Editing the Million-Line XML Document

Topologi Application Note

The Topologi Collaborative Markup Editor[1] is capable of editing large XML documents, which are the bane of other XML editors. This application note gives advise for handling large documents.

The Topologi Collaborative Markup Editor edits files not XML documents. The first option when considering how to handle large XML documents is whether you can use the XML entity mechanism (or perhaps Xinclude, some linking mechanism, or command-line concatenation of SGML files) to divide the large document into smaller files. Let us proceed on the assumption that you must handle large files, such as a single large document.

The tips for working with large files are:

- make sure you have enough memory
- start with a new editor session whenever possible
- open in the order of size, largest to smallest, and close in the reverse order
- use import rather than cut and paste for concatenating files

If you are using the Topologi Hierarchical Information Browser, which bundles the editor with a browser, and you have difficulties working with a large file, then use the Topologi Collaborative Editor instead.

1. This note applies to the Topologi Collaborative Markup editor, version 1.1.6 with JRE 1.4.2. See `http://www.topologi.com`

MEMORY

The Editor has been designed to handle single files of up to 5 Meg in normal use. However, this can be extended by at least a factor of ten (and probably much more) depending on three factors:

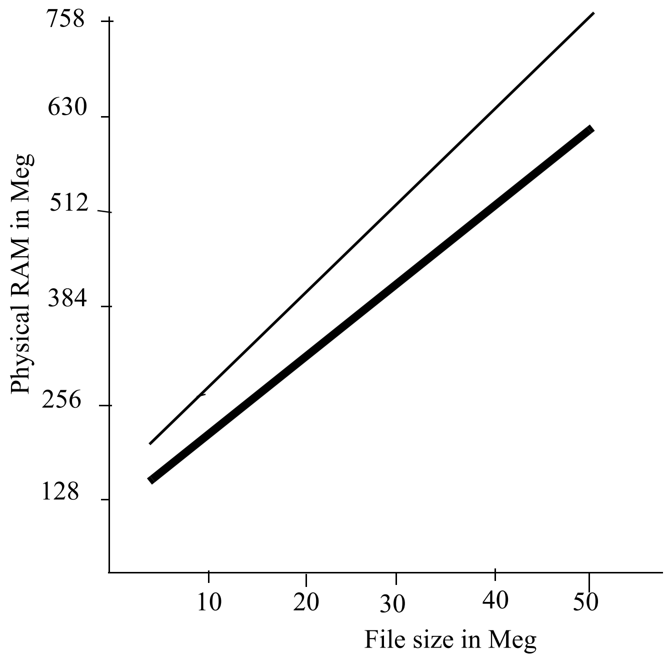
- Memory availability
- Java Heap settings
- Opening order and usage
-

MEMORY AVAILABILITY

Working with large documents requires a good amount of memory. There are two strategies for working with large documents:

- use lots of RAM, so that the entire application never requires swapping or paging when in use, or
- use lots of virtual memory, so the the application never needs to perform garbage collection.

The graph shows the approximate minimum physical memory (RAM) required to open and edit a single very large ASCII file in a new editor session.



The black line shows the memory required for a dense file with an average of 60 characters per line. The grey line shows the memory required for sparse file with an average of 20 characters per line. Windows XP users add 64 meg. Subtract 84 Meg to get the minimum heap size.

The following numbers are used to estimate memory requirements:

- *ltotal* = total number of lines
- *smax* = size of the (largest) document, in characters[1]
- *run* = ltotal*100 + smax*2
- *temp* = smax*6
- *overhead* = 20 meg
- *java* = 20 meg

The amount of physical RAM required is the greater of *run* + *temp* + *overhead* + *java*. So to open a single, million-line file that has 25 million characters requires 340 Meg RAM. Note that this is RAM in addition to the base RAM needed for your operating system: for Windows and Linux, this is 64 Meg RAM; for Windows XP this 128 Meg.

Conventional Linux wisdom is that swap space should be at least twice as large as physical RAM. For Windows, check that the System Control Panel allows at least *run + temp + overhead + java* plus the process sizes of any other processes that may also be running. If you plan to use the "use lots of virtual memory" strategy, a rule of thumb is to make sure you have at least twice as much as *run + temp + overhead* (for all files to be opened): err on the side of having too much swap space or virtual memory.

The reason for the two strategies is because garbage collection and paging interact badly: garbage collection may required a complete sweep thoughout the allocated memory "heap", perhaps with random access. This forces the virtual memory system to swap most of the pages in memory: the symptom is disk thrashing: large pauses where not much CPU activity occurs but much disk activity.

Many modern applications preload themslves into computer's memory so that they start up faster. In a computer with adequate RAM this is no harm; the applications will be swapped or paged to disk if physical RAM is not needed, and the user will only experience a slower startup. [2]However, if your remaining paging space is small, or the other applications are active, and you have difficulties even with the amounts of memory suggested in this application note, try killing these kinds of preloaded processes; the proceses may include web browsers, office applications, and other peer-to-peer file sharing programs.

1. If you are using ASCII or a Western language, this will probably be the same as the number of bytes in the file.
2. The Topologi Collaborative Markup Editor allows a version of this: starting it with the -exit option causes the editor to run then immediately exit: on some operating systems, the JVM and editor files will be cached in memory which can result in slightly faster startup times.

JAVA HEAP SETTINGS

The Topologi Collaborative Markup Editor is a Java program. Java allows you to limit the memory (“heap”) size, to prevent the application from greedily using all the available memory. The Windows executables come with fixed maximum heap sizes; however, the shell scripts (.sh for Linux or .bat for Windows) can be altered to allow more memory.

To change the heap settings, change the -Xmx setting for the Java, and also the -Xmx setting provided as an argument to the Java main() method being invoked. The first number sets the heap size, the second number is used by the editor to provide warnings about memory allocation. The two numbers should be the same (some versions of Java do not make the maximum heap size available to the

If *run+ temp + overhead.* is less then the size of available physical memory (i.e. the physical memory after the 84 Meg OS penalty has been subtracted--148 Meg for Windows XP), then use the “use lots of RAM” strategy. Set the -Xmx to between *run+ temp + overhead* and the size of the available physical memory. So in our example, a setting of 350000000 should be enough; our example PC has 512 Meg RAM, so we subtract 84 Meg and come up with a number of 428 Meg: this should reduce the number of times we need to perform garbage collection, but reduce the risk of swapping.

If you cannot fit into physical memory, you must use the “use lots of virtual memory” strategy. In this case, set the heap size to a large enough size that you can open your monster files, edit them, save them and quit without having to perform garbage collection. So, in our example, a setting of 700000000 should be enough; double the minimum required. When you take this strategy you will notice that some parts of the editor, such as the initial read of files, speeds up considerably: the downside is that if you do end up using so much virtual memory that the Java Virtual Machine needs to perform garbage collection, it will take several minutes probably with disk thrashing. So the “use lots of virtual memory” strategy is not really suited for prolonged editing sessions.

OPENING ORDER AND USAGE

The order in which large files are opened also impacts how much memory is required. When a large file is opened, validated or saved, it requires the *temp* amount of RAM but then this becomes available to subsequent processes.

So if you need to open several large files, open the **largest one first**, so that it has the most memory available for its *temp* requirements. Then open the others in order of their size. Only open files as you require them. Doing this should allow you to open a large file, then open and edit smaller files needed to work on the large file.

By the same logic, when **close files in the order of size**, starting with the smallest. That way, larger files will have more free memory available to them for their *temp* requirements.

Of course, you want to have the maxiumum memory available: so when opening large files, start with a fresh editor session — quit the editor if it is open and then close it, just in case there has been any memory leaks. (The editor has been profiled to prevent memory leaks, however when you are working with large files it is prudent to start with an empty slate.)

Lets say I have -Xmx of 350000000 and adequate physical RAM. By or calculations above, this should be just enough to open our million-line, 25Meg example file. However, once that file is open, it uses about 150M of *run* but it leaves 150M of *temp* space available. I can then open a smaller file using that 150Meg as the extra *run* and *temp.* So after I have opened the 1,000,000 line file, I have enough space for a 500,000 line file, then a 250,000 line file, then a 125,000 line file, and so on. (So, optimally packed, the editor can actually edit twice as many lines as in the largest possible file for a given memory. Geek note: for some versions of the Java Virtual machine, the memory requiements of a copying garbage collector may make it inadvisable to pack too much.)

However, if I opened the 500,000 line file first, I will not have enough space to then open the 1,000,000 line file. This order-dependent behaviour only affects files that are large compared to the available memory, and may be mystifying to users.

EXPENSIVE OPERATIONS

The figures general should give a good indication for most simple uses of large documents: opening, minor edits, search and replace, validation and saving.

However, some operations may be more expensive, and so require more memory (RAM and heap):

- large or multiple editing operations involving section moves, because the Undo mechanism may store the old lines and the new ones,
- use of XSLT, which may build a document tree that can be quite large indeed: this includes preview transformations and Schematron validation,
- printing and print previews.

These operations have not been benchmarked to give estimate of memory requirements.

JOINING SEVERAL FILES

If you have several files that you wish to join together into a single file, the most memory-efficient way to do this is to us the **Tools>Import>File** menu.

If you open each file then use cut-and-paste, you very rapidly blow-out your memory requirements. This is because you need an extra *run* amount of memory for the document being cut from, plus up to 2 times the number of characters for the strings copied to and from the clipboard.

If you need to copy-and-paste, and you are pushing the limits of RAM, try opening the documnent to be copied from, perform the copy operation, close the document, open the document to be copied to and perform the paste operation.

STACK SIZE

Some Java Virtual Machines allow the stack space available for threads to be set. The Topologi Collaborative Markup Editor uses dozens of threads, however it does not require a large stack space for any particular thread. The size for each thread can be set smaller than the default size.

GARBAGE COLLECTION AND FRAGMENTATION

Another aspect of performance is memory fragmentation. Different versions of the Java Virtual machine use different strategies for collecting garbage (i.e. figuring out which memory is available for use), but it is possible that even though there is enough total memory free to open another file, there is not enough contiguous (adjoining, contiuous) space in the heap to allow it. The tips above (start with a fresh session, open and close files in a particular order, close unneeded files) will help reduce this potential problem, which can be very frustrating for users, and only appears when working with relatively large files. Furthermore, some garbage collectors are conservative, and may under report the amount of memory available, which compounds the problem.

SINGLE-LINE FILES

What about the single line, million character file?

The Topologi Collaborative Markup Editor was not designed to display long lines; it will be slow and tedious.

When opening such a file, you should select to break the lines (and indent them, if you wish) and when closing it you can select to convert lines to spaces.