18] https://publib.boulder.ibm.com/httpserv/cookbook/Operating_Systems-Linux.html#Swappiness
[19] **nice** and **renice** are commands to change the relative scheduling priorities of processes. Nice% reflects non-default, positively niced processes' CPU utilization.
[20] In a virtualized environment, the percent of time this host wanted CPU but waited for the hypervisor. This may mean CPU overcommit and should be reviewed.

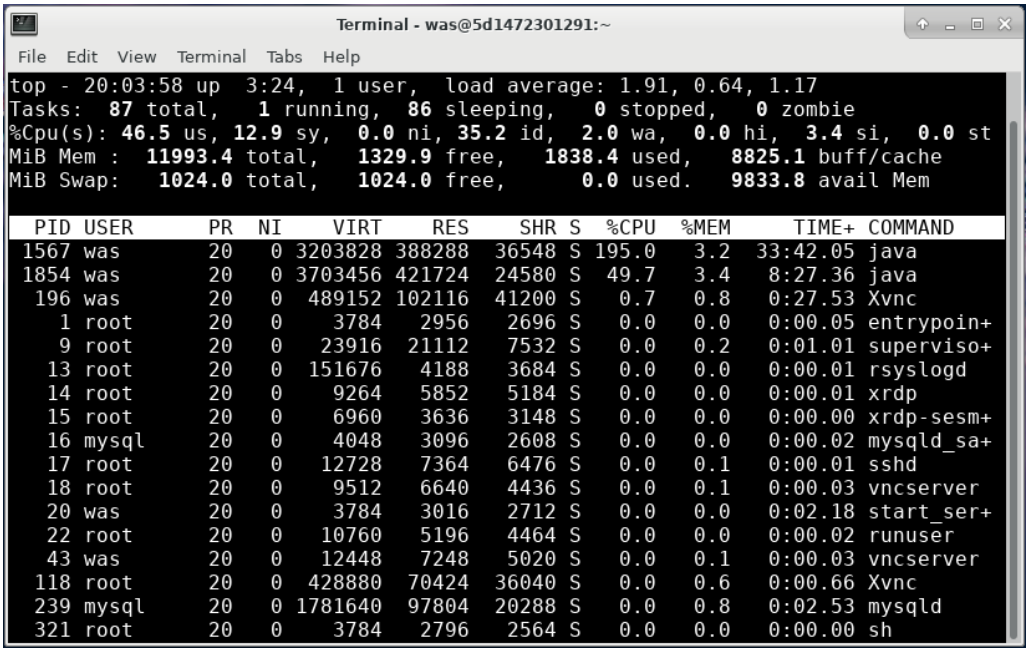
```
 1854 was       20   0 3701404 417256  24580 S  37.5   3.4   1:21.23 /usr/bin/+
  414 was       20   0  187948  19652  14296 S   6.2   0.2   0:00.22 xfsetting+ …
```

The two columns in bold are the important values:

1. The first bold column is the PID of each process which is useful for running more detailed commands.
2. The second bold column is the percent of CPU used by that PID for the interval as a percentage of one CPU. For example, PID 1567 consumed about 181.2% of one CPU which means that approximately the equivalent of 1.8 CPU threads were used. In this example, the container had 4 CPU threads available (see **/proc/cpuinfo** on your system), so PID 1567 consumed about (1.812 / 4) * 100 ~= 45.3% of total CPU.

The **top** command may be run in interactive mode by simply running the **"top"** command. This is a useful place to start when you begin investigating a system. The command will dynamically update every few seconds (this interval may be specified with the **-d S** options where **S** is in fractional seconds). Press **"q"** to quit top.





7 Linux top -H

top -H is similar to top except that the **-H** flag shows the top CPU usage by thread instead of by PID. Open **topdashH*.out** to review the output. Again, this file shows multiple intervals, so it's important to review all intervals to understand CPU usage over time. Here is an example interval:

```
Tue Apr 23 19:29:27 UTC 2019

Collected against PID 1567.

top - 19:29:27 up  2:49,  1 user,  load average: 5.59, 2.41, 1.16
Threads:  88 total,  12 running,  76 sleeping,   0 stopped,   0 zombie
%Cpu(s): 54.8 us, 19.4 sy,  0.0 ni, 24.2 id,  1.6 wa,  0.0 hi,  0.0 si,  0.0 st
```

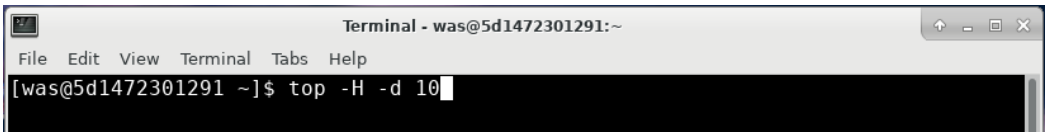
```
MiB Mem :  11993.4 total,    395.8 free,   1777.5 used,   9820.1 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1571 was       20   0 3216340 374372  36356 S  12.5   3.0   0:00.02 Signal Re+
 1638 was       20   0 3216340 374372  36356 S  12.5   3.0   0:08.09 Inbound R+
 2347 was       20   0 3216340 374372  36356 S  12.5   3.0   0:06.54 Default E+
 2386 was       20   0 3216340 374372  36356 S  12.5   3.0   0:04.94 Default E+
 2406 was       20   0 3216340 374372  36356 S  12.5   3.0   0:04.59 Default E+
 2439 was       20   0 3216340 374372  36356 S  12.5   3.0   0:03.52 Default E+
 2514 was       20   0 3216340 374372  36356 S  12.5   3.0   0:01.58 Default E+
 2539 was       20   0 3216340 374372  36356 S  12.5   3.0   0:00.80 Default E+ …
```

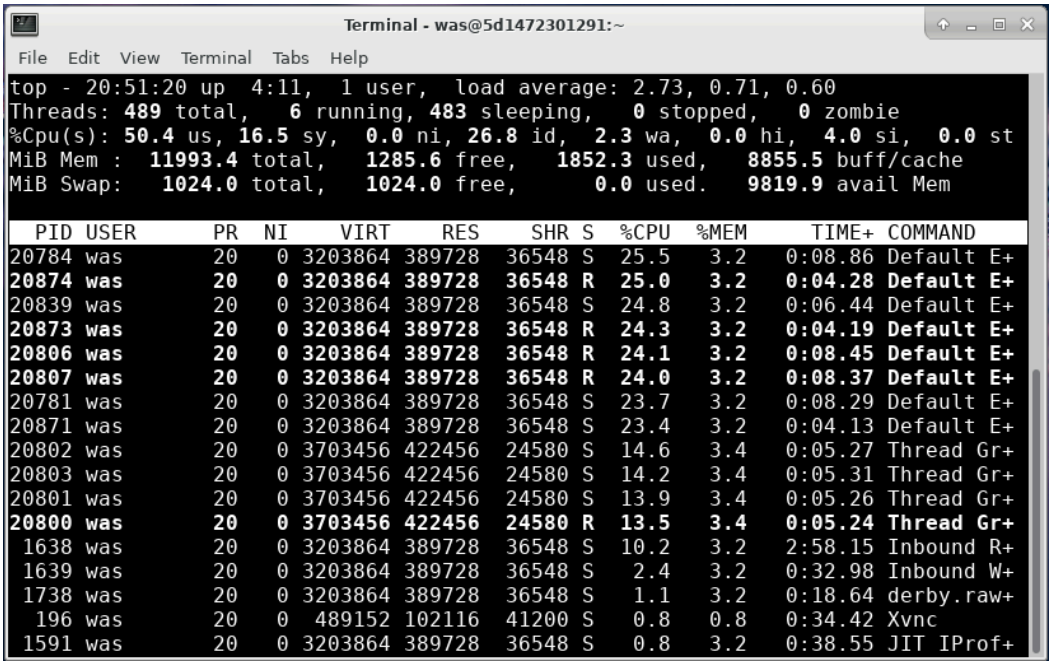
The three columns in bold are the important values:

1. The first bold column is the thread ID (TID) of each thread (the column is still called "PID" because Linux treats threads as "lightweight processes") which is useful for running more detailed commands. This value may be converted to hexadecimal and searched for in a matching thread dump.
2. The second bold column is the percent of CPU used by that TID for the interval as a percentage of one CPU (similar to the previous top output, except it's for the TID instead of the PID).
3. On recent versions of Linux, the third bold column is the name of the thread. This is incredibly useful to get a quick understanding of what threads in the Java process are consuming most of the CPU. For example:
    a. **"Default Executor"** threads are generally application threads processing HTTP and other user work on WAS Liberty,
    b. **"WebContainer"** threads are application threads processing HTTP work on Traditional WAS,
    c. **"Inbound…"** threads are WAS threads processing new inbound user requests,
    d. **"GC Slave"** threads are JVM threads processing garbage collection,
    e. **"JIT Comp…"** threads are JVM threads processing Just-in-Time (JIT) compilation,
    f. etc.

In the above example, the top threads are mostly **"Default Executor"** threads, each using about (0.125 / 4) * 100 ~= 3.125% of total CPU which means that most of the CPU usage is application threads handling user work, spread about evenly across threads.

As in the case of top, the **top -H** command may be run in interactive mode and could be considered an even better place to start when you begin investigating a system; however, note that **top -H** is much more expensive than top (especially if you don't provide a particular PID with **-p**) because it must traverse the data for all PIDs and all TIDs. Therefore, if you want to use **top -H** in interactive mode, consider using a large interval such as 10 seconds or more:





8 IBM Java and OpenJ9 Thread Dumps

Thread dumps are snapshots of process activity, including the thread stacks that show what each thread is doing. Thread dumps are one of the best places to start to investigate problems. If a lot of threads are in similar stacks, then that behavior might be an issue.

For IBM Java or OpenJ9, a thread dump is also called a javacore or javadump. HotSpot-based thread dumps are covered elsewhere[21].

This exercise will demonstrate how to review thread dumps in the free IBM Thread and Monitor Dump Analyzer (TMDA) tool[22].

An IBM Java or OpenJ9 thread dump is generated in a **javacore*.txt** in the working directory of the process with a snapshot of process activity, including:

- Each Java thread and its stack.
- A list of all Java synchronization monitors, which thread owns each monitor, and which threads are waiting for the lock on a monitor.
- Environment information, including Java command line arguments and operating system

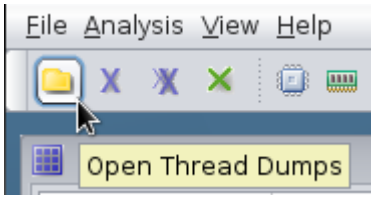
[21] https://publib.boulder.ibm.com/httpserv/cookbook/Troubleshooting-Troubleshooting_Java-Troubleshooting_Oracle_Java.html#Troubleshooting-Troubleshooting_Oracle_Java-Thread_Dump
[22] https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=2245aa39-fa5c-4475-b891-14c205f7333c

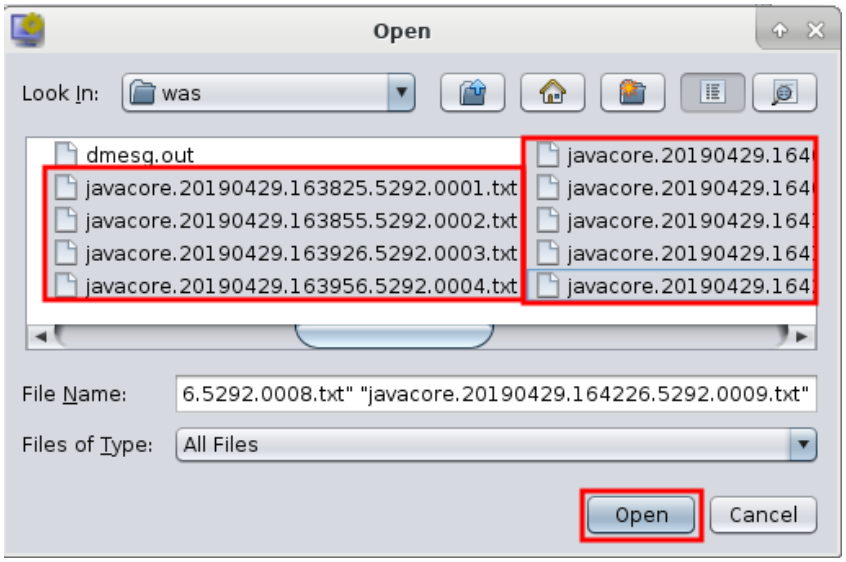
ulimits.

- Java heap usage and information about the last few garbage collections.
- Detailed native memory and classloader information.

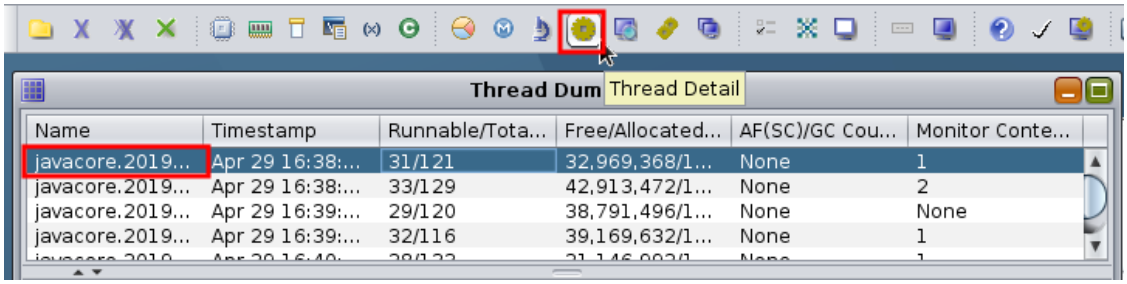
We will review the thread dumps gathered by linperf.sh above:

1. Open /opt/programs/ in the file browser and double click on **TMDA**:
2. Click Open Thread Dumps and select all of the **javacore*.txt** files using the Shift key:



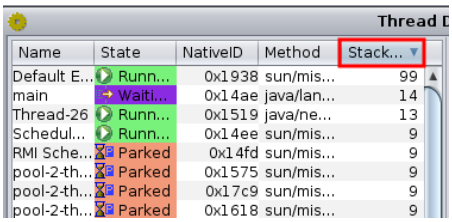


3. Select a thread dump and click the “Thread Detail” button:

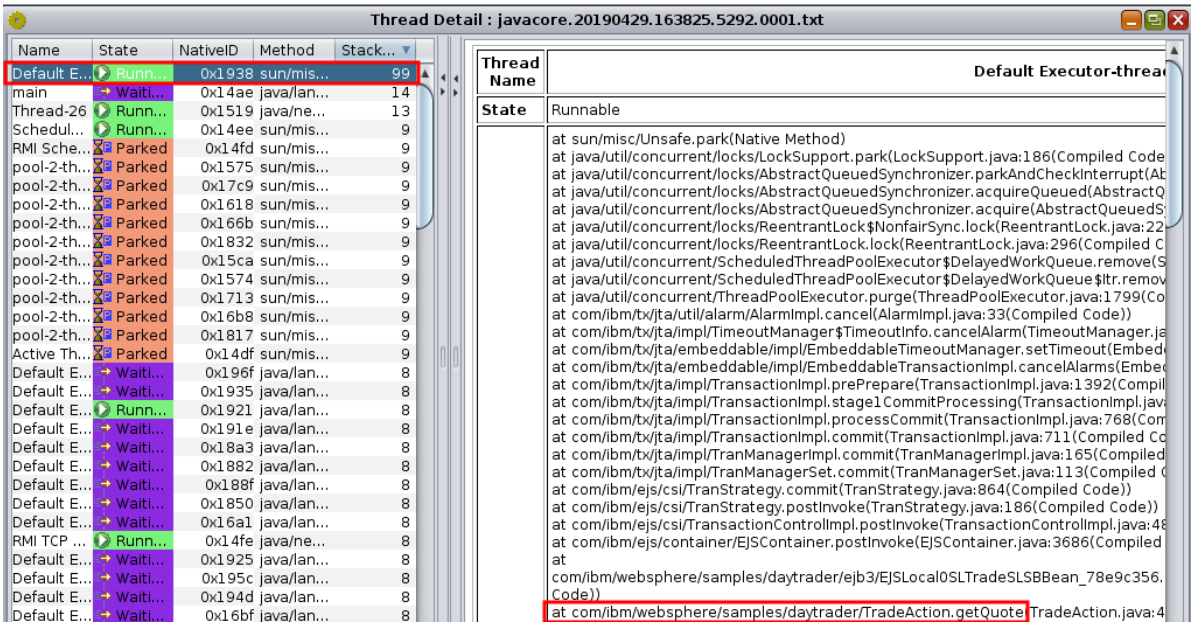


4. Click on the “Stack Depth” column to sort by thread stack depth in ascending order.

5. Click on the “Stack Depth” column again to sort again in descending order:

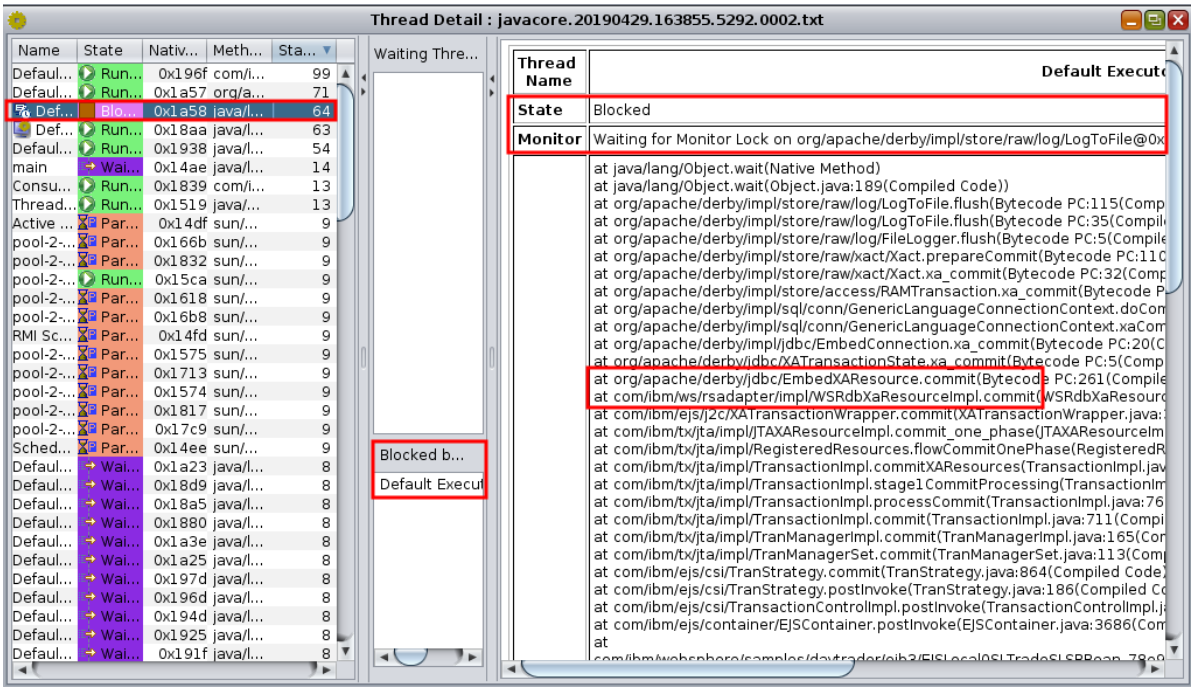


6. Generally, the threads of interest are those with stack depths greater than ~20. Select any such rows and review the stack on the right (if you don’t see any, then close this thread dump and select another from the list):



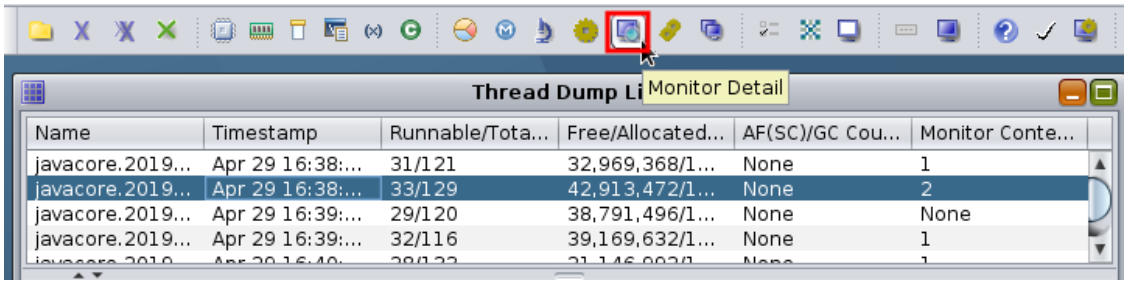
1. Generally, to understand which code is driving the thread, skip any non-application stack frames. In the above example, the first application stack frame is TradeAction.getQuote.
2. Thread dumps are simply snapshots of activity, so just because you capture threads in some stack does not mean there is necessarily a problem. However, if you have a large number of thread dumps, and an application stack frame appears with high frequency, then this may be a problem or an area of optimization. You may send the stack to the developer of that component for further research.

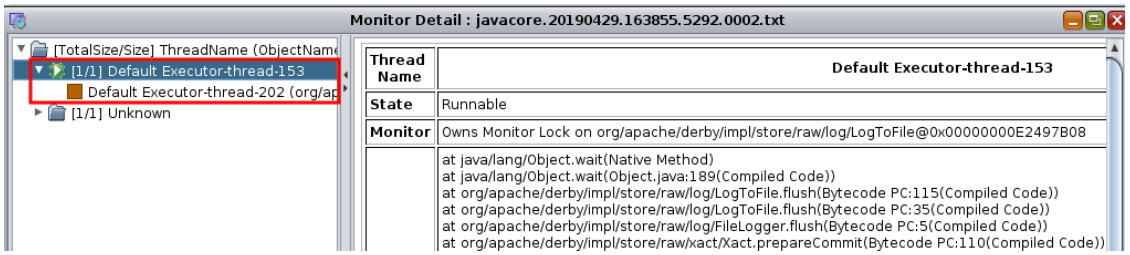
7. In some cases, you may see that one thread is blocked on another thread. For example:



1. The “Monitor” line shows which monitor this thread is waiting for, and the stack shows the path to the request for the monitor. In this example, the application is trying to commit a database transaction. This lab uses the Apache Derby database engine which is not a very scalable database. In this example, optimizing this bottleneck may not be easy and may require deep Apache Derby expertise.
2. You may click on the thread name in the “Blocked by” view to quickly see the thread stack of the other thread that owns the monitor.
3. Lock contention is a common cause of performance issues and may manifest with poor performance and low CPU usage.

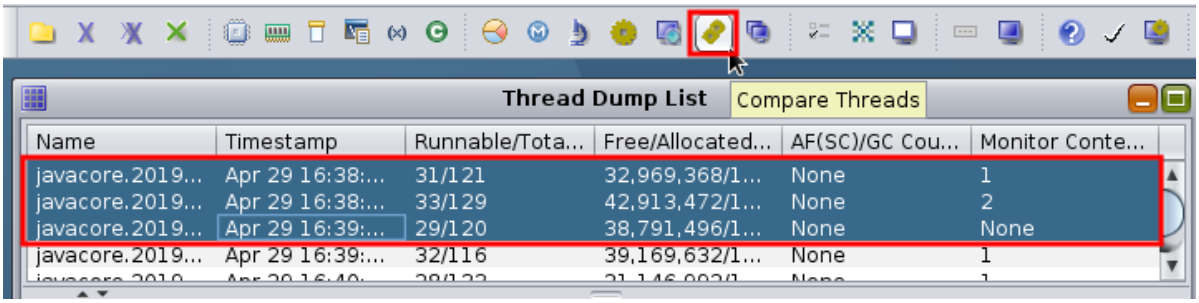
8. An alternative way to review lock contention is by selecting a thread dump and clicking "Monitor Detail":

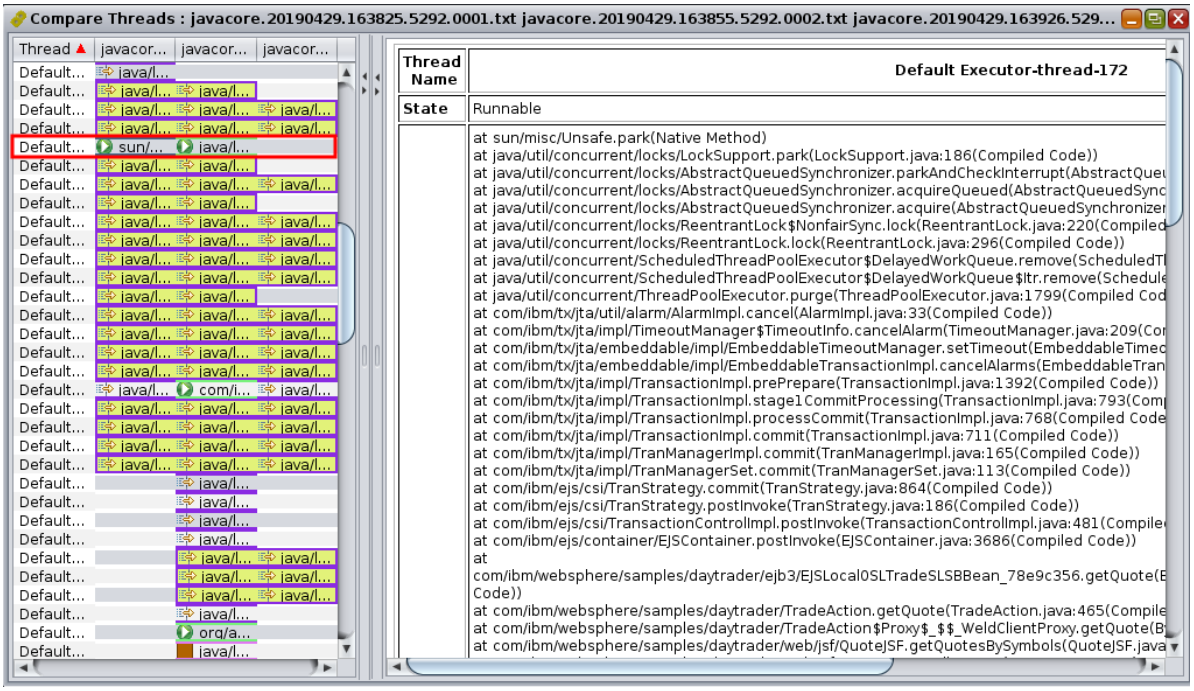




1. This shows a tree view of the monitor contention which makes it easier to explore the relationships and number of threads contending on monitors. In the above example, "Default Executor-thread-153" owns the monitor and "Default Executor-thread-202" is waiting for the monitor.

9. You may also select multiple thread dumps and click the “Compare Thread Dumps” button to see thread movement over time:





1. Each column is a thread dump and shows the state of each thread (if available) over time. Generally, you’re interested in thread that are runnable (Green Arrow) or otherwise in the same concerning top stack frame. Click on each cell in that row and review the thread dump on the right. If the thread dump is always in the same stack, this is a potential issue. If the thread stack is changing a lot, then this is usually normal behavior.
2. In general, focus on the main application thread pools such as DefaultExecutor, WebContainer, etc.

Next, let’s simulate a hung thread situation and analyze the problem with thread dumps:

1. Open a browser to http://localhost:9080/swat/
2. Scroll down and click on http://localhost:9080/swat/Deadlocker:

Deadlocker (Dining Philosophers)	/Deadlocker	None	Attempt to create a deadlock with an algorithm that emulates the Dining Philosophers problem. You will know a deadlock has occurred if messages stop being written to the HTML output. To confirm if a deadlock has occurred, take a javadump and search for "deadlock." It is possible, based on CPU availability, OS timing, and thread dispatching that a true deadlock will not occur.

3. Gather a thread dump of the Liberty process by sending it the SIGQUIT (3) signal. Although the name of the signal includes the word “QUIT”, the signal is captured by the JVM, the JVM

pauses for a few hundred milliseconds to produce the thread dump, and then the JVM continues. This same command is performed by linperf.sh. It is a quick and cheap way to quickly understand what your JVM is doing:

```
kill -3 $(pgrep -f defaultServer)
```

a. Note that here we are using a sub-shell to send the output of the pgreg command (which finds the PID of defaultServer) as the argument for the kill command.

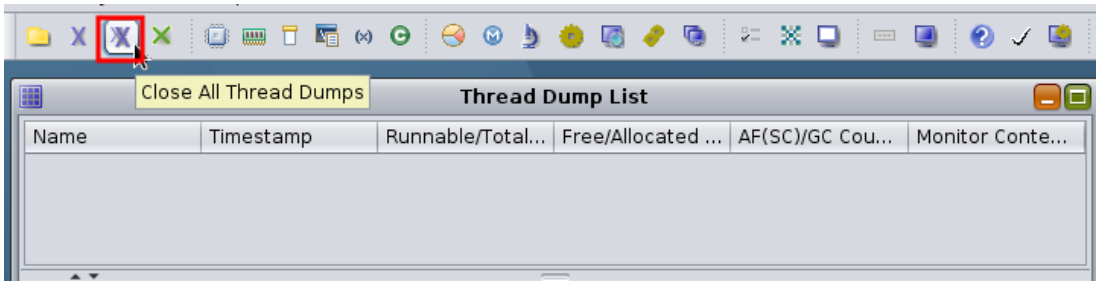
b. This can be simplified even further with the pkill command which combines pgrep functionality:

```
pkill -3 -f defaultServer
```

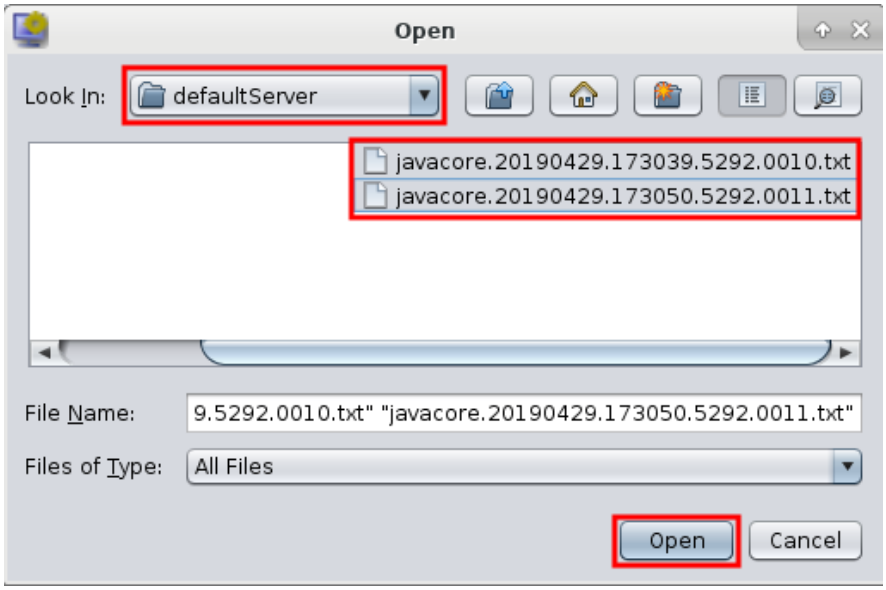
4. Wait about 30 seconds and perform the same command:

```
pkill -3 -f defaultServer
```

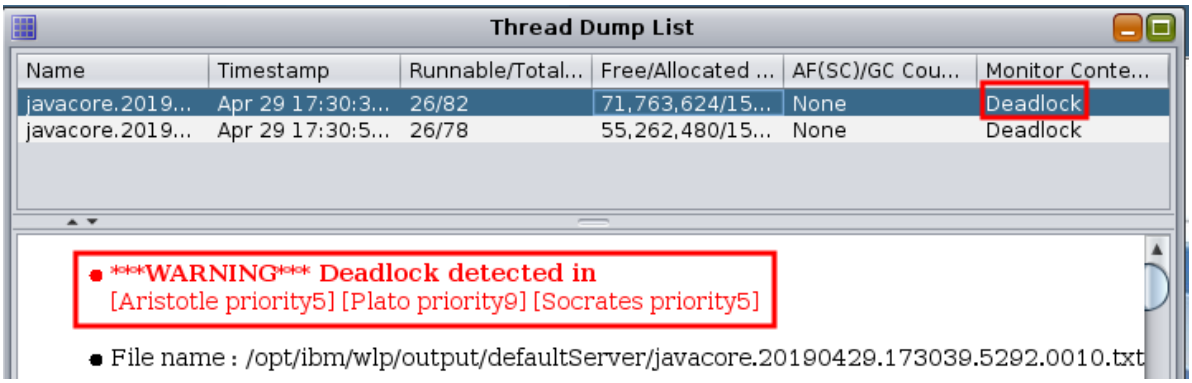
5. In the TMDA tool, clear the previous list of thread dumps:



6. Click File > Open Thread Dumps and navigate to /opt/ibm/wlp/output/defaultServer and select both thread dumps and click Open:

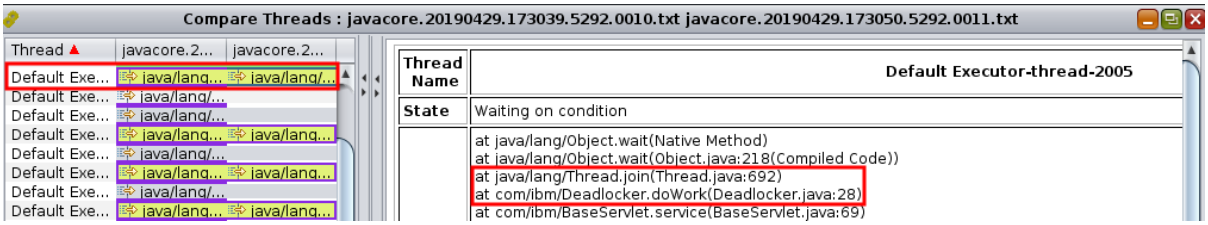


7. When you select the first thread dump, TMDA will warn you that a deadlock has been detected:

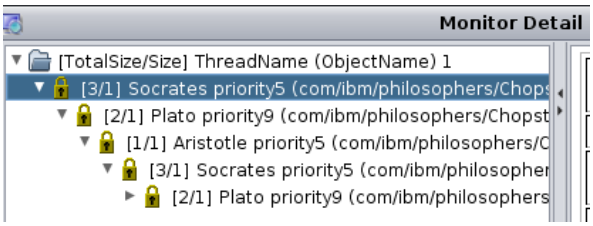


   a. Deadlocks are not common and mean that there is a bug in the application or product.

8. Use the same procedure as above to review the Monitor Details and Compare Threads to find the thread that is stuck. In this example, the DefaultExecutor application thread actually spawns threads and waits for them to finish, so the application thread is just in a Thread.join:

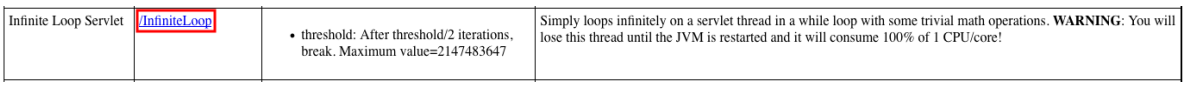


9. The actual spawned threads are named differently and show the blocking:



Next, let's combine what we've learned about the **top -H** command and thread dumps to simulate a thread that is using a lot of CPU:

1. Go to http://localhost:9080/swat/
2. Scroll down and click on http://localhost:9080/swat/InfiniteLoop:

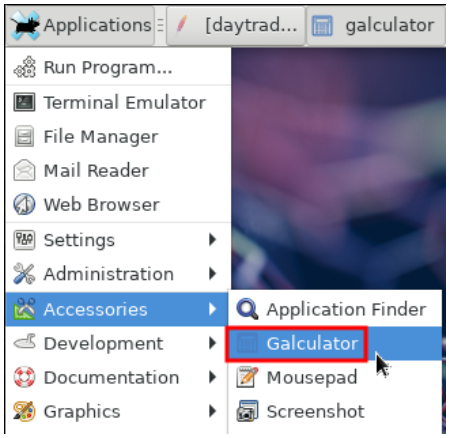


3. Go to the container terminal and start **top -H** with a 10 second interval:

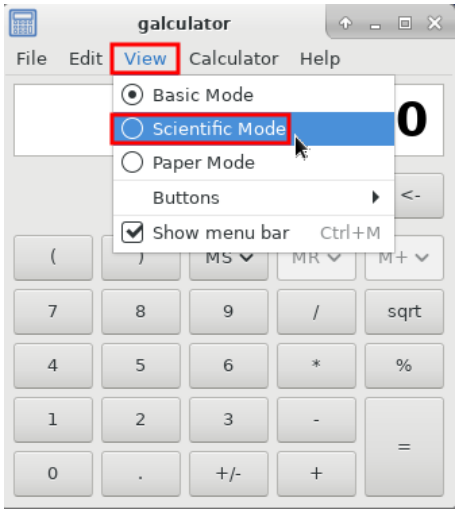
```
top -H -d 10
```

```
top - 17:48:13 up  2:34,  0 users,  load average: 0.92, 0.42, 0.28
Threads: 485 total,   2 running, 483 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25.6 us,  0.3 sy,  0.0 ni, 74.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11993.4 total,   1454.5 free,   1774.4 used,   8764.5 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.9 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
22129 was       20   0 3183240 243412  34792 R  99.9   2.0   1:41.74 Default E+
22007 was       20   0 3183240 243412  34792 S   0.6   2.0   0:00.64 JIT Sampl+
  161 was       20   0  494588 106308  40600 S   0.3   0.9   0:34.47 Xvnc
```

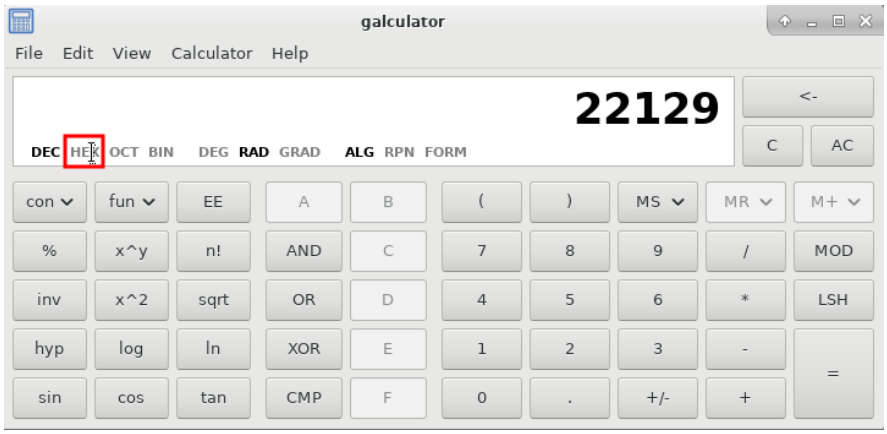
4. Notice that a single thread is consistently consuming ~100% of a single CPU thread.
5. Convert the PID to hexadecimal. In the example above, 22129 = 0x5671.
    a. In the container, open Galculator:



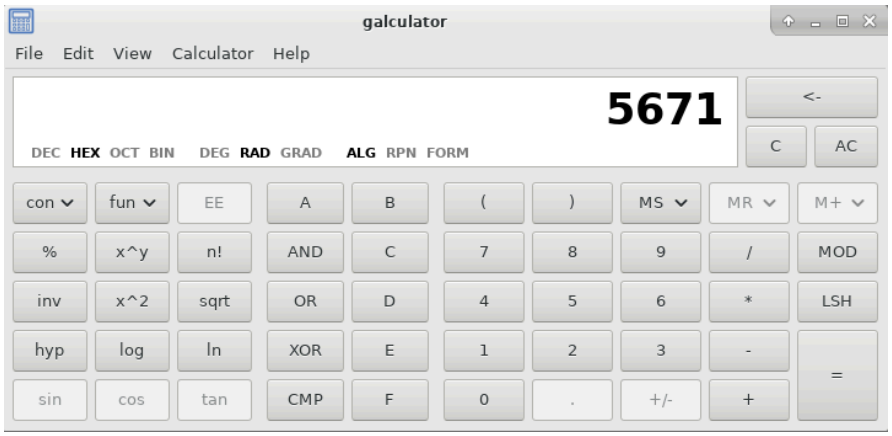
    b. Click View > Scientific Mode:



c. Enter the decimal number (in this example, 22129), and then click on HEX:



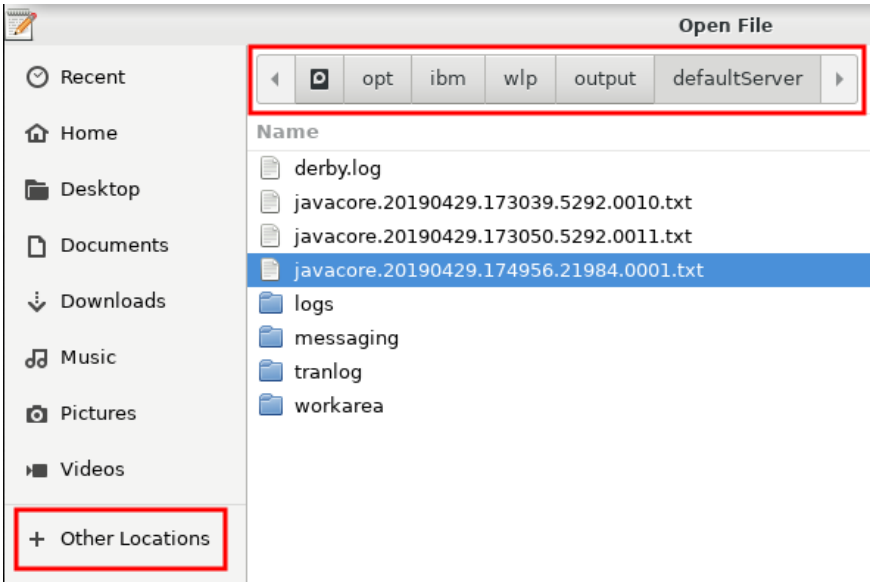
d. The result is 0x5671:



6. Take a thread dump of the parent process:

```
pkill -3 -f defaultServer
```

7. Open the most recent thread dump from **/opt/ibm/wlp/output/defaultServer/** in a text editor such as **mousepad**:



8. Search for the native thread ID in hex (in this example, 0x5671) to find the stack trace consuming the CPU (if captured during the thread dump):

```
3XMTHREADINFO      "Default Executor-thread-16" J9VMThread:0x0000000001D87000, omrthread_t:0x00007F8518037A60, java/lang/Thread:
3XMJAVALTHREAD            (java/lang/Thread getId:0x5C, isDaemon:true)
3XMTHREADINFO1            (native thread ID:0x5671, native priority:0x5, native policy:UNKNOWN, vmstate:CW, vm thread flags:0x00
3XMTHREADINFO2            (native stack address range from:0x00007F858C73B000, to:0x00007F858C77B000, size:0x40000)
3XMCPUTIME                CPU usage total: 205.198890739 secs, current category="Application"
3XMHEAPALLOC              Heap bytes allocated since last GC cycle=0 (0x0)
3XMTHREADINFO3            Java callstack:
4XESTACKTRACE                at com/ibm/InfiniteLoop.doWork(InfiniteLoop.java:27(Compiled Code))
4XESTACKTRACE                at com/ibm/BaseServlet.service(BaseServlet.java:69)
4XESTACKTRACE                at javax/servlet/http/HttpServlet.service(HttpServlet.java:790)
4XESTACKTRACE                at com/ibm/ws/webcontainer/servlet/ServletWrapper.service(ServletWrapper.java:1255)
```

9 Garbage Collection

Garbage collection (GC) automatically frees unused objects. Healthy garbage collection is one of the most important aspects of Java programs. The proportion of time spent in garbage collection versus application time should be less than 10% and ideally less than 1%[23].

This lab will demonstrate how to enable verbose garbage collection in WAS for the sample DayTrader application, exercise the application using Apache JMeter, and review verbose garbage collection data in the IBM Garbage Collection and Memory Visualizer (GCMV) tool.

All major Java Virtual Machines (JVMs) are designed to work with a maximum Java heap size. When the Java heap is full (or various sub-heaps), an allocation failure occurs and the garbage collector will run to try to find space. Verbose garbage collection (verbosegc) prints detailed information about each one of these allocation failures.

Always enable verbose garbage collection, including in production (benchmarks show an overhead of ~0.13% for IBM Java[24]), using the options to rotate the verbosegc logs. For IBM Java[25] - 5 historical files of roughly 20MB each:

```
-Xverbosegclog:verbosegc.%seq.log,5,50000
```

Add the verbosegc option to the jvm.options file:

1. Stop the JMeter test.
2. Stop the Liberty server.

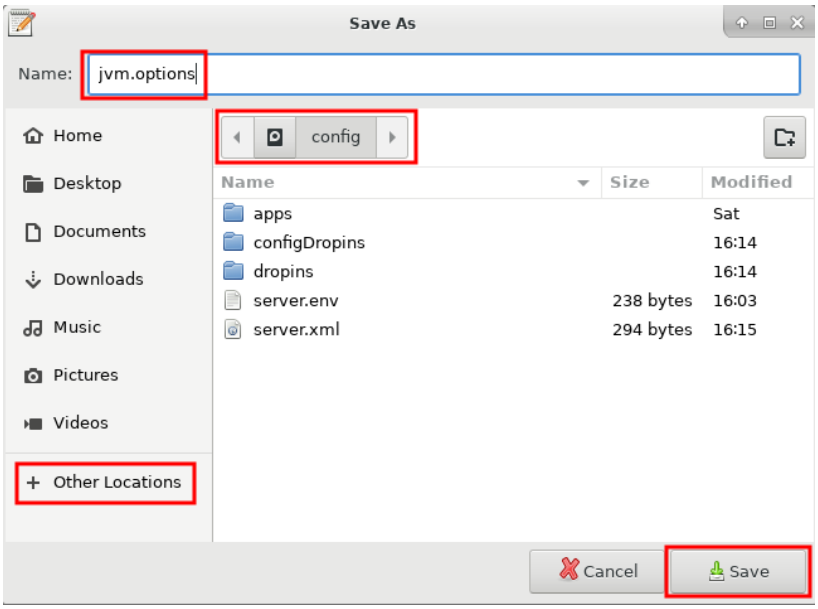
```
/opt/ibm/wlp/bin/server stop defaultServer
```

3. Open a text editor such as mousepad and add the following line to it:

```
-Xverbosegclog:logs/verbosegc.%seq.log,5,50000
```

[23] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-Garbage_Collection_and_Memory_Visualizer_GCMV.html#Major_Tools-Garbage_Collection_and_Memory_Visualizer_GCMV-Analysis
[24] https://publib.boulder.ibm.com/httpserv/cookbook/Java-IBM_Java_Runtime_Environment.html#Java-IBM_Java_Runtime_Environment-Garbage_Collection-Verbose_garbage_collection_verbosegc
[25] http://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/diag/appendixes/cmdline/xverbosegclog.html

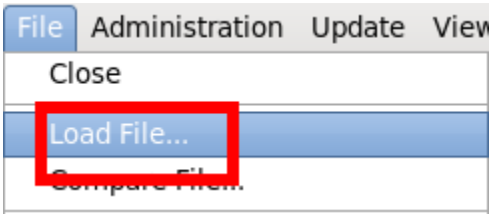
4. Save the file to /config/jvm.options



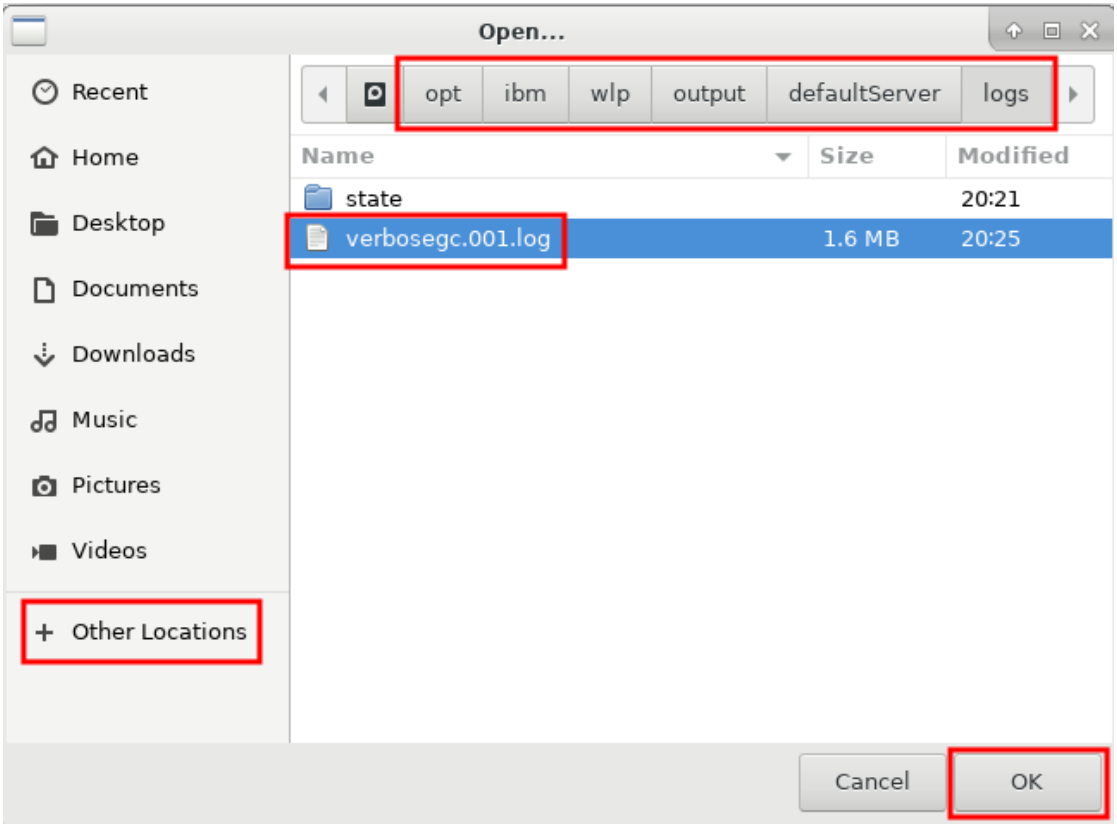
5. Start the Liberty server

```
/opt/ibm/wlp/bin/server start defaultServer
```

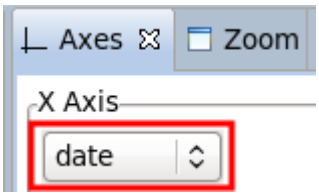
6. Start the JMeter test
7. Run the test for about 5 minutes
8. Stop the JMeter test
9. Open /opt/programs/ in the file browser and double click on **GCMV**:
10. Click File > Load File... and select the verbosegc.001.log file. For example:



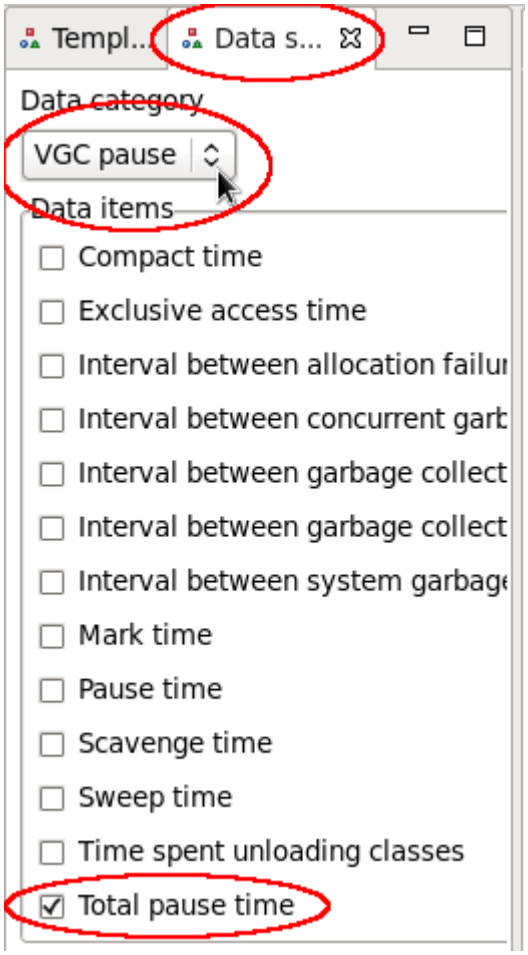
11. Select /opt/ibm/wlp/output/defaultServer/logs/verbosegc.001.log



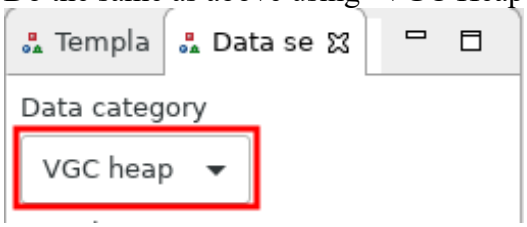
12. Once the file is loaded, you will see the default line plot view. It is common to change the X-axis to "date" to see absolute timestamps:

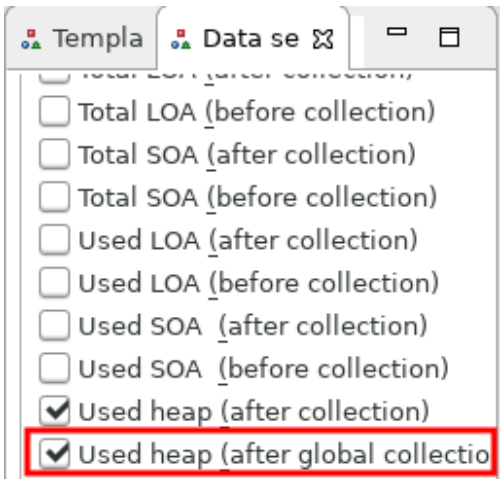


13. Click the "Data Selector" tab in the top left, choose "VGC Pause" and check "Total pause time" to add the total garbage collection pause time plot to the graph:

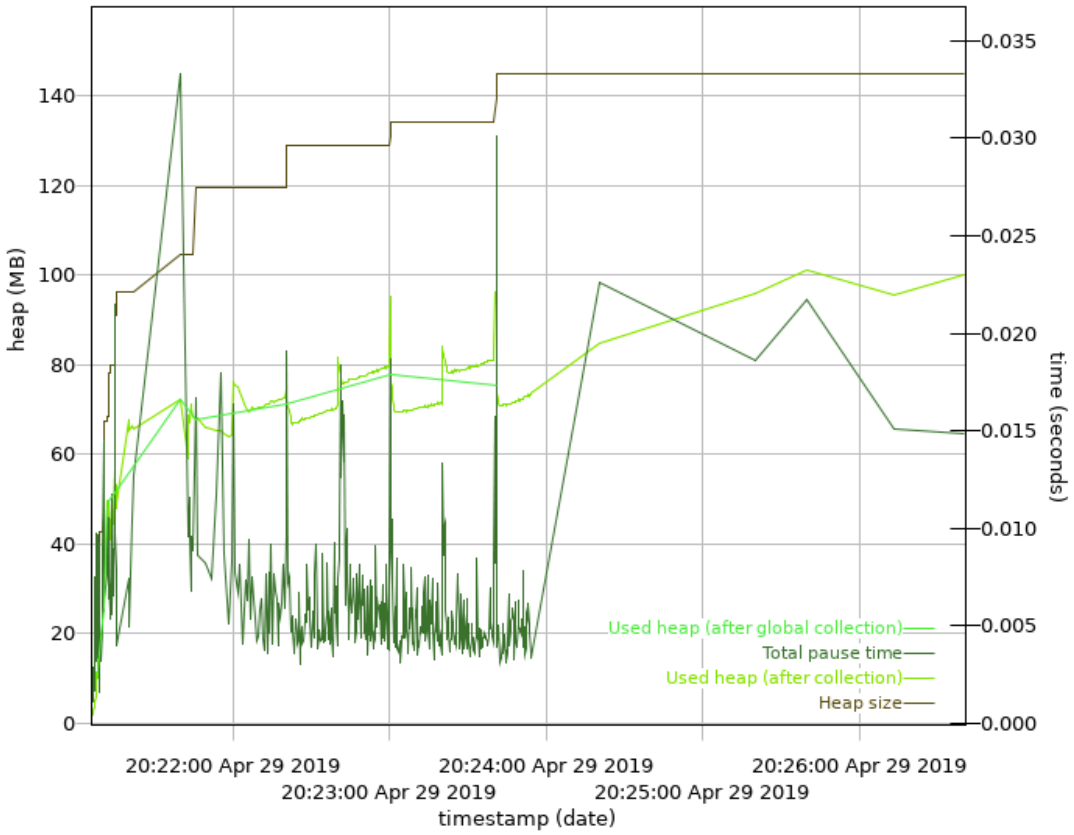


14. Do the same as above using "VGC Heap" and check "Used heap (after global collection)":





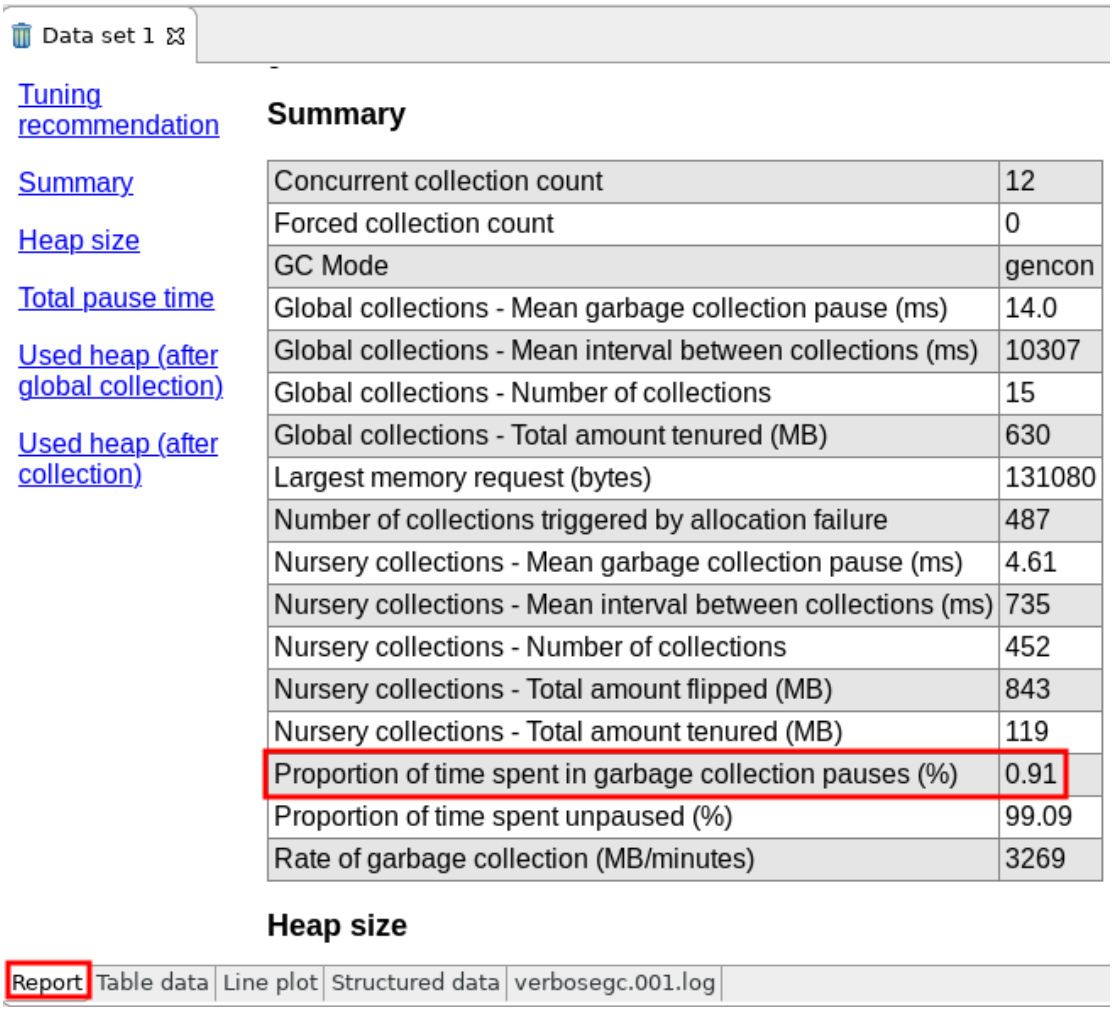
15. Observe the heap usage and pause time magnitude and frequency over time. For example:



1. This shows that the heap size reaches 145MB and the heap usage (after global collection)

reached ~80MB.

16. More importantly, we want to know the proportion of time spent in GC. Click the "Report" tab and review the "Proportion of time spent in garbage collection pauses (%)":



1. If this number is less than 1%, then this is very healthy. If it's less than 5% than it's okay. If it's less than 10%, then there is significant room for improvement. If it's greater than 10%, then this is concerning.

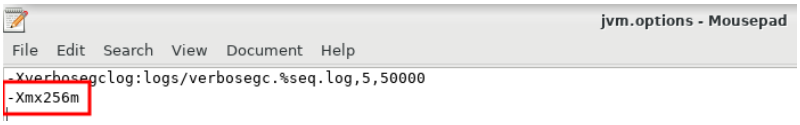
Next, let's simulate a memory issue.

1. Stop the JMeter test.
2. Stop Liberty:

```
/opt/ibm/wlp/bin/server stop defaultServer
```

3. Edit /config/jvm.options, add an explicit maximum heap size of 256MB on a new line and save the file:

```
-Xmx256m
```



4. Start Liberty

```
/opt/ibm/wlp/bin/server start defaultServer
```

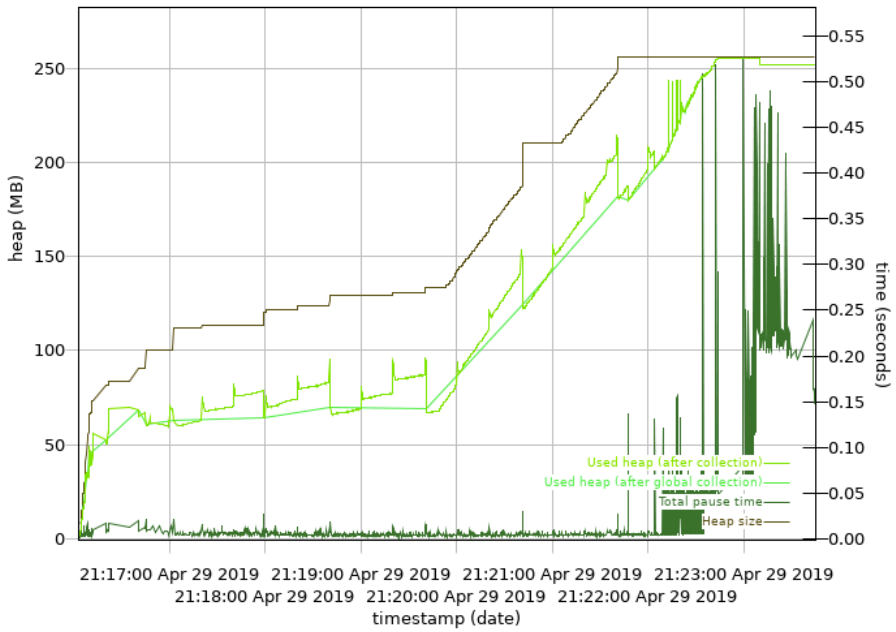
5. Start the JMeter test.
6. Open your browser to http://localhost:9080/swat/AllocateObject?size=1048576&iterations=300&waittime=1000&retainData=true
    1. This will allocate three hundred 1MB objects with a delay of 1 second between each allocation, and hold on to all of them to simulate a leak.
    2. This will take about 5 minutes to run and you can watch your browser output for progress.
    3. You can run **top -H** while this is running. As memory pressure builds, you'll start to see "GC Slave" threads consuming most of the CPU instead of application threads (garbage collection also happens on the thread where the allocation failure occurs, so you may also see a single application thread consuming a similar amount of CPU as the GC Slave threads):

```
top -H -p $(pgrep -f defaultServer) -d 5
```

```
top - 21:23:12 up  6:09,  0 users,  load average: 3.82, 3.66, 2.14
Threads:  81 total,   1 running,  80 sleeping,   0 stopped,   0 zombie
%Cpu(s): 39.1 us,  1.7 sy,  0.0 ni, 59.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11993.4 total,   2105.8 free,   2583.6 used,   7304.1 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9059.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
67934 was       20   0 2953764 506200  36360 S  38.9   4.1   0:04.87 Default Executo
66515 was       20   0 2953764 506200  36360 S  25.2   4.1   0:15.55 GC Slave
66517 was       20   0 2953764 506200  36360 S  20.9   4.1   0:15.33 GC Slave
66513 was       20   0 2953764 506200  36360 S  19.9   4.1   0:15.39 GC Slave
66563 was       20   0 2953764 506200  36360 R  15.3   4.1   0:34.18 Inbound Read Se
68007 was       20   0 2953764 506200  36360 S  13.6   4.1   0:01.24 Default Executo
67549 was       20   0 2953764 506200  36360 S  13.3   4.1   0:11.35 Default Executo
```

4. At some point, browser output will stop because the JVM has thrown an OutOfMemoryError.

7. Close and re-open the verbosegc*log file in GCMV:



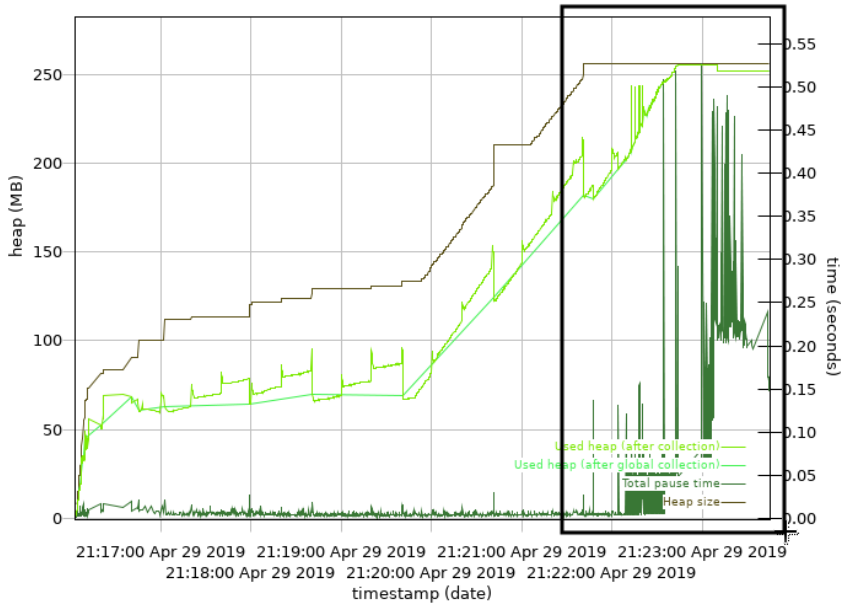
   1. We can quickly see how the heap usage reaches 256MB and the pause time magnitude and durations increase significantly.

8. Click on the “Report” tab and review the “Proportion of time spent in garbage collection pauses (%)”:

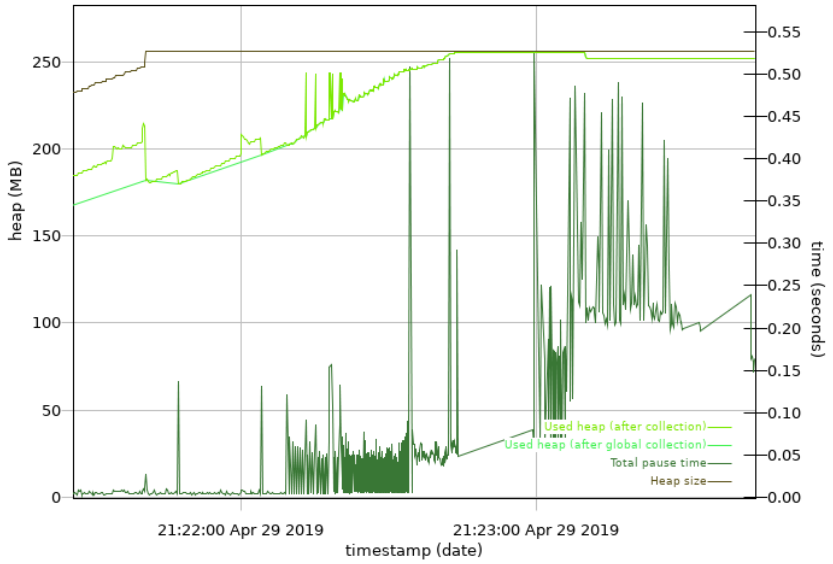
Proportion of time spent in garbage collection pauses (%)	11.29

9. At first, 11% might not seem too bad and doesn’t line up with what we know about what happened. This is because, by default, the GCMV Report tab shows statistics for the entire duration of the verbosegc log file.

10. Click on the “Line plot” tab and zoom in to the area of high pause times by using your mouse button to draw a box around those times:

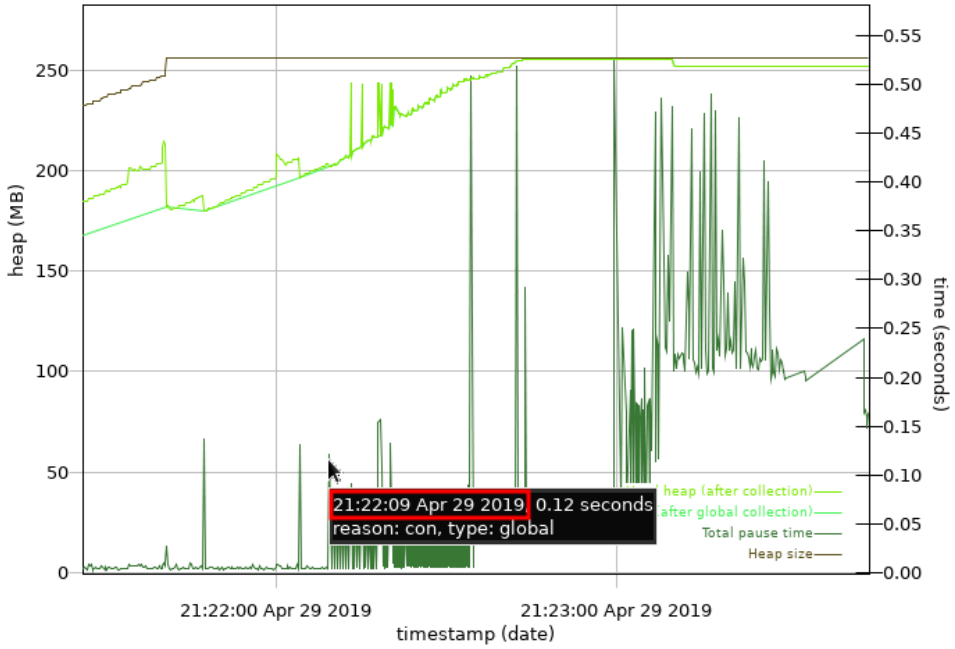


11. This will zoom the view to that bounding box:

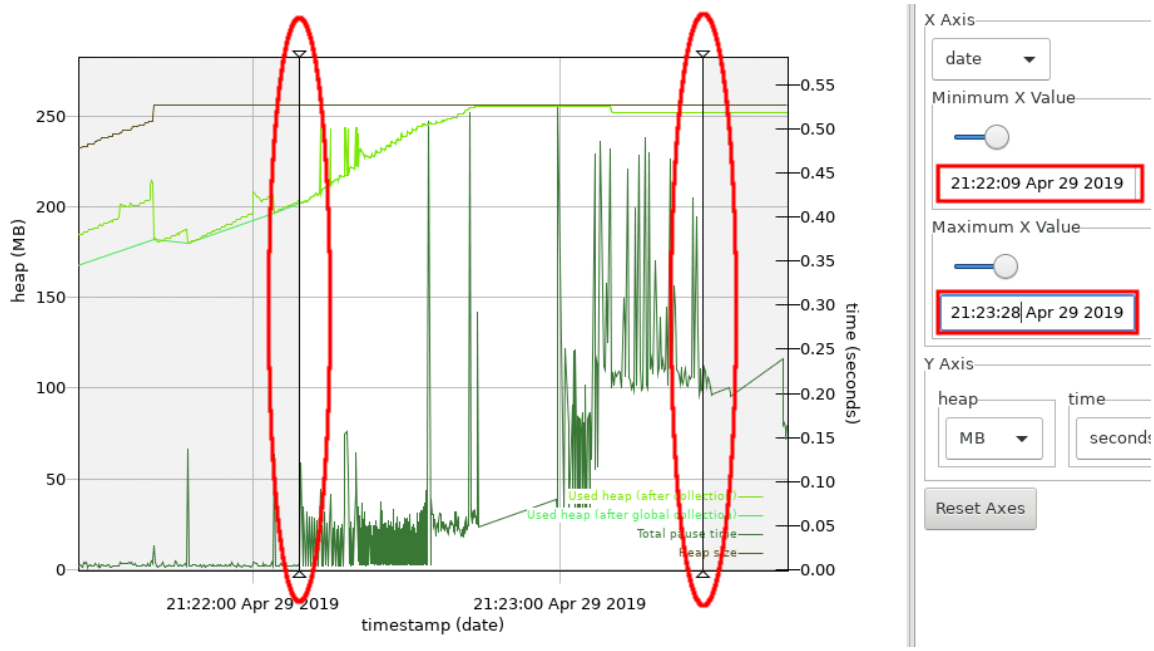


12. However, zooming in is just a visual aid. To change the report statistics, we need to match the X-axis to the period of interest.

13. Hover your mouse over the approximate start and end points of the section of concern and note the times of those points (in terms of your selected X Axis type):



14. Enter each of the values in the minimum and maximum input boxes and press **Enter** on your keyboard in each one to apply the value. The tool will show vertical lines with triangles showing the area of the graph that you've cropped to.



15. Click on the "Report" tab at the bottom and observe the proportion of time spent in garbage collection for this period is very high (in this example, ~50%).

Proportion of time spent in garbage collection pauses (%)	50.7

16. This means that the application is doing very little work and is very unhealthy. In general, there are a few, non-exclusive ways to resolve this problem:
    1. Increase the maximum heap size.
    2. Decrease the object allocation rate of the application.
    3. Resolving memory leaks through heapdump analysis.
    4. Decrease the maximum thread pool size.
17. You may increase the maximum heap size by changing the -Xmx value in jvm.options.

10 Other Topics

The above three sections – operating system CPU and memory, thread dumps, and garbage collection – are the three key elements that should be reviewed for all problems and performance issues. The rest of the lab will review other problem types and performance tuning and other types of tools.

10.1 Methodology

First, let's review some general tips about problem determination and performance methodology:

10.1.1 The Scientific Method

Troubleshooting is the act of understanding problems and then changing systems to resolve those problems. The best approach to troubleshooting is the scientific method which is basically as follows:

1. Observe and measure evidence of the problem. For example: "Users are receiving HTTP 500 errors when visiting the website."
2. Create prioritized hypotheses about the causes of the problem. For example: "I found exceptions in the logs. I hypothesize that the exceptions are creating the HTTP 500 errors."
3. Research ways to test the hypotheses using experiments. For example: "I searched the documentation and previous problem reports and the exceptions may be caused by a default setting configuration. I predict that changing this setting will resolve the problem if this hypothesis is true."
4. Run experiments to test hypotheses. For example: "Please change this setting and see if the user errors are resolved."
5. Observe and measure experimental evidence. If the problem is not resolved, repeat the steps above; otherwise, create a theory about the cause of the problem.

10.1.2 Organizing an Investigation

Keep track of a summary of the situation, a list of problems, hypotheses, and experiments/tests. Use numbered items so that people can easily reference things in phone calls or emails. The summary should be restricted to a single sentence for problems, resolution criteria, statuses, and next steps. Any details are in the subsequent tables. The summary is a difficult skill to learn, so try to constrain yourself to a single (short!) sentence. For example:

Summary

1. Problems: 1) Average website response time of 5000ms and 2) website error rate > 10%.
2. Resolution criteria: 1) Average response time of 300ms and 2) error rate of <= 1%.
3. Statuses: 1) Reduced average response time to 2000ms and 2) error rate to 5%.
4. Next steps: 1) Investigate database response times and 2) gather diagnostic trace.

Problems

#	Problem	PMR #	Status	Next Steps
1	Average response time greater than 300ms	12345,000,000	Reduced average response time to 2000ms by increasing heap size	Investigate database response times
2	Website error rate greater than 1%	67890,000,000	Reduced website error rate to 5% by fixing an application bug	Run diagnostic trace for remaining errors

Hypotheses for Problem #1

#	Hypothesis	Evidence	Status
1	High proportion of time in garbage collection leading to reduced performance	• Verbosegc showed proportion of time in GC of 20% • Increased Java maximum heap size to -Xmx1g and proportion of time in GC went down to 5%	• Further fine-tuning can be done, but at this point 5% is a reasonable number
2	Slow database response times	• Thread stacks showed many threads waiting on the database	• Gather database response times

Hypotheses for Problem #2

#	Hypothesis	Evidence	Status
1	NullPointerException in com.application.foo is causing errors	• NullPointerExceptions in the logs correlate with HTTP 500 response codes	• Application fixed the NullPointerException and error rates were halved
2	ConcurrentModificationException in com.websphere.bar is causing errors	• ConcurrentModificationExceptions correlate with HTTP 500 response codes	• Gather WAS diagnostic trace capturing some exceptions

Experiments/Tests

#	Experiment/Test	Date/Time	Environment	Changes	Results
1	Baseline	2017-01-01 09:00:00 UTC to 2017-01-01 17:00:00 UTC	Production server1	None	• Average response time 5000ms • Website error rate 10%

2	Reproduce in a test environment	2017-01-02 11:00:00 UTC to 2017-01-01 12:00:00 UTC	Test server1	None	• Average response time 8000ms • Website error rate 15%
3	Test problem #1 - hypothesis #1	2017-01-03 12:30:00 UTC to 2017-01-01 14:00:00 UTC	Test server1	Increase Java heap size to 1g	• Average response time 4000ms • Website error rate 15%
4	Test problem #1 - hypothesis #1	2017-01-04 09:00:00 UTC to 2017-01-01 17:00:00 UTC	Production server1	Increase Java heap size to 1g	• Average response time 2000ms • Website error rate 10%
5	Test problem #2 - hypothesis #1	2017-01-05	Production server1	Application bugfix	• Average response time 2000ms • Website error rate 5%
6	Test problem #1 - hypothesis #2	TBD	Test server1	Gather WAS JDBC PMI	• TBD
7	Test problem #2 - hypothesis #2	TBD	Test server1	Enable WAS diagnostic trace com.ibm.foo=all	• TBD

10.1.3 Performance Tuning Tips

1. Performance tuning is usually about focusing on a few key variables. We will highlight the most common tuning knobs that can often improve the speed of the average application by 200% or more relative to the default configuration. The first step, however, should be to use and be guided by the tools and methodologies. Gather data, analyze it and create hypotheses: then test your hypotheses. Rinse and repeat. As Donald Knuth says: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time […]. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail." (Donald Knuth, Structured Programming with go to Statements, Stanford University, 1974, Association for Computing Machinery)
2. There is a seemingly daunting number of tuning knobs. Unless you are trying to squeeze out every last drop of performance, we do not recommend a close study of every tuning option.
3. In general, we advocate a bottom-up approach. For example, with a typical WebSphere Application Server application, start with the operating system, then Java, then WAS, then the application, etc. Ideally, investigate these at the same time. The main goal of a performance tuning exercise is to iteratively determine the bottleneck restricting response times and throughput. For

example, investigate operating system CPU and memory usage, followed by Java garbage collection usage and/or thread dumps/sampling profilers, followed by WAS PMI, etc.

4. One of the most difficult aspects of performance tuning is understanding whether or not the architecture of the system, or even the test itself, is valid and/or optimal.
5. Meticulously describe and track the problem, each test and its results.
6. Use basic statistics (minimums, maximums, averages, medians, and standard deviations) instead of spot observations.
7. When benchmarking, use a repeatable test that accurately models production behavior, and avoid short term benchmarks which may not have time to warm up.
8. Take the time to automate as much as possible: not just the testing itself, but also data gathering and analysis. This will help you iterate and test more hypotheses.
9. Make sure you are using the latest version of every product because there are often performance or tooling improvements available.
10. When researching problems, you can either analyze or isolate them. Analyzing means taking particular symptoms and generating hypotheses on how to change those symptoms. Isolating means eliminating issues singly until you've discovered important facts. In general, we have found through experience that analysis is preferable to isolation.
11. Review the full end-to-end architecture. Certain internal or external products, devices, content delivery networks, etc. may artificially limit throughput (e.g. Denial of Service protection), periodically mark services down (e.g. network load balancers, WAS plugin, etc.), or become saturated themselves (e.g. CPU on load balancers, etc.).

10.2 Heap Dumps

Heap dumps are snapshots of Java objects in a process. On IBM Java, the two heapdump formats are Portable Heapdump (PHD) and System Dump (core), with a core including memory contents and thread stacks.

This lab will demonstrate how to exercise the sample DayTrader application using Apache JMeter, request a heap dump of WebSphere Application Server, and review the heapdump file in the IBM Memory Analyzer Tool (MAT)[26].

Heap dumps are used for investigating the causes of OutOfMemoryErrors, sizing applications, and reviewing memory contents under various conditions. Starting with WAS 8.0.0.2, the first OutOfMemoryError produces both a PHD and system core file.

System dumps are a superset of PHD files and are generally recommended, although they may contain sensitive customer information. The general recommendation is to always use system cores, and if security is a concern, extract a PHD file from the core using jextract for normal usage, and save the core file in case it is needed for advanced analysis.

A few key definitions:

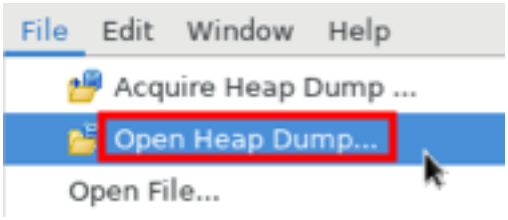
- The dominator tree is a transformation of the graph which creates a spanning tree, removes cycles, and models the keep-alive dependencies.
- The retained set of X is the set of objects which would be removed by the garbage collector when X is garbage collected.

Ensure the Liberty server is started and JMeter is running.

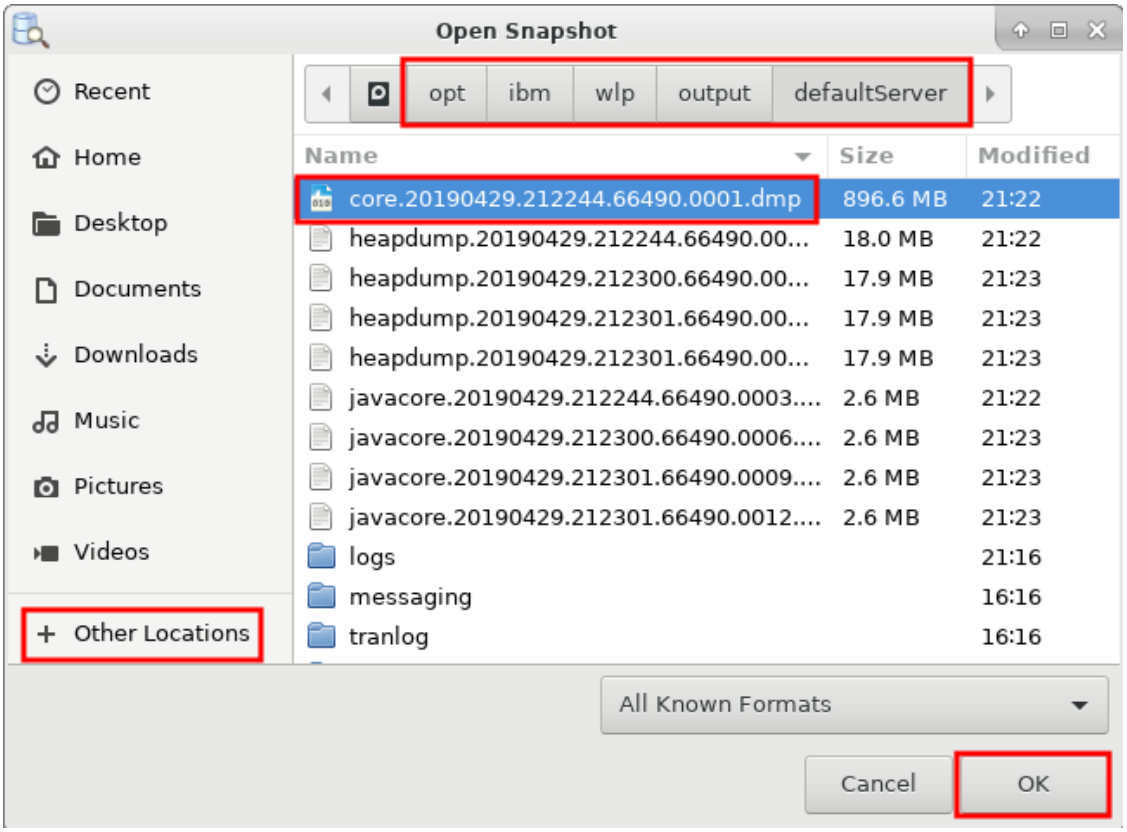
[26] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Memory_Analyzer_Tool.html#Major_Tools-IBM_Memory_Analyzer_Tool_MAT-Standalone_Installation

1. Open /opt/programs/ in the file browser and double click on **MAT**.

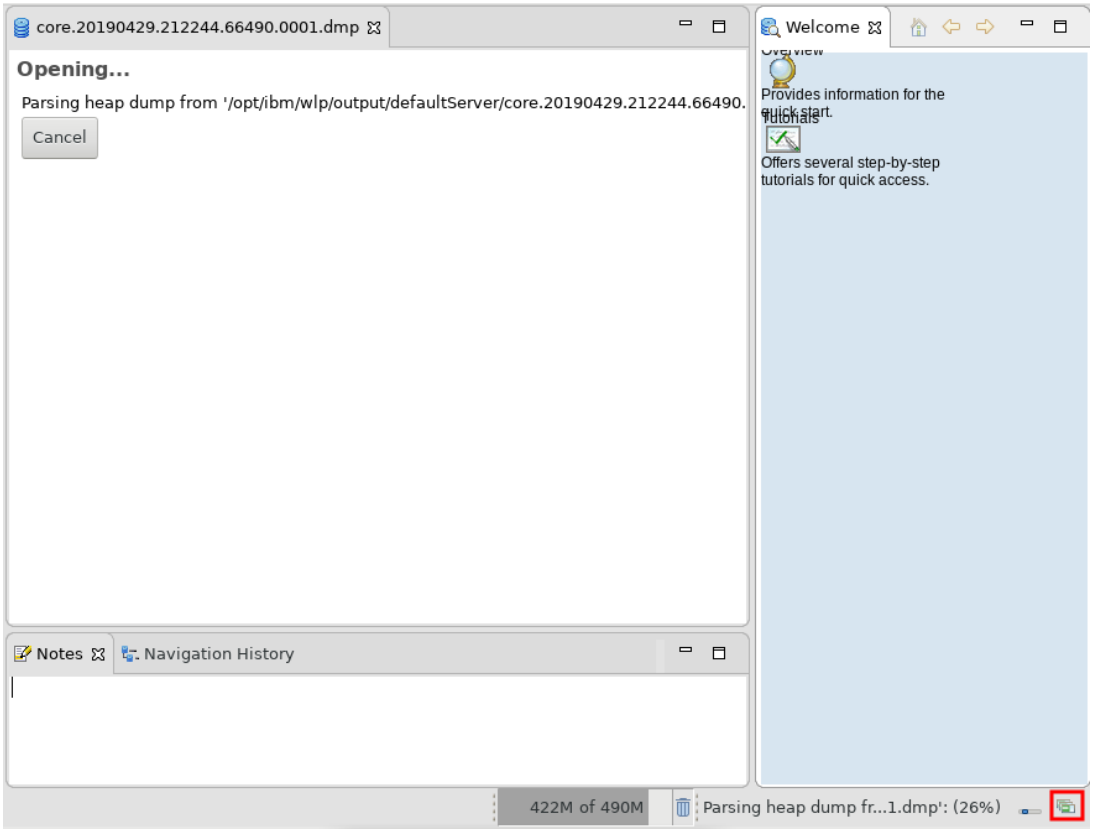
2. Click File > Open Heap Dump...



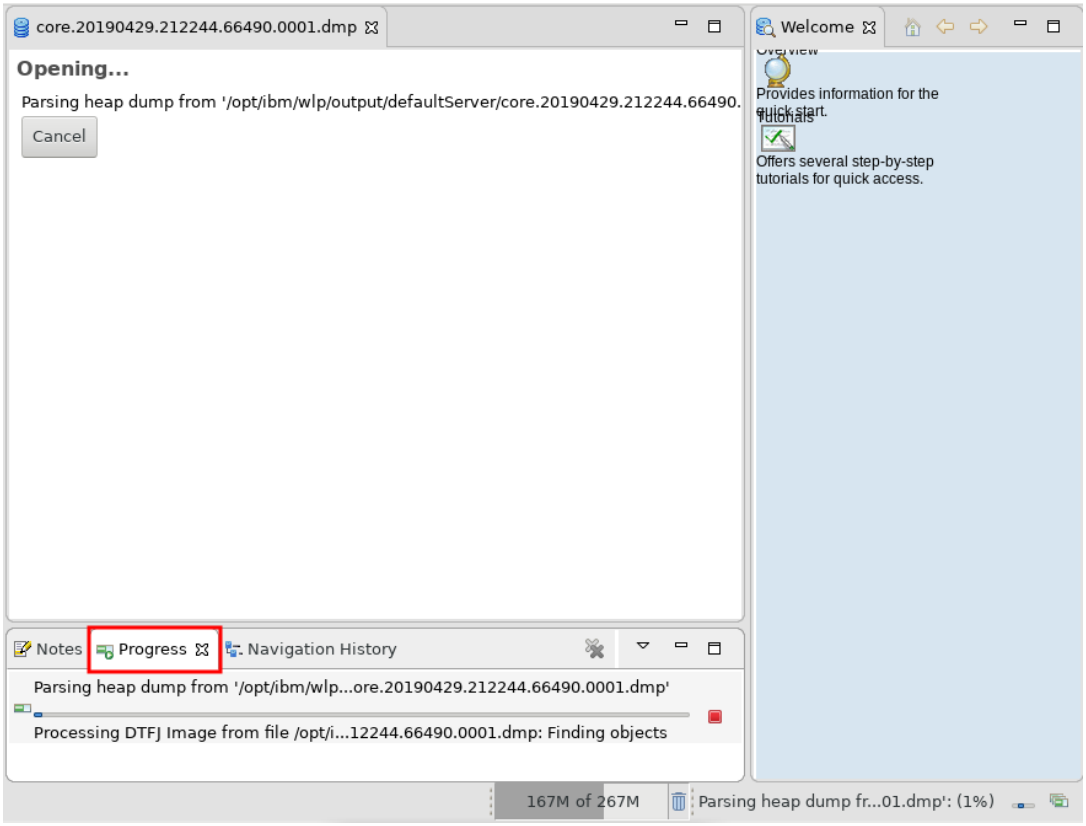
3. Select the core.*.dmp file:



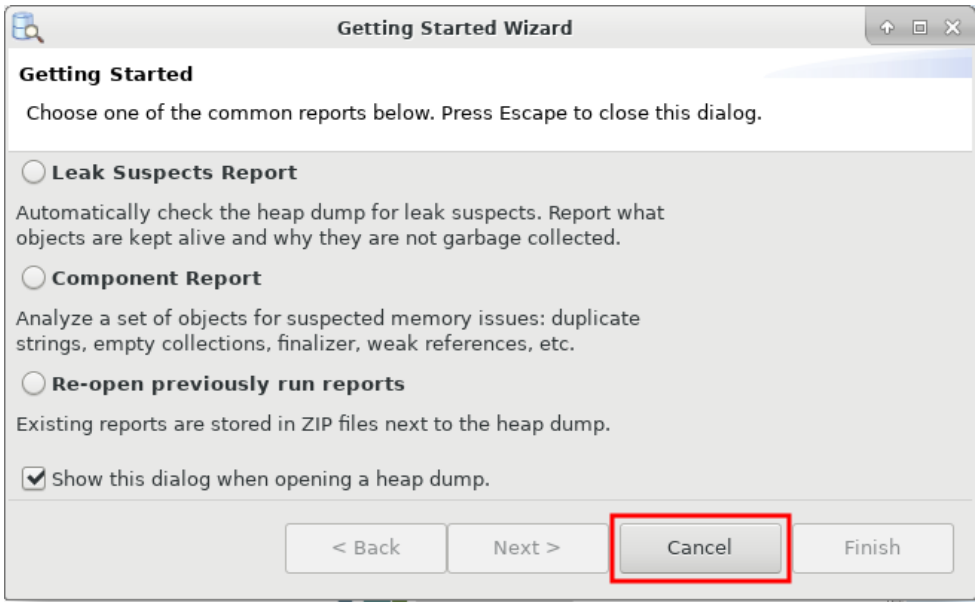
4. Click on the progress icon in the bottom right corner to get a detailed view of the progress:



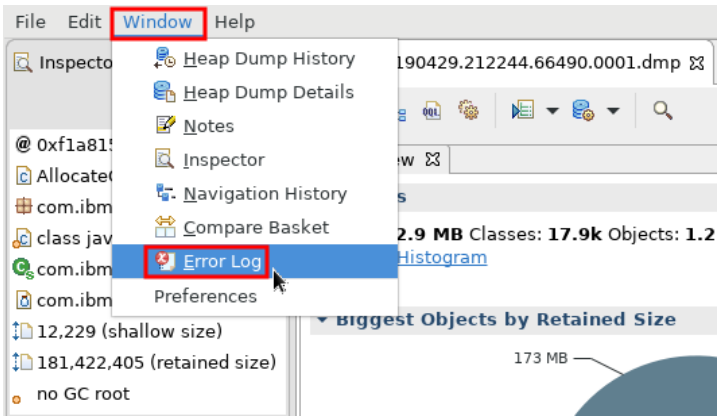
5. Now the “Progress” view is opened:



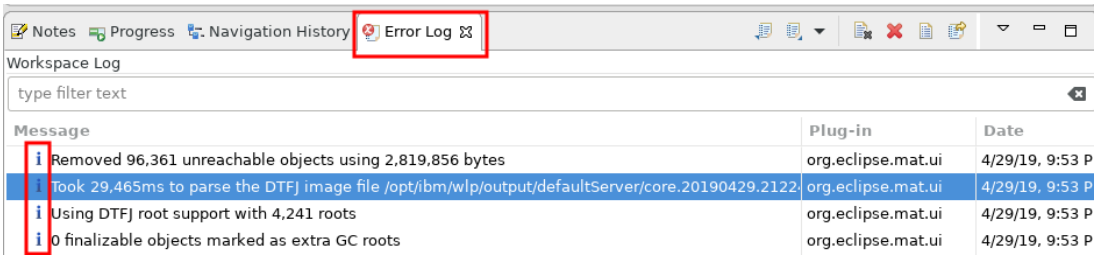
6. After the dump finishes loading, a pop-up will appear with suggested actions such as running the leak suspect report. For now, just click “Cancel”:



7. The first thing to check is to see whether there were any errors processing the dump. Click Window > Error Log:

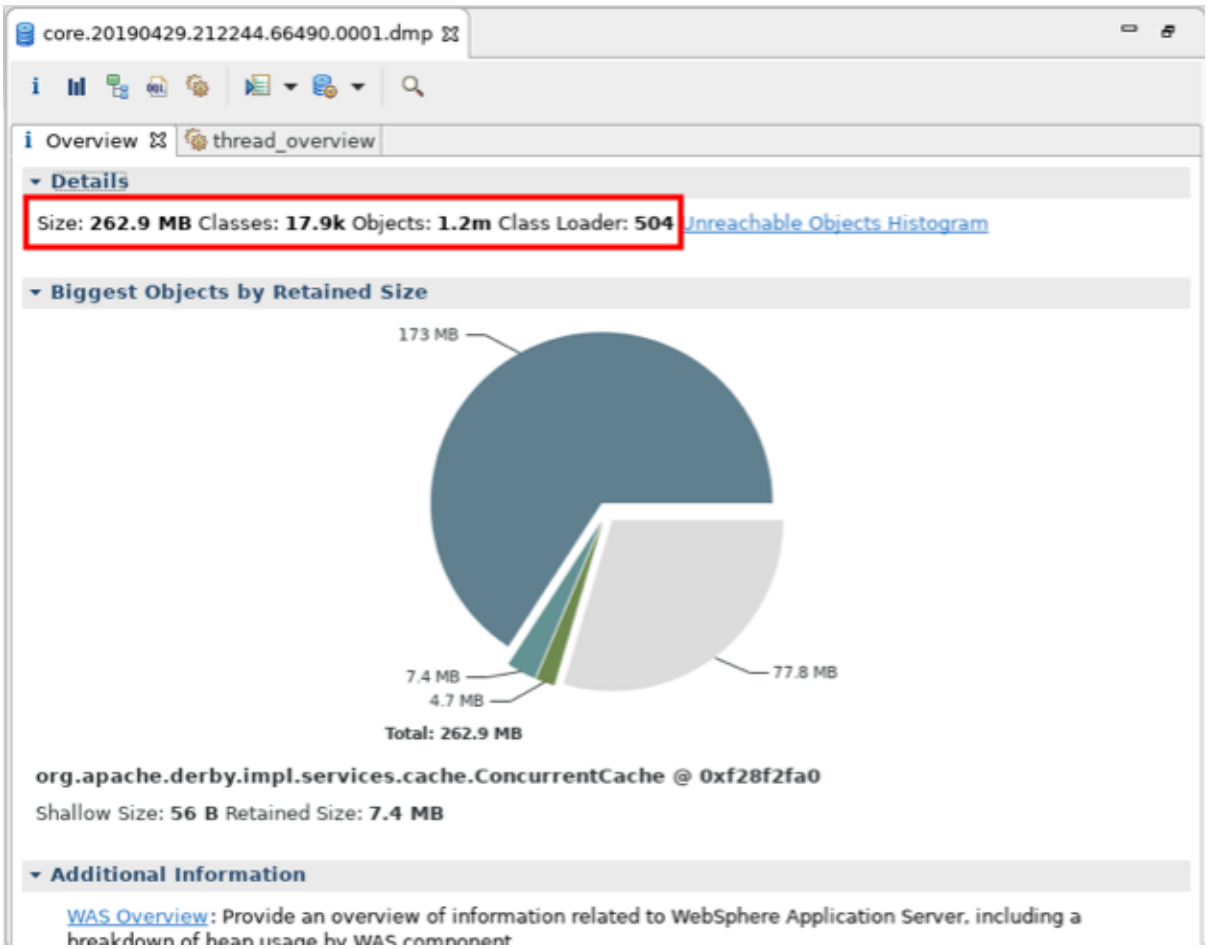


8. Review the list and check for any warnings or errors:



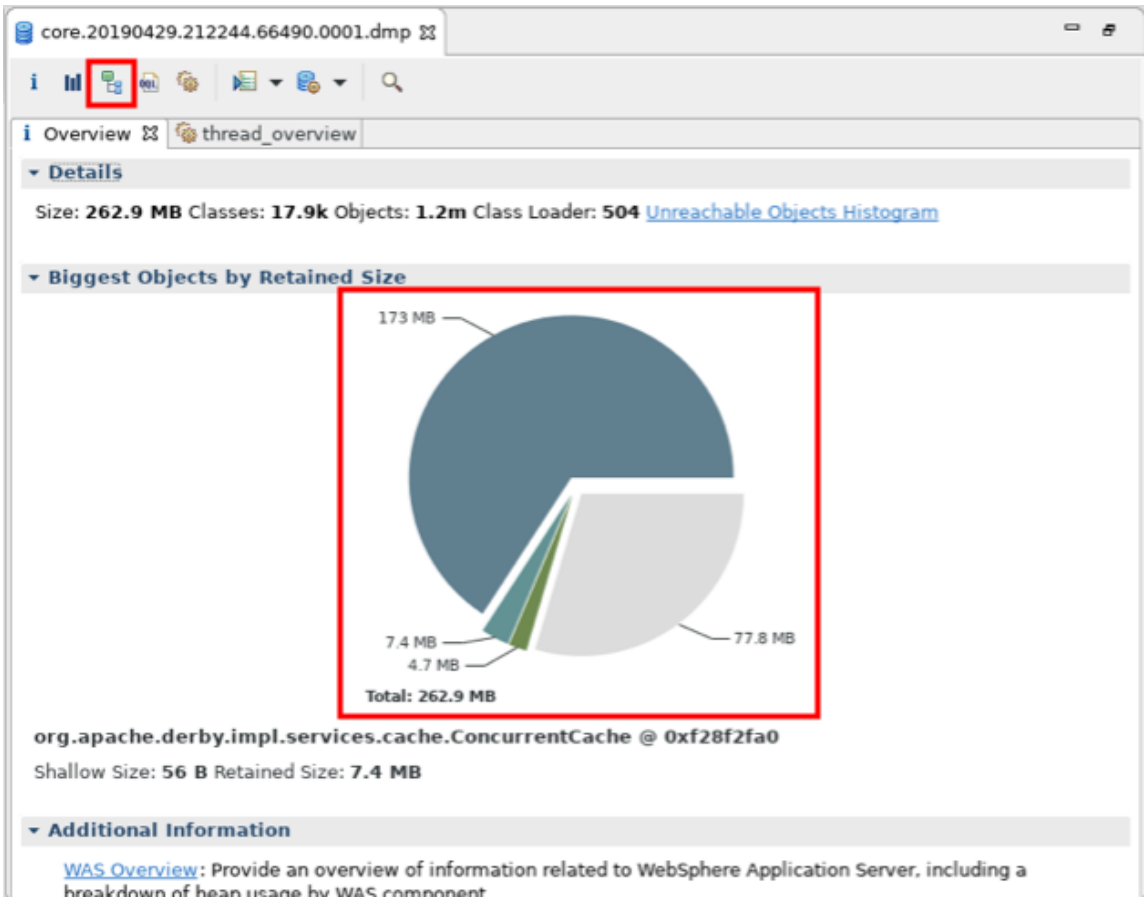
    1. It is possible to have a few warnings without too many problems. If you believe the warnings are limiting your analysis, consider opening an IBM Support case to investigate the issue with the IBM Java support team.

9. The overview tab shows the total live Java heap usage and the number of live classes, classloaders, and objects:

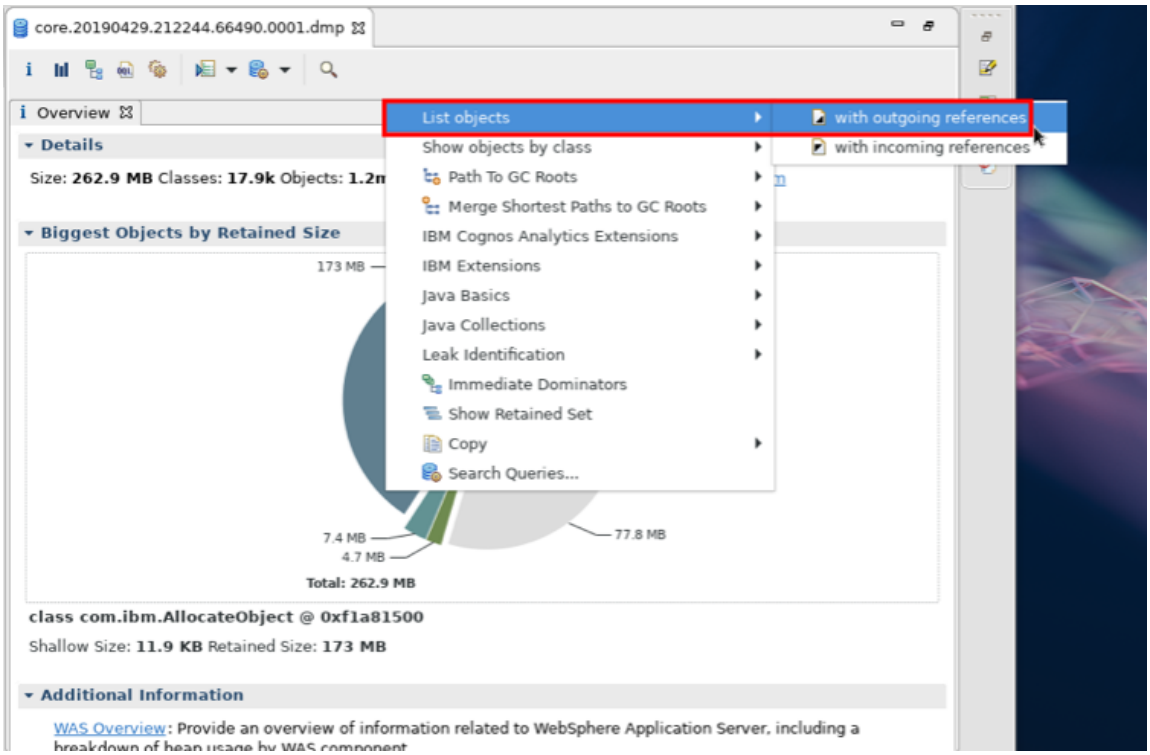


1. By default, MAT performs a full "garbage collection" when it loads the dump so everything you see is only pertaining to live Java objects. You can click on the "Unreachable Objects Histogram" link to see a histogram of any objects that are trash.

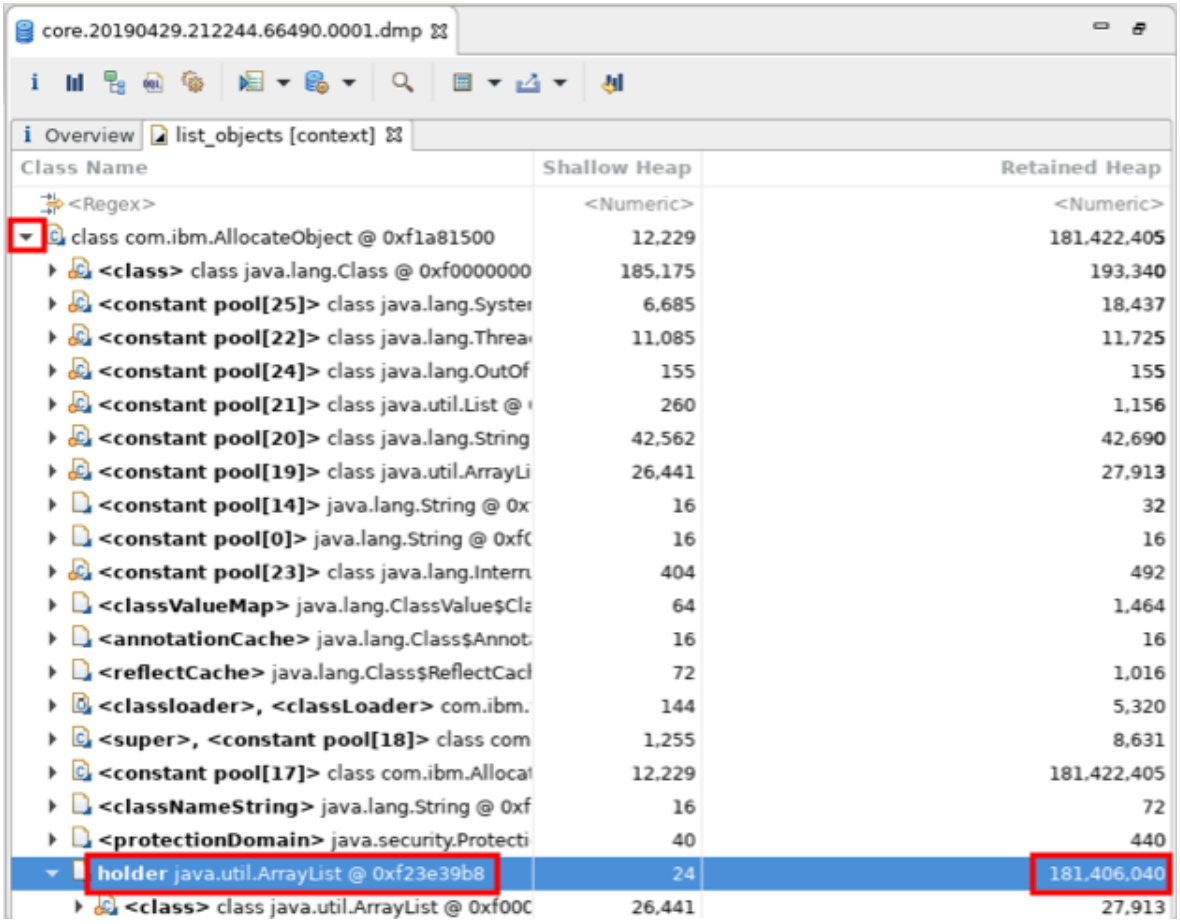
10. The pie chart on the Overview tab shows the largest dominator objects so it's a subset of the Dominator Tree button:



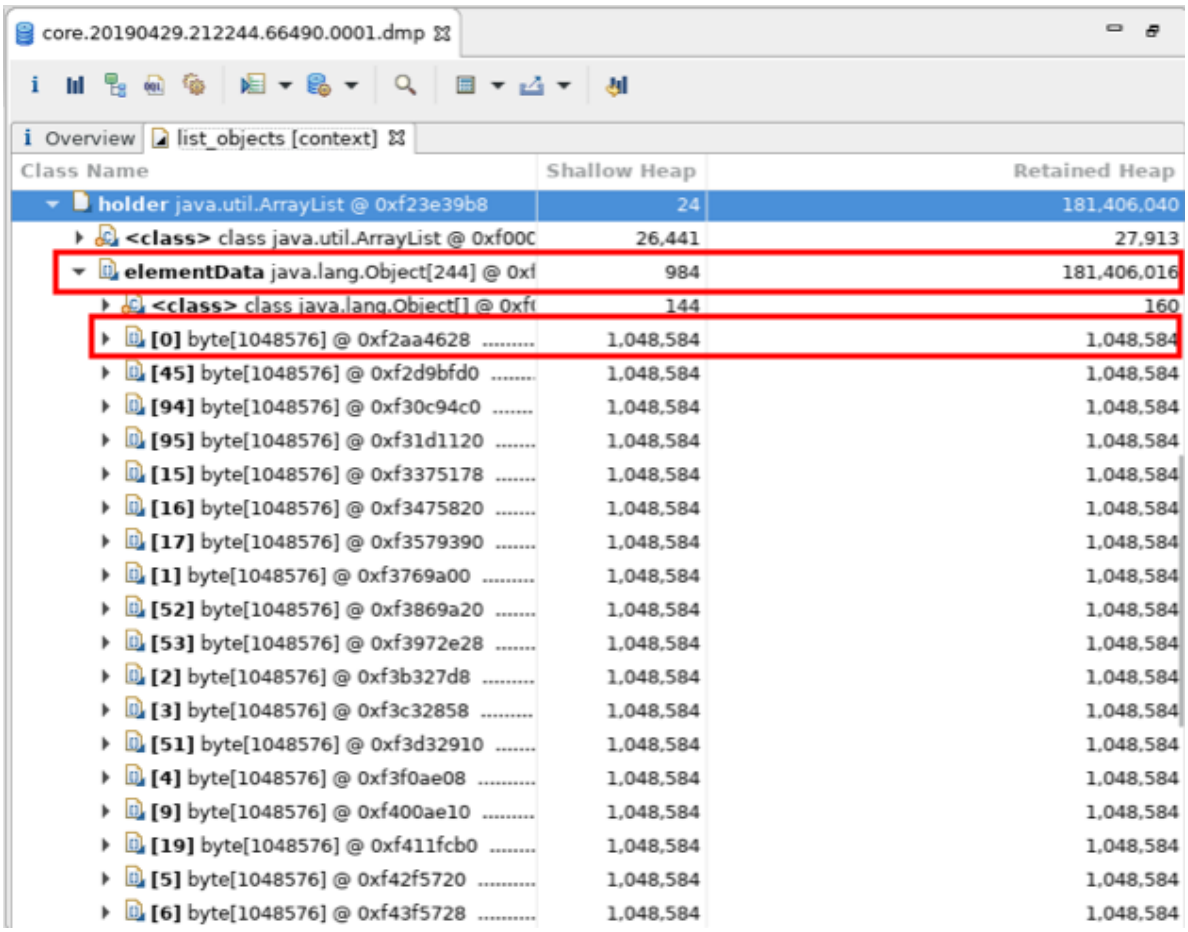
11. You may left click on a pie slice and select List objects > with outgoing references to review the object graph of the large dominator:



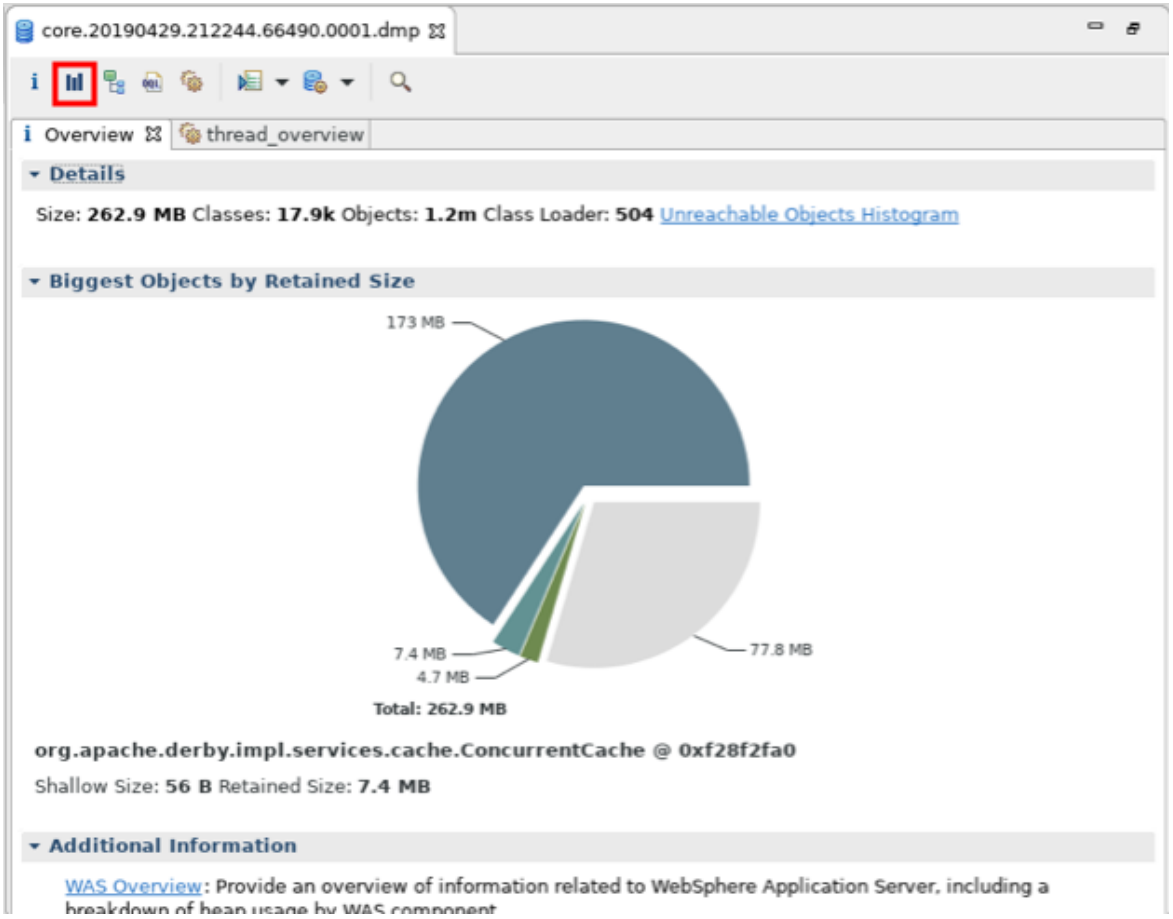
12. Expand the outgoing references tree and walk down the path with the largest “Retained Heap” values; in this example, there is an ArrayList called “holder”:



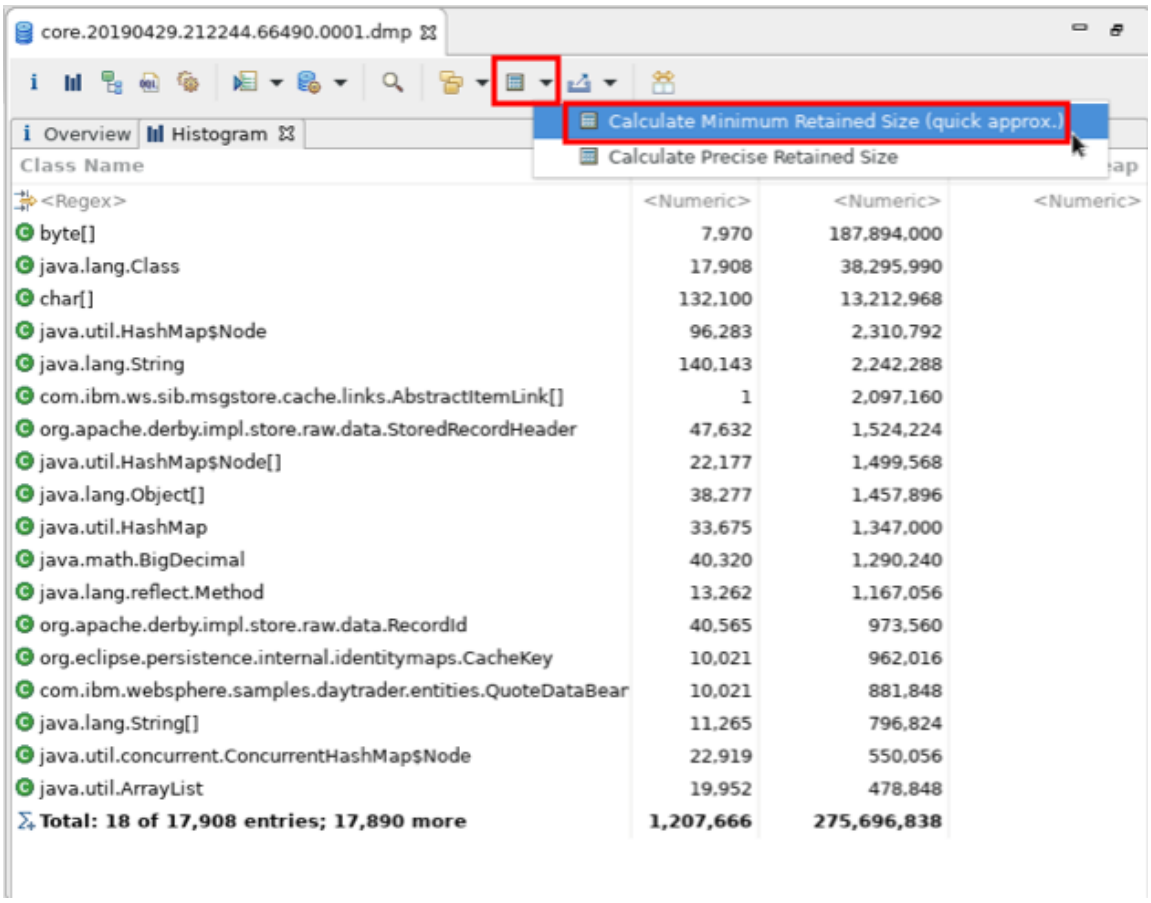
13. Continue walking down the tree and you will find an Object array with undreds of elements, each of about 1MB, which matches what we executed to create the OutOfMemoryError:



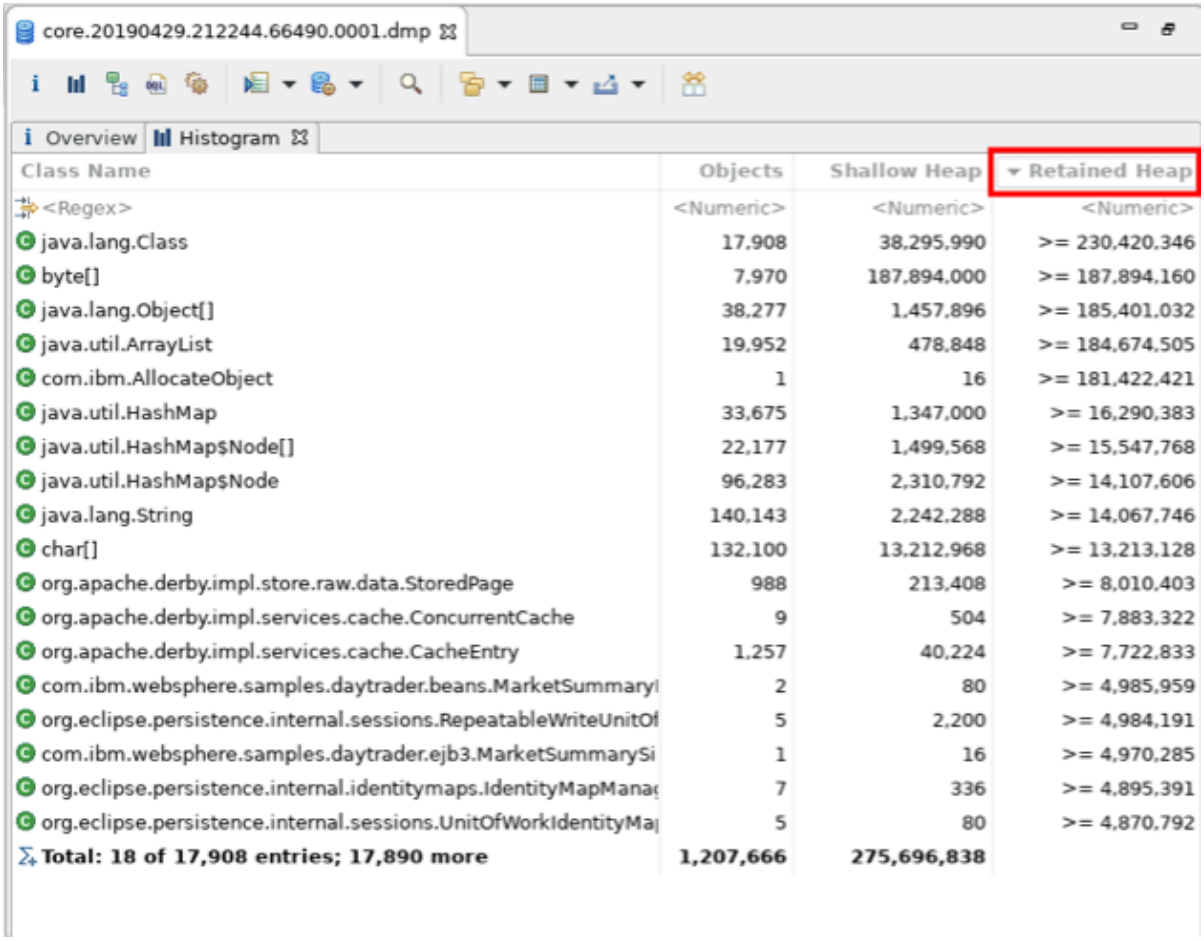
14. The next common view is to explore the Histogram:



15. Click on the calculator button and select “Calculate Minimum Retained Size (quick approx.)” to populate the Retained Heap column for each class:



16. This fills in the retained heap column which then you can click to sort descending:

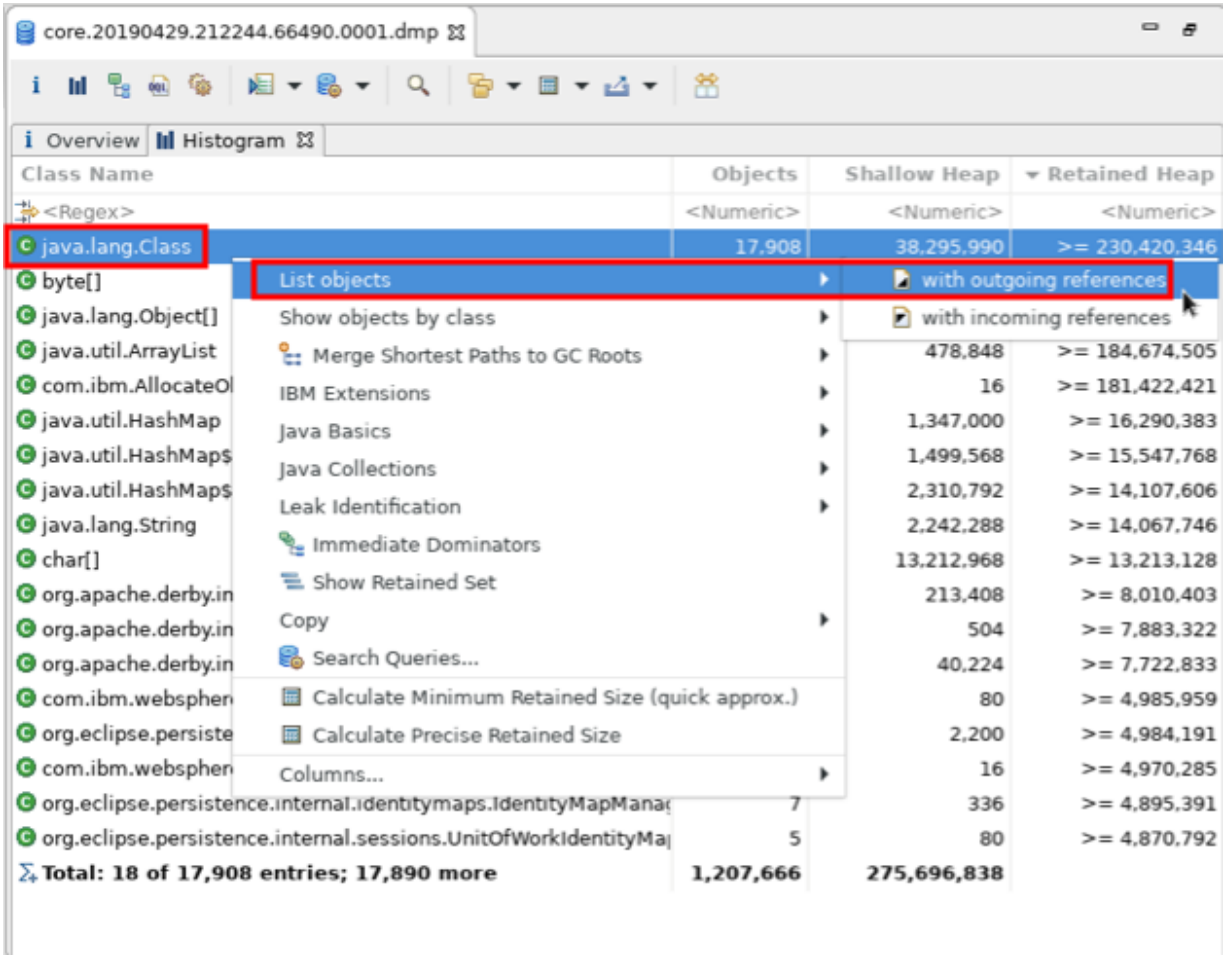


core.20190429.212244.66490.0001.dmp

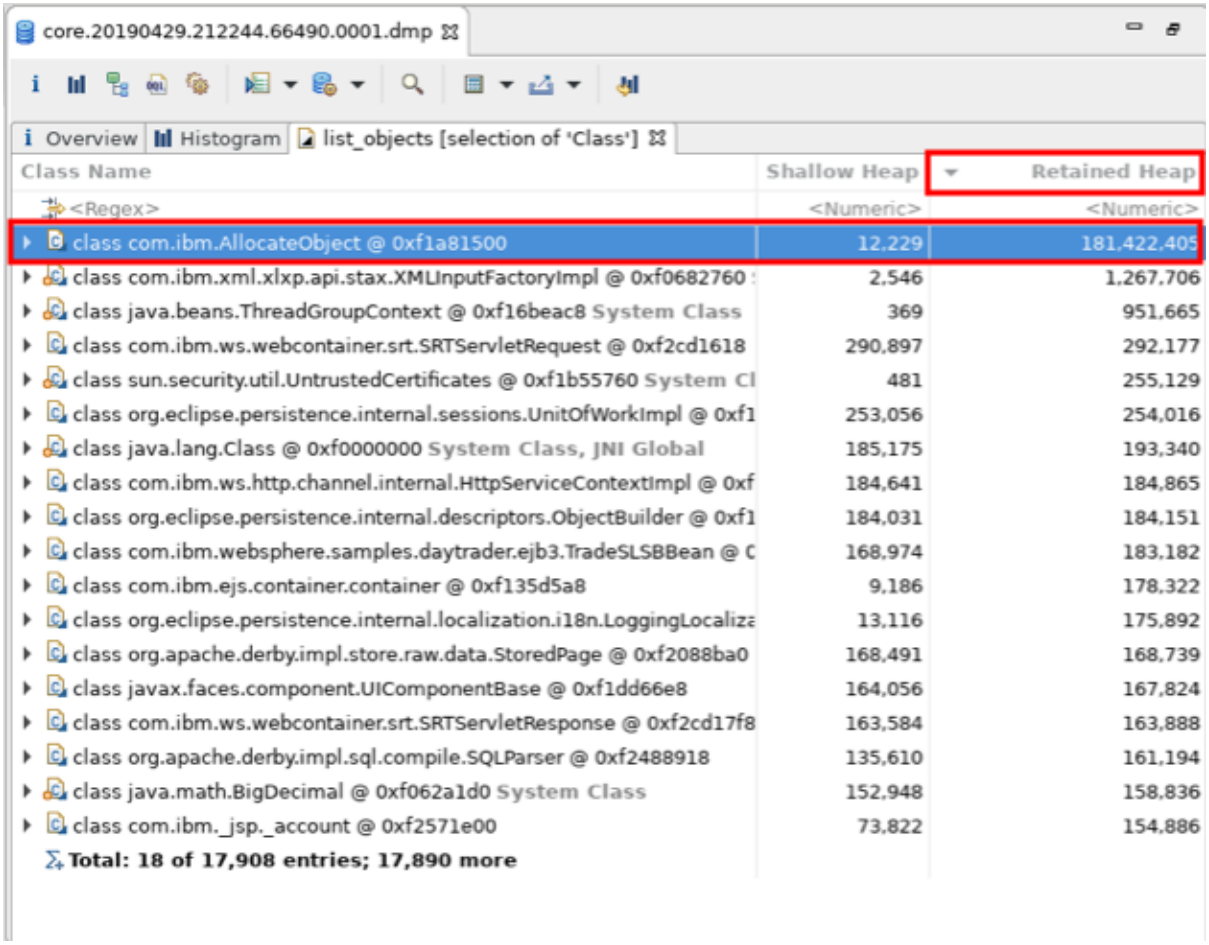
Overview | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.Class	17,908	38,295,990	>= 230,420,346
byte[]	7,970	187,894,000	>= 187,894,160
java.lang.Object[]	38,277	1,457,896	>= 185,401,032
java.util.ArrayList	19,952	478,848	>= 184,674,505
com.ibm.AllocateObject	1	16	>= 181,422,421
java.util.HashMap	33,675	1,347,000	>= 16,290,383
java.util.HashMap$Node[]	22,177	1,499,568	>= 15,547,768
java.util.HashMap$Node	96,283	2,310,792	>= 14,107,606
java.lang.String	140,143	2,242,288	>= 14,067,746
char[]	132,100	13,212,968	>= 13,213,128
org.apache.derby.impl.store.raw.data.StoredPage	988	213,408	>= 8,010,403
org.apache.derby.impl.services.cache.ConcurrentCache	9	504	>= 7,883,322
org.apache.derby.impl.services.cache.CacheEntry	1,257	40,224	>= 7,722,833
com.ibm.websphere.samples.daytrader.beans.MarketSummaryI	2	80	>= 4,985,959
org.eclipse.persistence.internal.sessions.RepeatableWriteUnitOf	5	2,200	>= 4,984,191
com.ibm.websphere.samples.daytrader.ejb3.MarketSummarySi	1	16	>= 4,970,285
org.eclipse.persistence.internal.identitymaps.IdentityMapManag	7	336	>= 4,895,391
org.eclipse.persistence.internal.sessions.UnitOfWorkIdentityMa	5	80	>= 4,870,792
Total: 18 of 17,908 entries; 17,890 more	**1,207,666**	**275,696,838**	

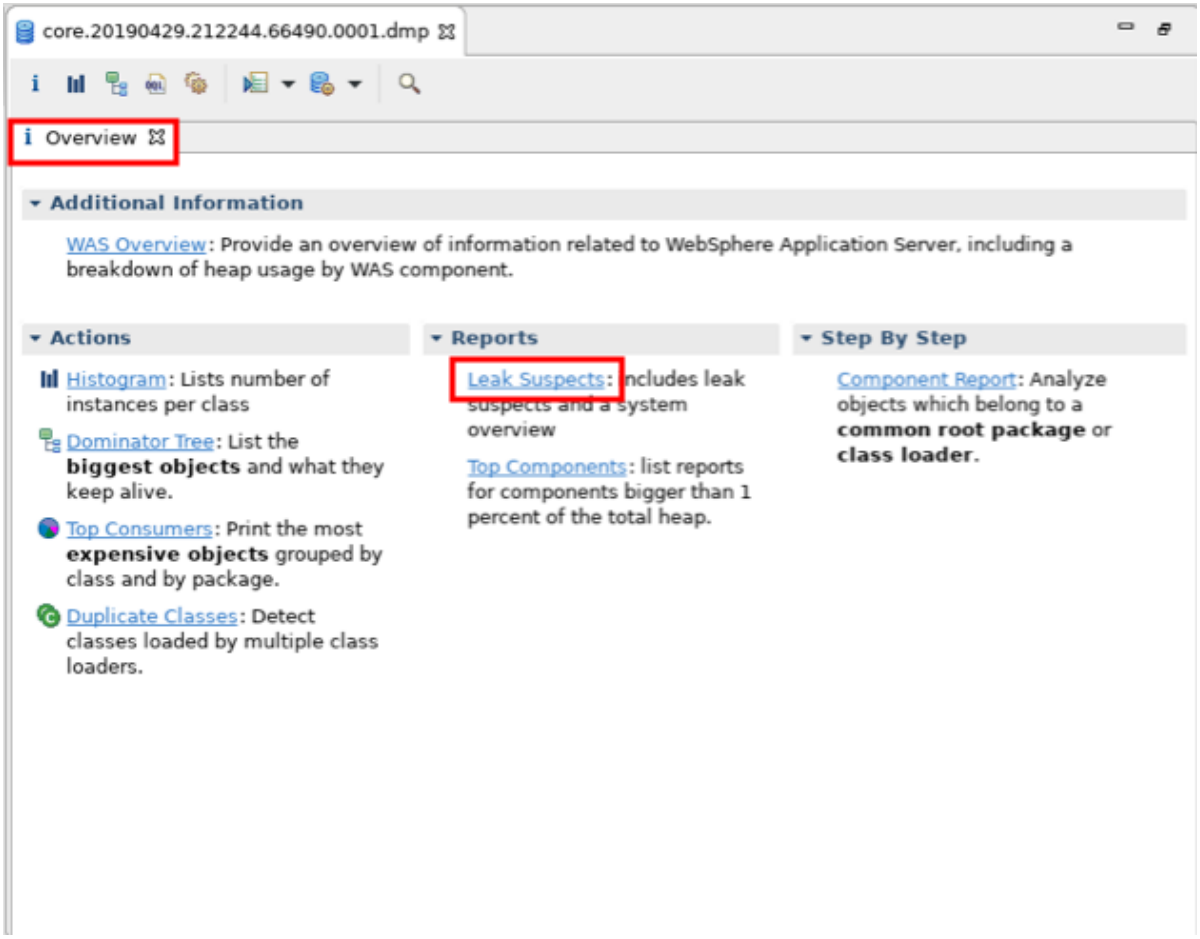
17. You may click on a row with a large retained heap size, right click and select outgoing references:



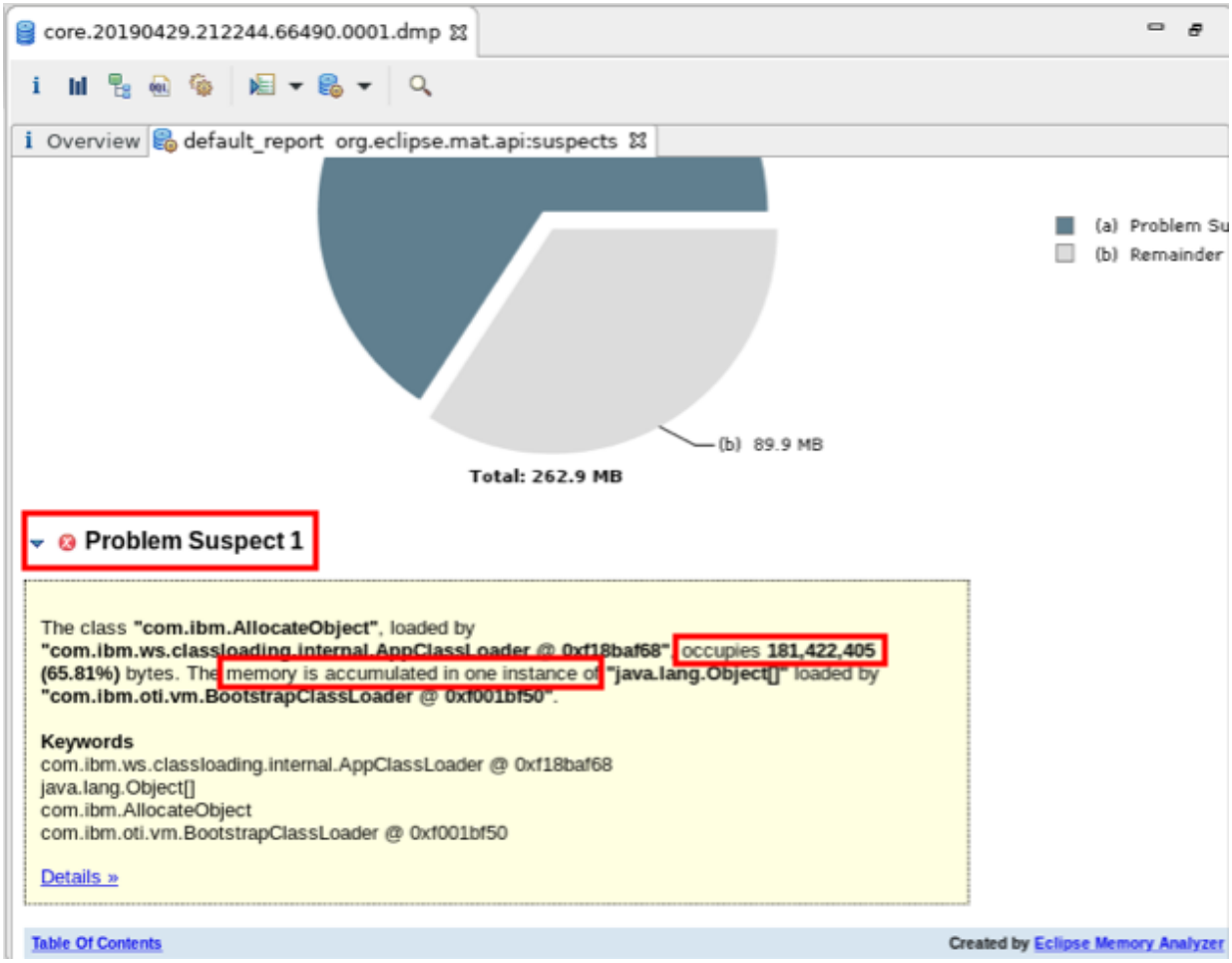
18. Then sort by Retained Heap and again you will find the large class:



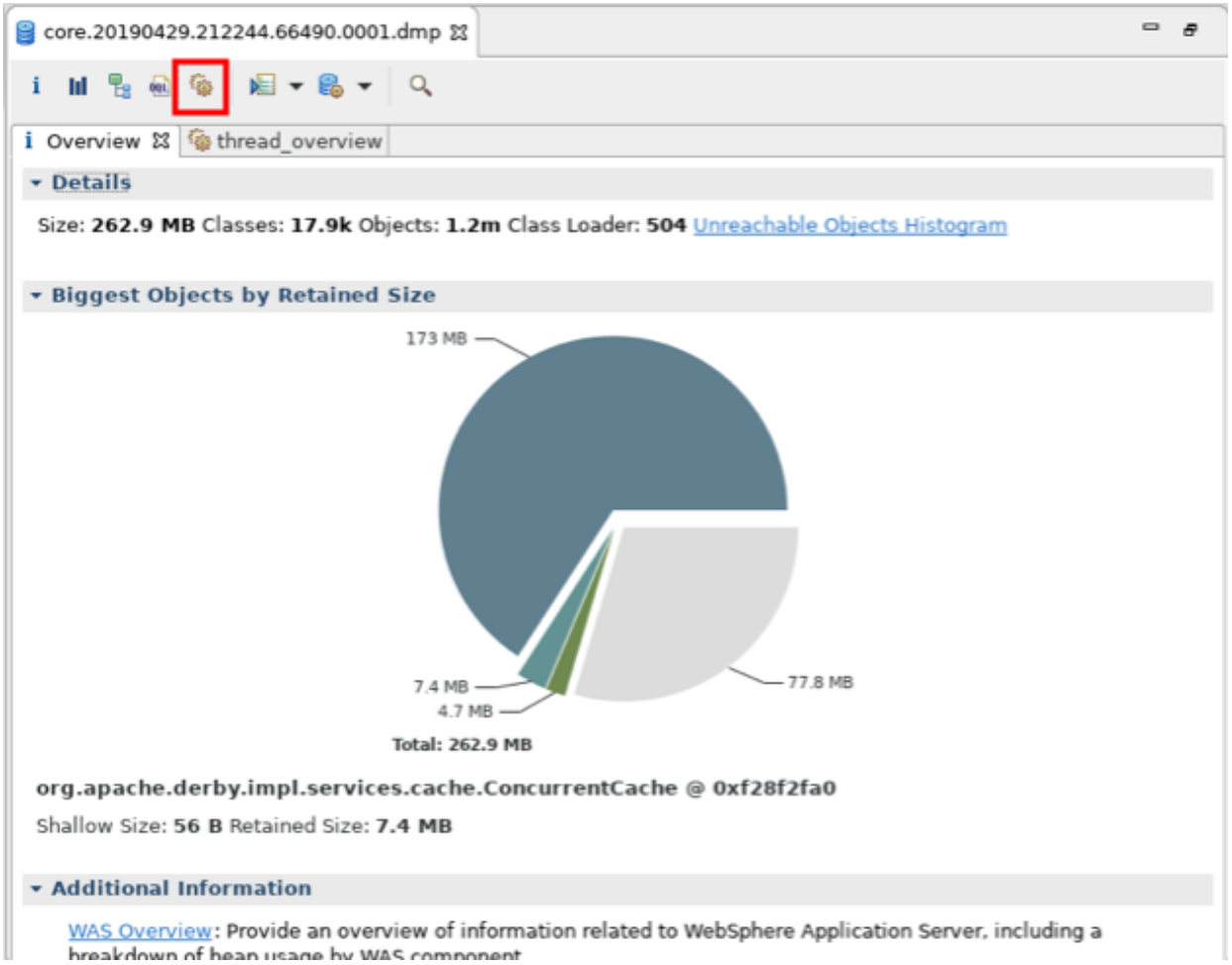
19. The next common view to explore is the Leak Suspects view. On the Overview tab, scroll down and click on Leak Suspects:



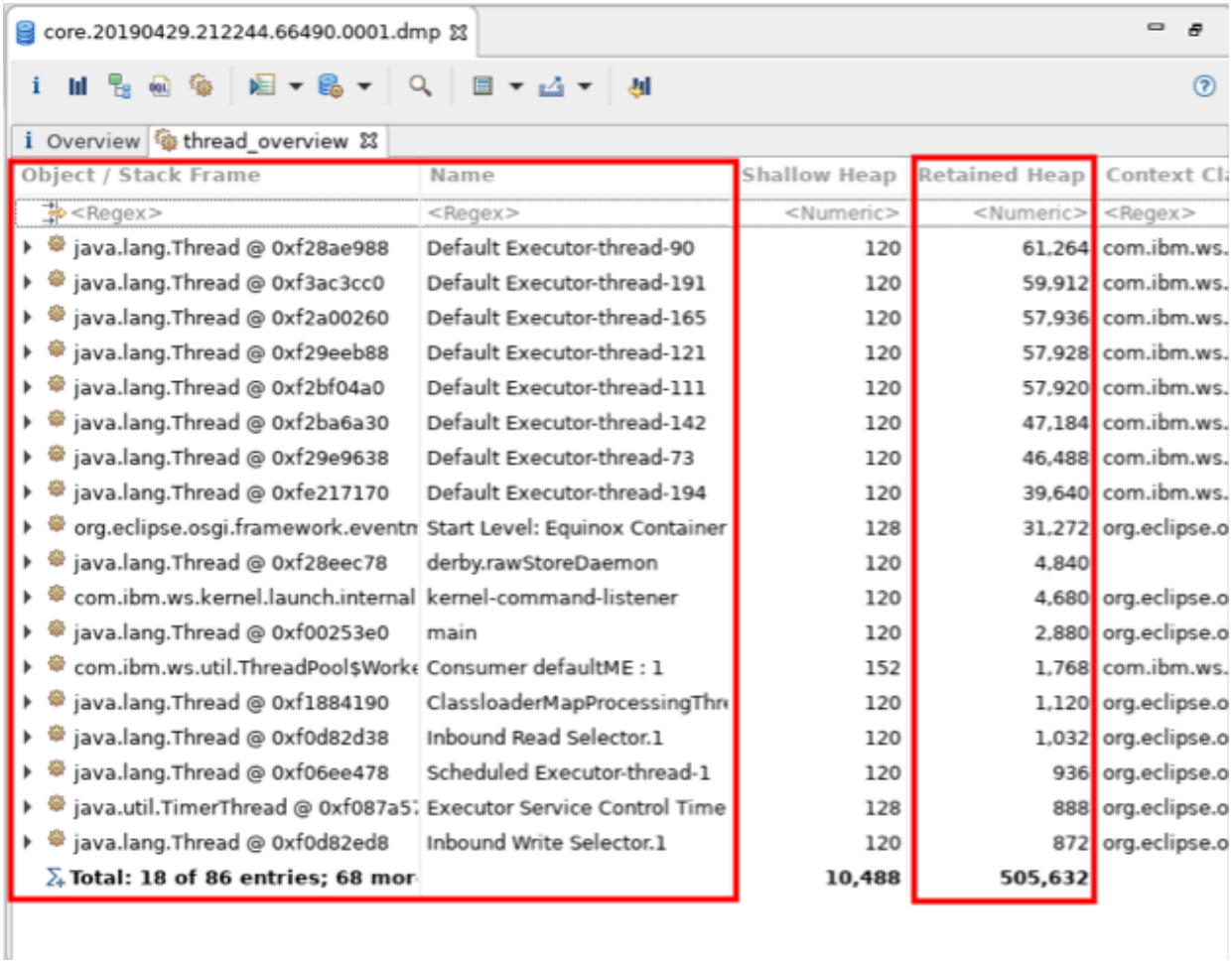
20. The report will list leak suspects in the order of their size. The following example shows the same leaking class Object array:



21. The next common view is the Threads view:

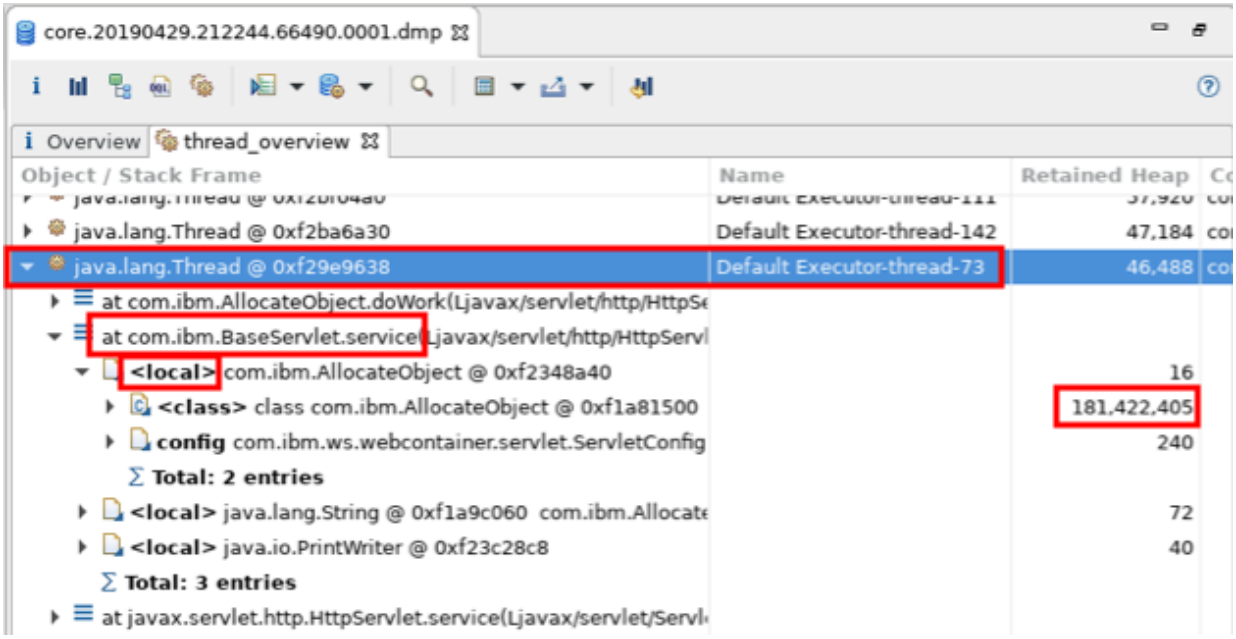


22. This will list every thread, the thread name, the retained heap of the thread, other thread details, and the stack frame along with stack frame locals:

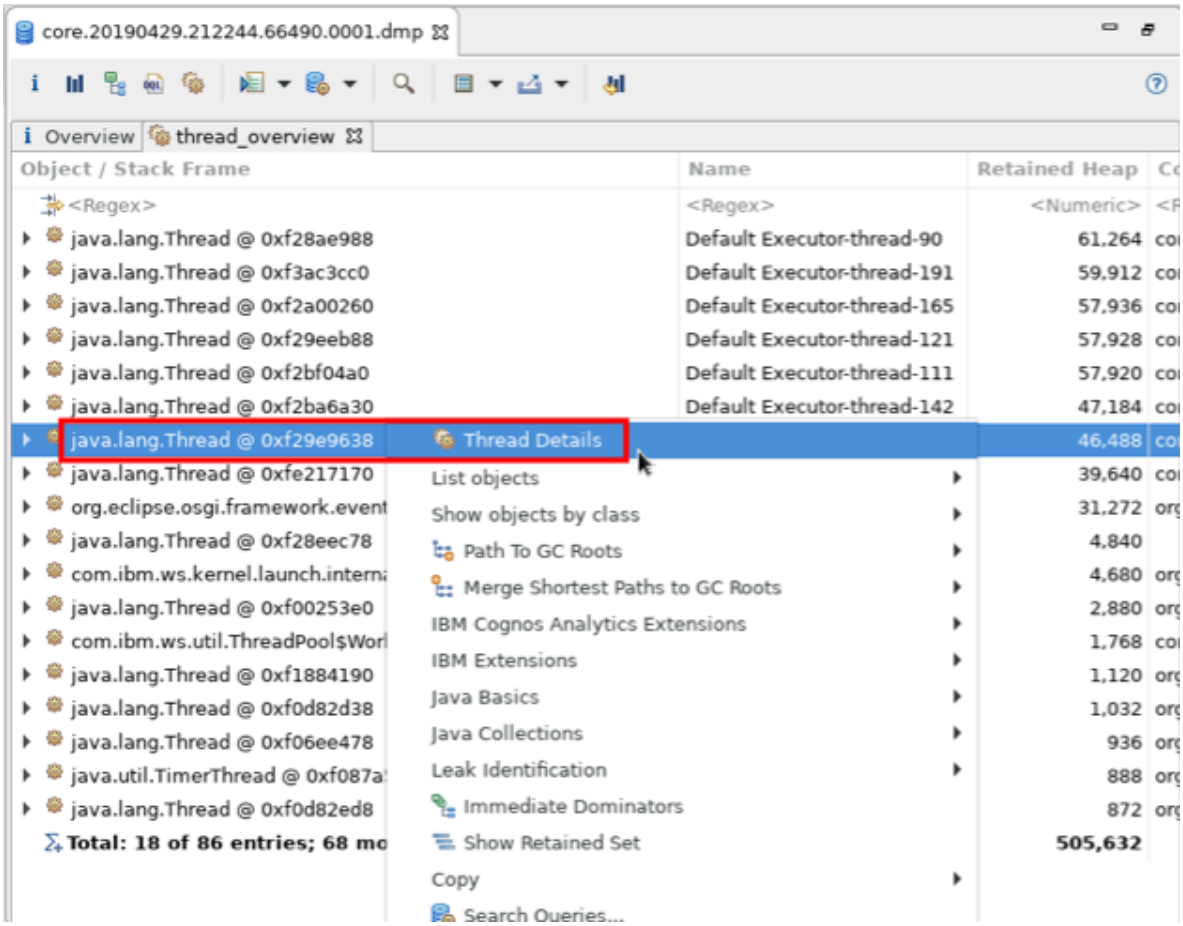


Object / Stack Frame	Name	Shallow Heap	Retained Heap	Context Cl
<Regex>	<Regex>	<Numeric>	<Numeric>	<Regex>
java.lang.Thread @ 0xf28ae988	Default Executor-thread-90	120	61,264	com.ibm.ws.
java.lang.Thread @ 0xf3ac3cc0	Default Executor-thread-191	120	59,912	com.ibm.ws.
java.lang.Thread @ 0xf2a00260	Default Executor-thread-165	120	57,936	com.ibm.ws.
java.lang.Thread @ 0xf29eeb88	Default Executor-thread-121	120	57,928	com.ibm.ws.
java.lang.Thread @ 0xf2bf04a0	Default Executor-thread-111	120	57,920	com.ibm.ws.
java.lang.Thread @ 0xf2ba6a30	Default Executor-thread-142	120	47,184	com.ibm.ws.
java.lang.Thread @ 0xf29e9638	Default Executor-thread-73	120	46,488	com.ibm.ws.
java.lang.Thread @ 0xfe217170	Default Executor-thread-194	120	39,640	com.ibm.ws.
org.eclipse.osgi.framework.eventn	Start Level: Equinox Container	128	31,272	org.eclipse.o
java.lang.Thread @ 0xf28eec78	derby.rawStoreDaemon	120	4,840	
com.ibm.ws.kernel.launch.internal	kernel-command-listener	120	4,680	org.eclipse.o
java.lang.Thread @ 0xf00253e0	main	120	2,880	org.eclipse.o
com.ibm.ws.util.ThreadPool$Worke	Consumer defaultME : 1	152	1,768	com.ibm.ws.
java.lang.Thread @ 0xf1884190	ClassloaderMapProcessingThr	120	1,120	org.eclipse.o
java.lang.Thread @ 0xf0d82d38	Inbound Read Selector.1	120	1,032	org.eclipse.o
java.lang.Thread @ 0xf06ee478	Scheduled Executor-thread-1	120	936	org.eclipse.o
java.util.TimerThread @ 0xf087a5	Executor Service Control Time	128	888	org.eclipse.o
java.lang.Thread @ 0xf0d82ed8	Inbound Write Selector.1	120	872	org.eclipse.o
Total: 18 of 86 entries; 68 mor		**10,488**	**505,632**	

23. You may expand the thread stacks until you find the servlet that caused the leak:

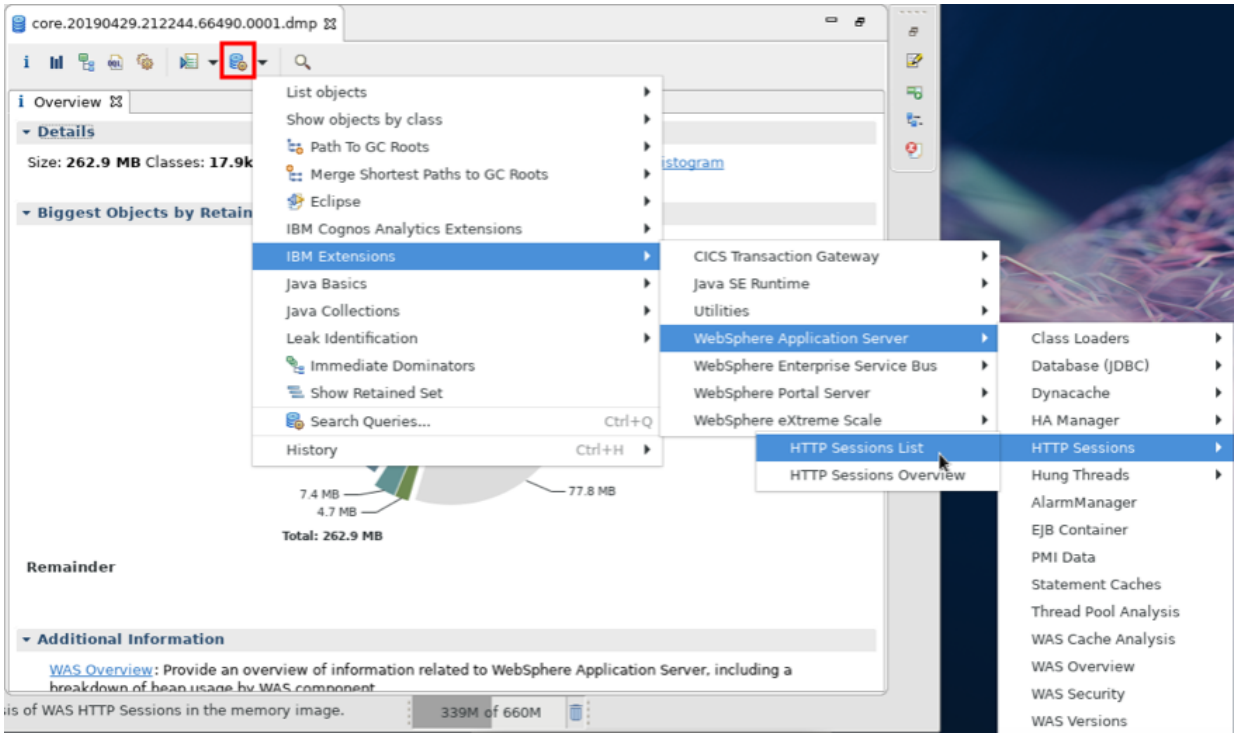


1. Note that you can see the actual objects on each stack frame. In this case, we can clearly see the servlet has reference to the AllocateObject class which is retaining most of the heap. This stack usually makes it much easier for the application developer to understand what happened. Right click on the thread and select Thread Details to get a full thread stack that may be copy-and-pasted:


core.20190429.212244.66490.0001.dmp
Overview
thread_overview
Object / Stack Frame
Name
Retained Heap
<Regex>
<Regex>
<Numeric>
java.lang.Thread @ 0xf28ae988
Default Executor-thread-90
61,264
java.lang.Thread @ 0xf3ac3cc0
Default Executor-thread-191
59,912
java.lang.Thread @ 0xf2a00260
Default Executor-thread-165
57,936
java.lang.Thread @ 0xf29eeb88
Default Executor-thread-121
57,928
java.lang.Thread @ 0xf2bf04a0
Default Executor-thread-111
57,920
java.lang.Thread @ 0xf2ba6a30
Default Executor-thread-142
47,184
java.lang.Thread @ 0xf29e9638
46,488
java.lang.Thread @ 0xfe217170
39,640
org.eclipse.osgi.framework.event
31,272
java.lang.Thread @ 0xf28eec78
4,840
com.ibm.ws.kernel.launch.interna
4,680
java.lang.Thread @ 0xf00253e0
2,880
com.ibm.ws.util.ThreadPool$Worl
1,768
java.lang.Thread @ 0xf1884190
1,120
java.lang.Thread @ 0xf0d82d38
1,032
java.lang.Thread @ 0xf06ee478
936
java.util.TimerThread @ 0xf087a
888
java.lang.Thread @ 0xf0d82ed8
872
Total: 18 of 86 entries; 68 mo
505,632
Thread Details
List objects
Show objects by class
Path To GC Roots
Merge Shortest Paths to GC Roots
IBM Cognos Analytics Extensions
IBM Extensions
Java Basics
Java Collections
Leak Identification
Immediate Dominators
Show Retained Set
Copy
Search Queries...

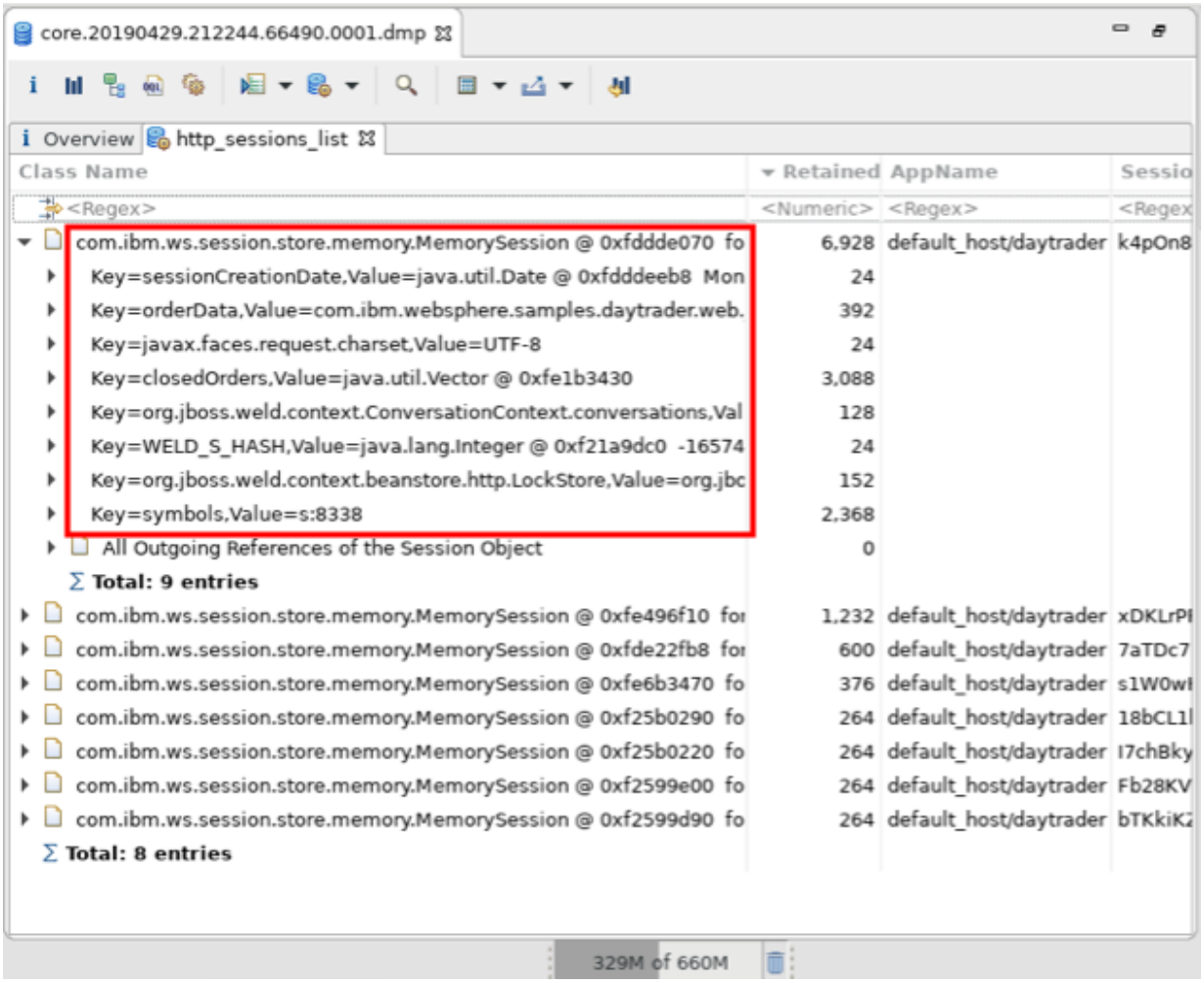
The IBM Extensions for Memory Analyzer (IEMA)[27] provide additional extensions on top of MAT with WAS, Java, and other related queries.

1. As one example, you can see a list of all HTTP sessions and their attributes with: Open Query Browser > IBM Extensions > WebSphere Application Server > HTTP Sessions > HTTP Sessions List:



[27] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Memory_Analyzer_Tool.html#Major_Tools-IBM_Memory_Analyzer_Tool_MAT-Installation

2. Each HTTP session is listed, as well as how much Java heap it retains, which application it's associated with, and other details, including all of the attribute names and values:



10.3 Health Center

IBM Monitoring and Diagnostics for Java - Health Center[28] is free and shipped with IBM Java. Among other things, Health Center includes a statistical CPU profiler that samples Java stacks that are using CPU at a very high rate to determine what Java methods are using CPU. Health Center generally has an overhead of less than 1% and is suitable for production use. In recent versions, it may also be enabled dynamically without restarting the JVM.

This lab will demonstrate how to enable Java Health Center, exercise the sample DayTrader application using Apache JMeter, and review the health center file in the IBM Java Health Center Client Tool.

The Health Center agent gathers sampled CPU profiling data, along with other information:

- Classes: Information about classes being loaded
- Environment: Details of the configuration and system of the monitored application

[28] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Java_Health_Center.html

- Garbage collection: Information about the Java heap and pause times
- I/O: Information about I/O activities that take place.
- Locking: Information about contention on inflated locks
- Memory: Information about the native memory usage
- Profiling: Provides a sampling profile of Java methods including call paths

The Health Center agent can be enabled in two ways:

1. At startup by adding -Xhealthcenter:level=headless to the JVM arguments
2. At runtime, by running ${IBM_JAVA}/bin/java -jar ${IBM_JAVA}/jre/lib/ext/healthcenter.jar ID=${PID} level=headless

Note: For both items, you may add the following arguments to limit and roll the total file usage of Health Center data: -Dcom.ibm.java.diagnostics.healthcenter.headless.files.max.size=BYTES -Dcom.ibm.java.diagnostics.healthcenter.headless.files.to.keep=N (N=0 for unlimited)

The key step is that to produce the final Health Center HCD file, the JVM should be gracefully stopped (there are alternatives to this by packaging the temporary files but this isn't generally recommended).

Consider always enabling HealthCenter in headless mode[29] for post-mortem debugging of issues.

Exercise:

1. Stop the Apache JMeter test.
2. Stop the Liberty server.

```
/opt/ibm/wlp/bin/server stop defaultServer
```

3. Add the following line to /config/jvm.options:

```
-Xhealthcenter:level=headless
```

4. Start the Liberty server

```
/opt/ibm/wlp/bin/server start defaultServer
```

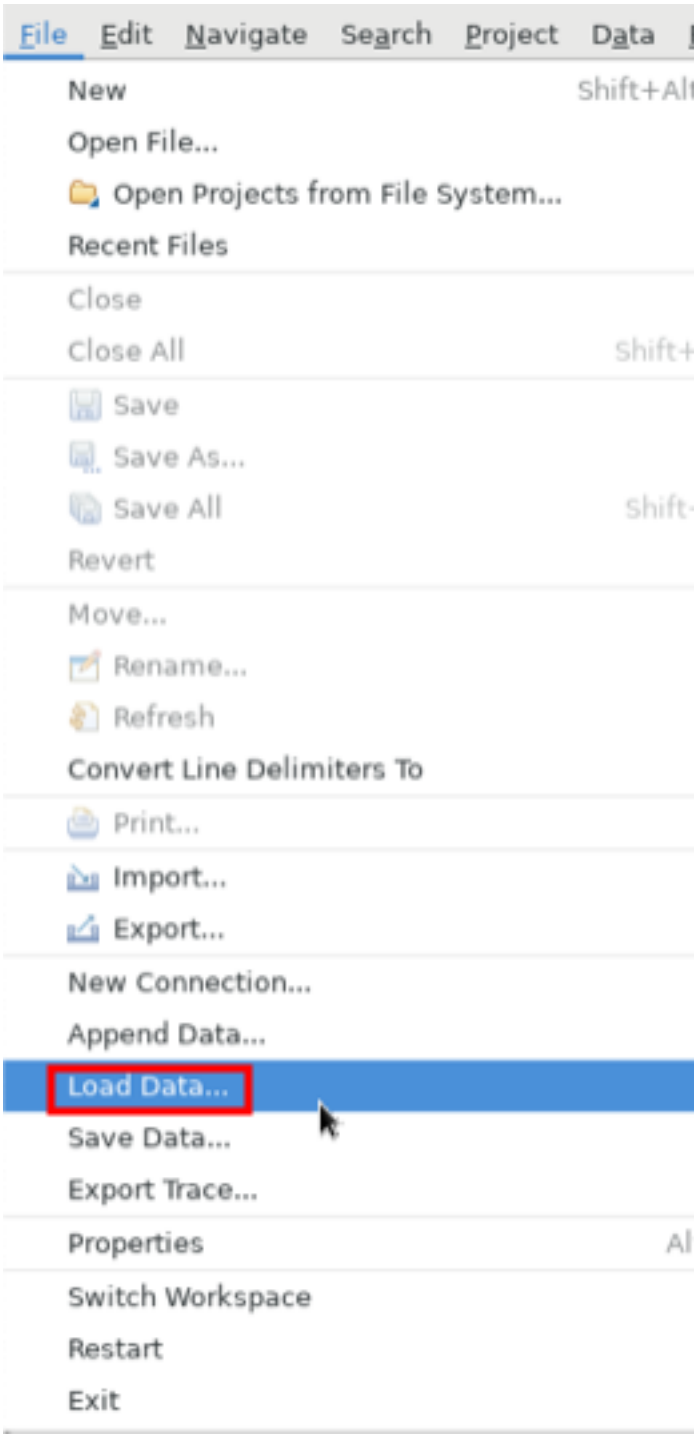
5. Start the Apache JMeter test and run it for some time.
6. Stop the Liberty server

```
/opt/ibm/wlp/bin/server stop defaultServer
```

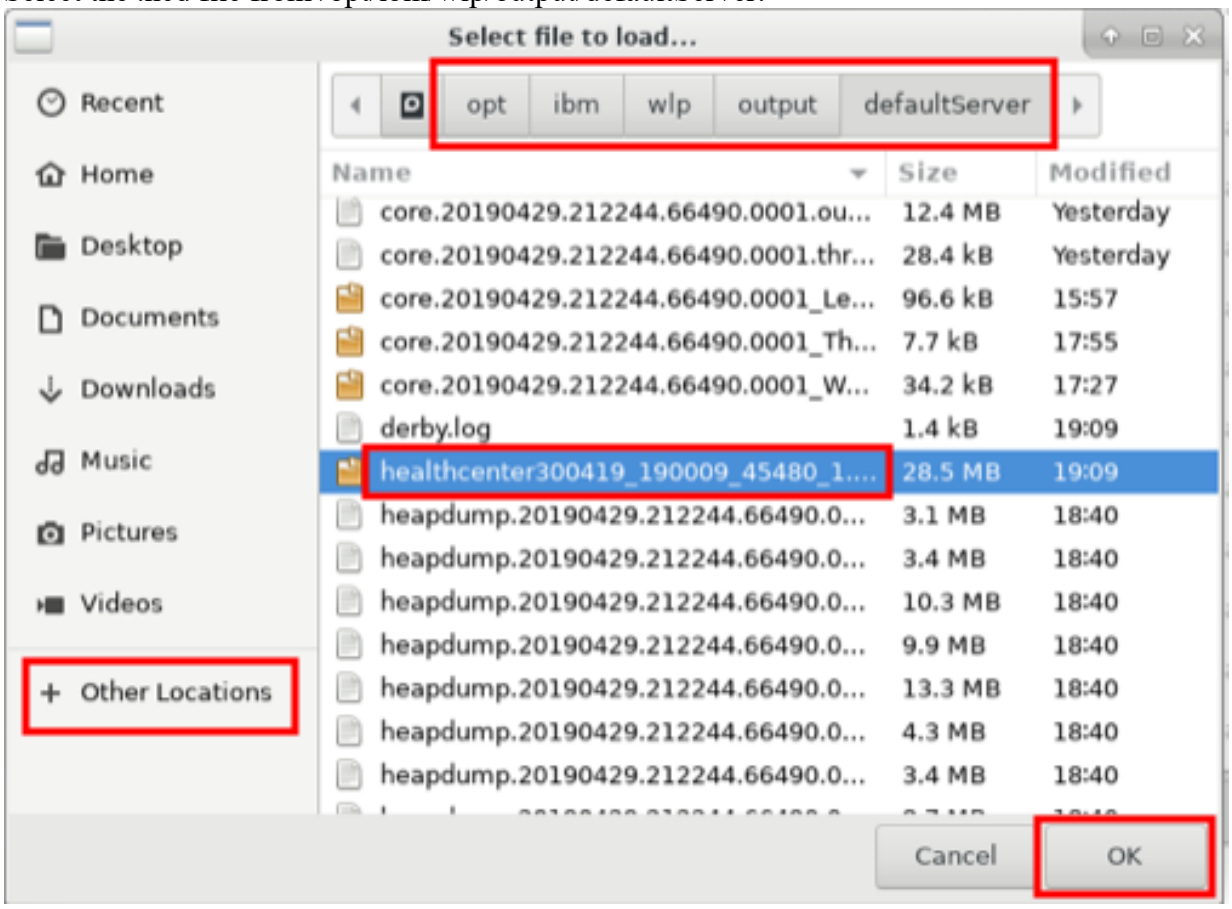
7. Open /opt/programs/ in the file browser and double click on **Health Center**.

[29] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Java_Health_Center.html#Major_Tools-IBM_Java_Health_Center-Gathering_Data

8. Click File > Load Data... (note that it's towards the bottom of the File menu; Open File does not work)

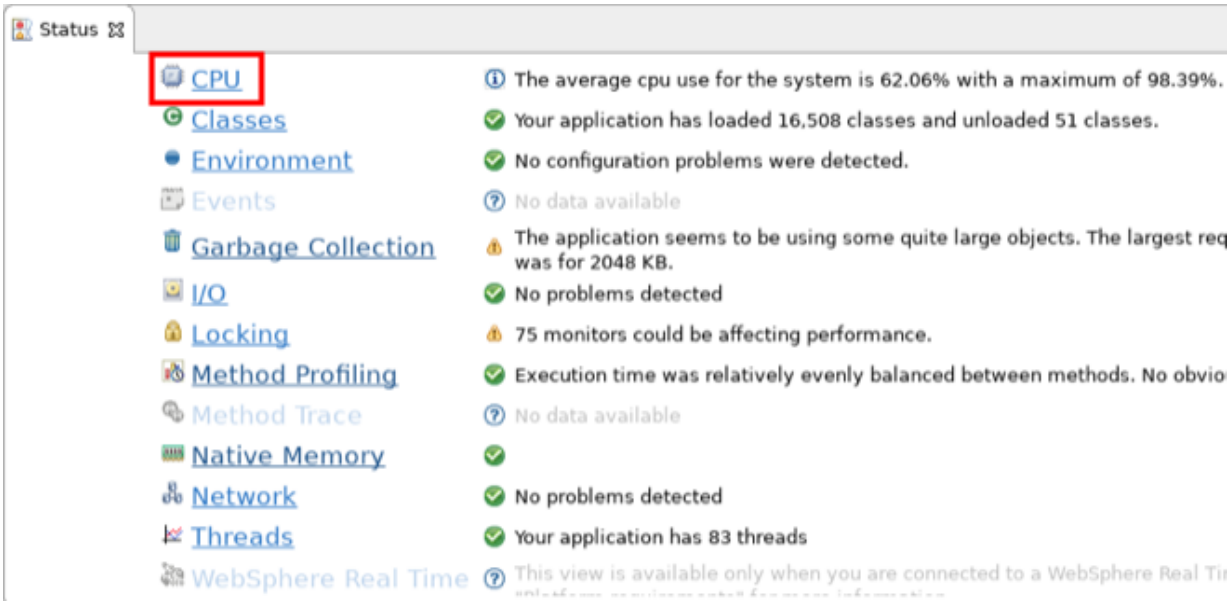


9. Select the .hcd file from /opt/ibm/wlp/output/defaultServer:

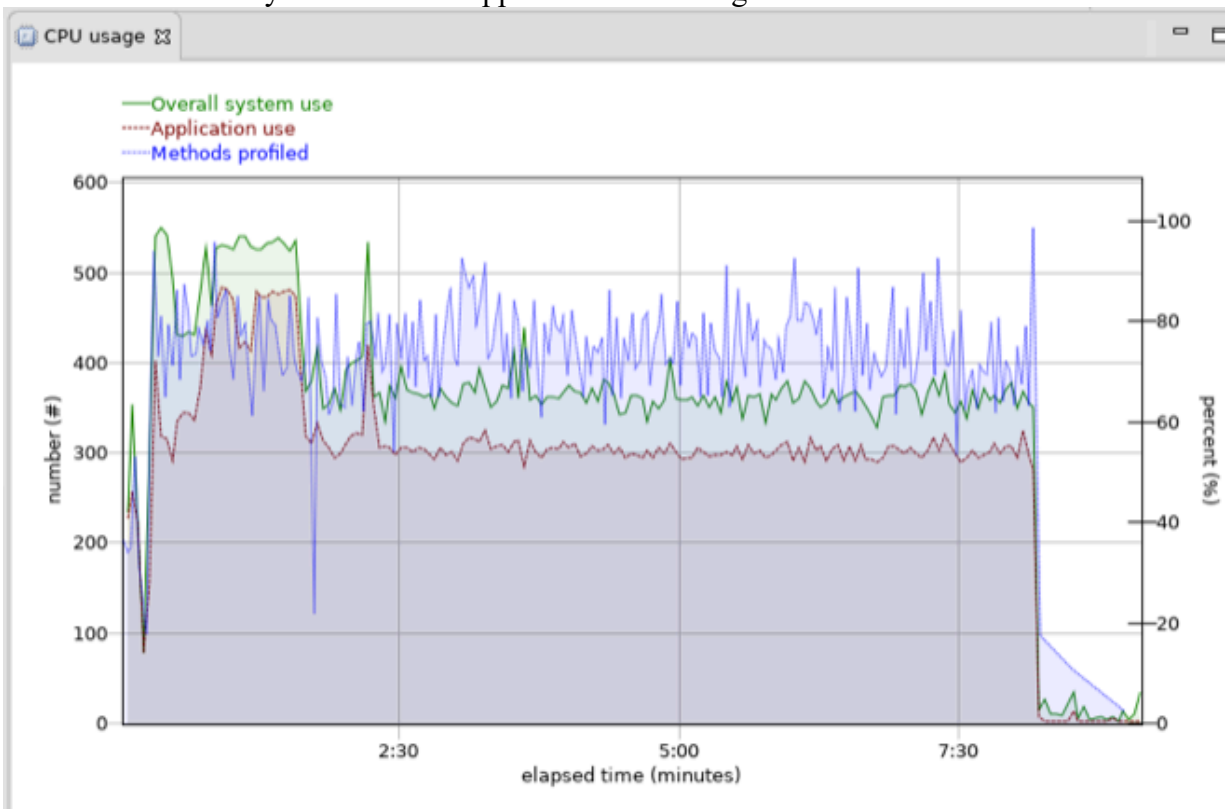


10. Wait for the data to complete loading:

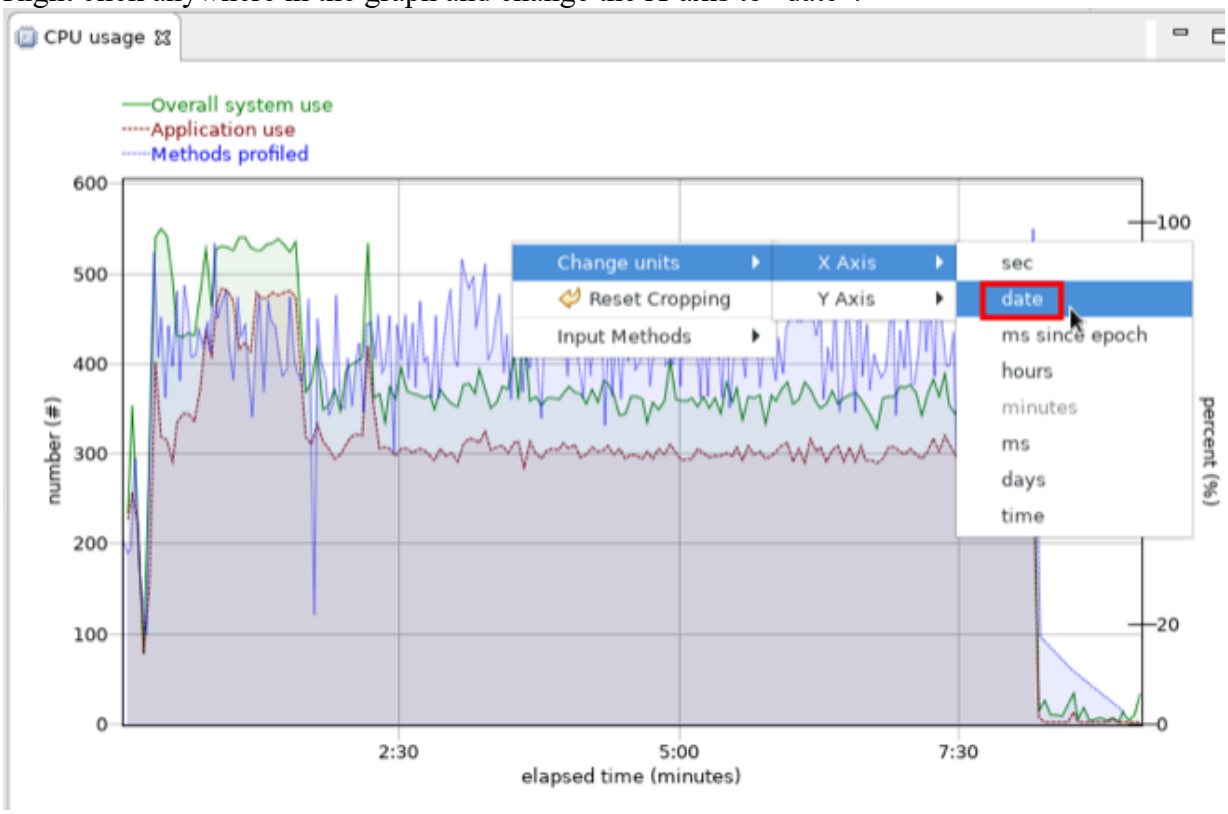
11. Click on CPU:



12. Review the overall system and Java application CPU usage:

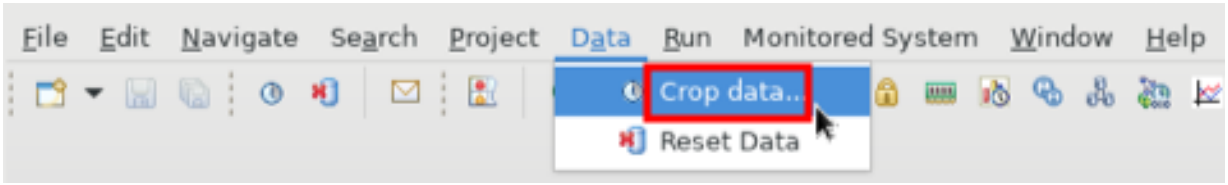


13. Right click anywhere in the graph and change the X-axis to “date”:

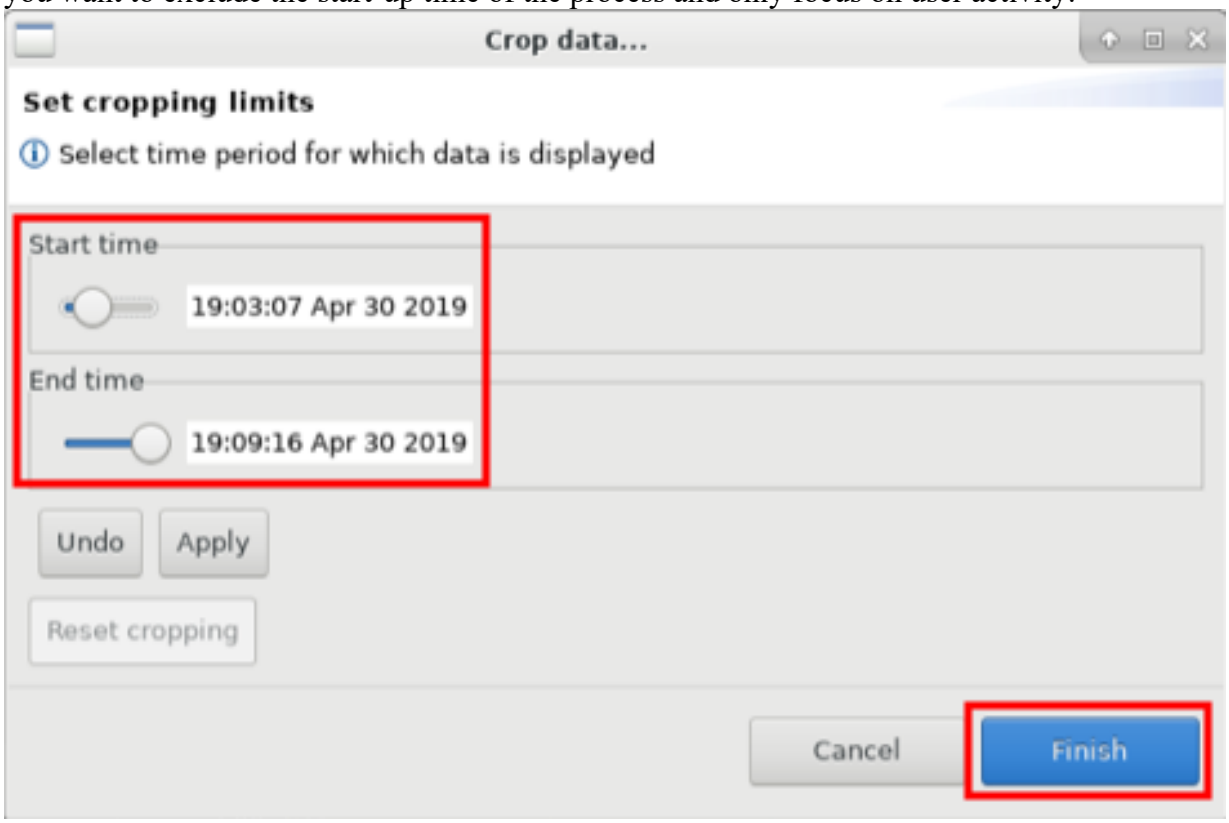


1. This will change all others views to date as well.

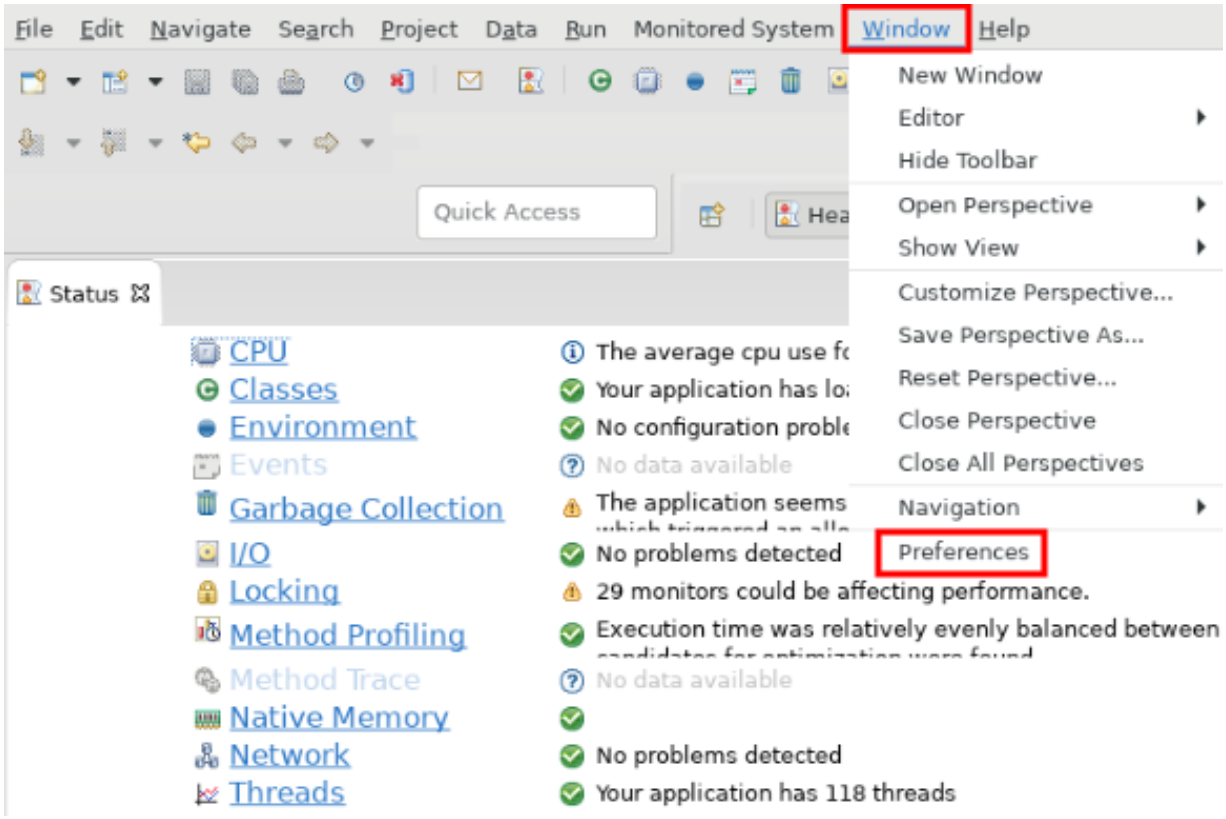
14. Click Data > Crop Data



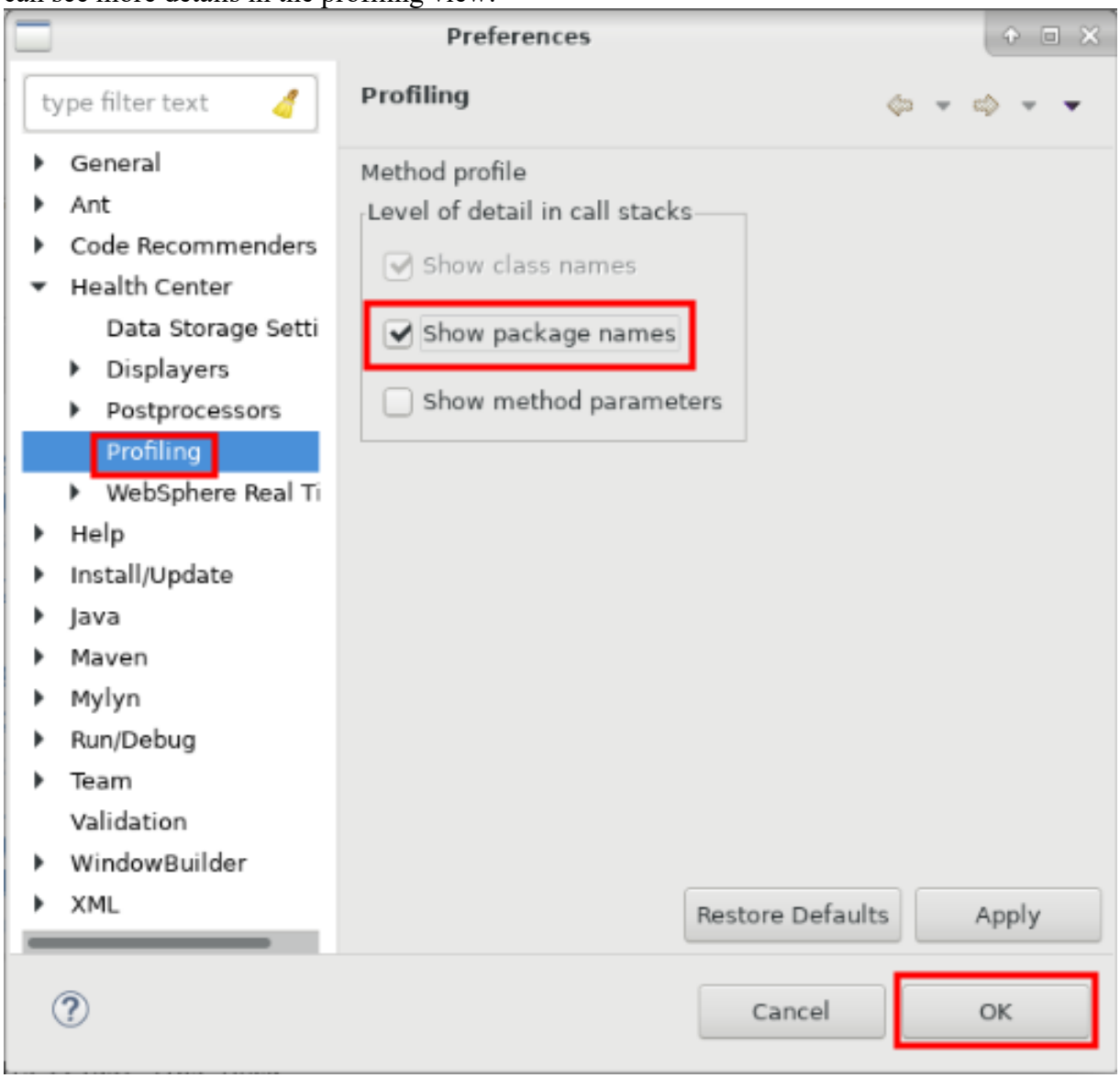
15. Change the "Start time" and "End time" to match the period of interest. For example, usually you want to exclude the start-up time of the process and only focus on user activity:



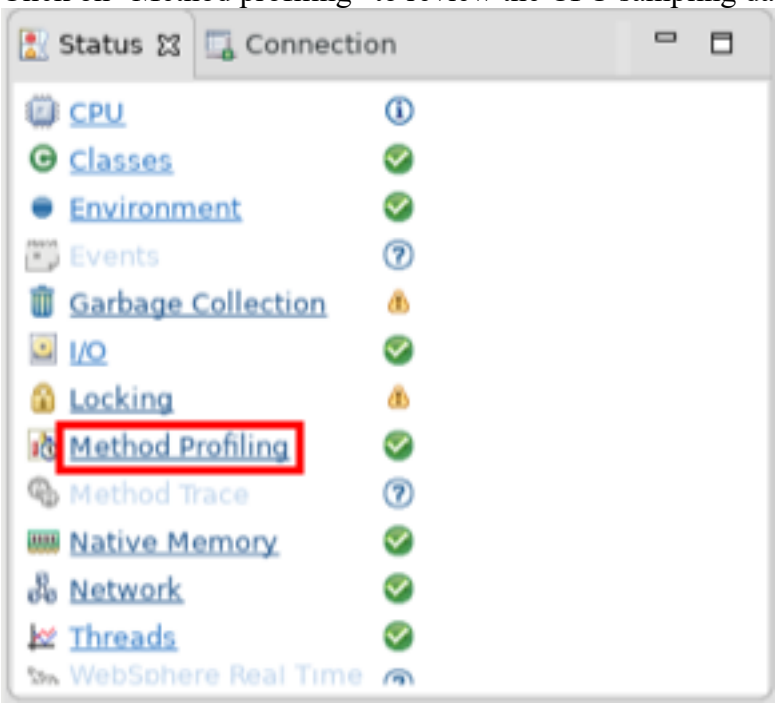
16. Clik Window > Preferences:



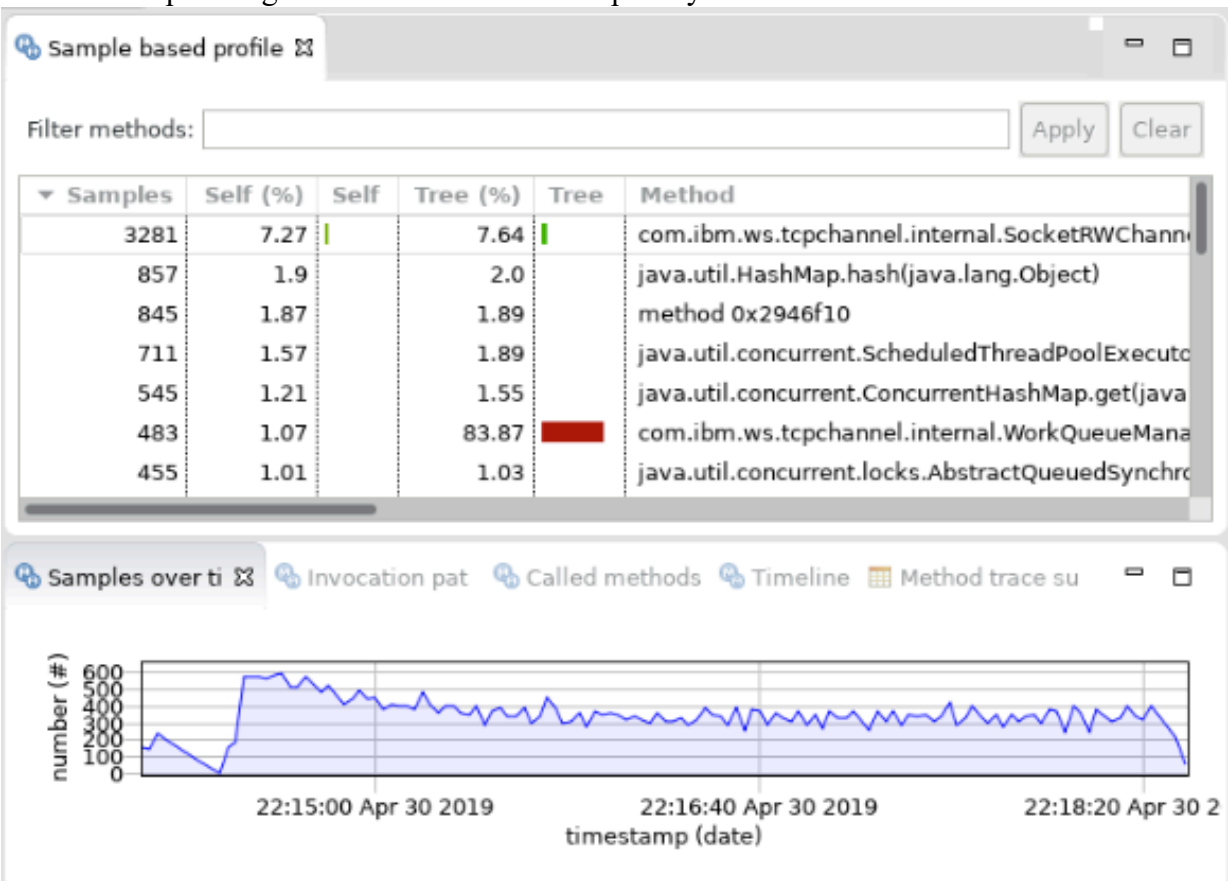
17. Check the “Show package names” box under Health Center > Profiling and press OK so that we can see more details in the profiling view:



18. Click on “Method profiling” to review the CPU sampling data:



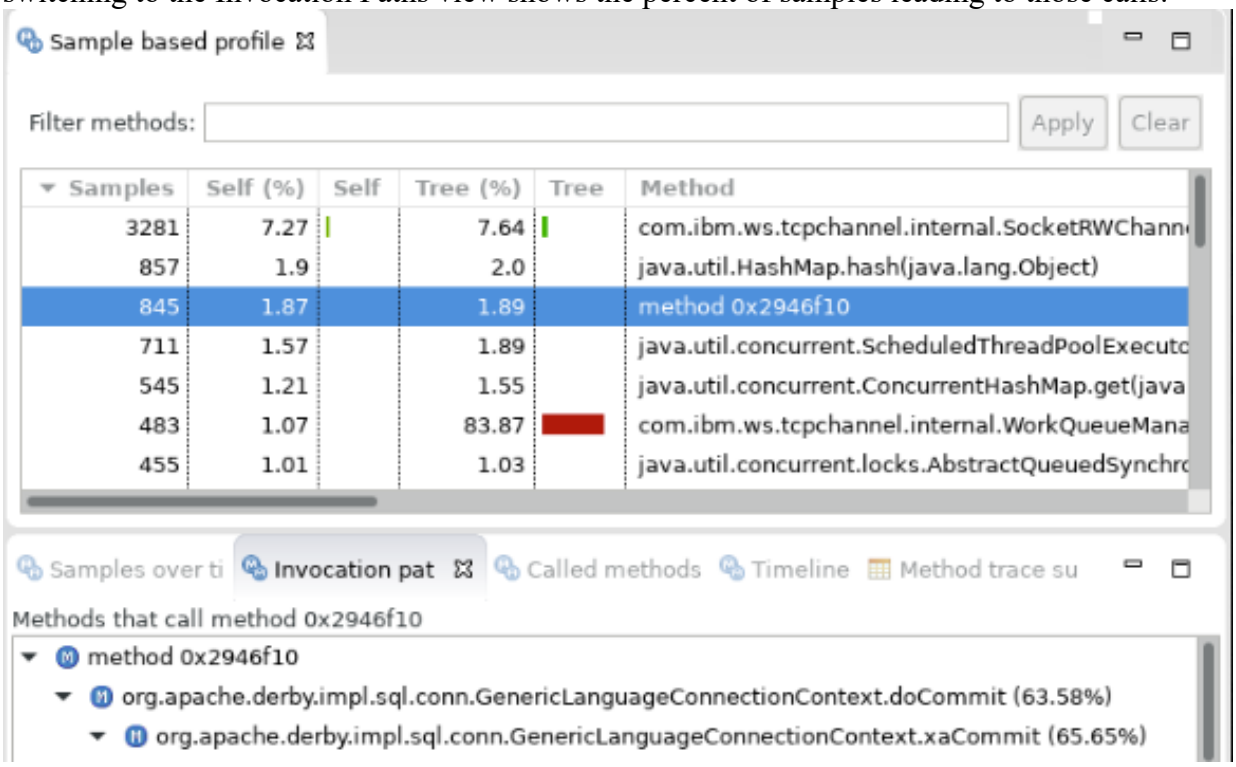
19. The Method profiling view will show CPU samples by method:



20. The Self (%) column reports the percent of samples where a method was at the top of the stack. The Tree (%) column reports the percent of samples where a method was somewhere else in the stack. Make sure to check that the “Samples” column is at least in the hundreds or thousands;

otherwise, the CPU usage is likely not that high or a problem did not occur. The Self an Tree percentages are a percent of samples, not of total CPU.

21. Any methods over ~1% are worthy of considering how to optimize or to avoid. For example, ~2% of samples were in method 0x2946f10 (for various reasons, some methods may not resolve but you can usually figure things out from the invocation paths). Selecting that row and switching to the Invocation Paths view shows the percent of samples leading to those calls:



1. In the aboe example, 63.58% of samples were invoked by org.apache.derby.impl.sql.conn.GenericLanguageConnectionContext.doCommit.

22. If you sort by Tree %, skip the framework methods from Java and WAS, and find the first application method. In this example, about 30% of total samples was consumed by com.ibm.websphere.samples.daytrader.web.TradeAppServlet.performTask and all of the methods it called. The "Called Methods" view may be further reviewed to investigate the details of this usage; in this example, doPortfolio drove most of the CPU samples.