APACHE SLING & FRIENDS TECH MEETUP

10-12 SEPTEMBER 2018

SLING MEMORY DEEP DIVE

@ValentinOlteanu, @AdobeCH

Speaker notes

Physical memory is one of the most critical system resources, along with CPU, that a Sling application relies on to achieve maximum performance and best response times. It is commonly accepted that sling applications perform the best when the whole state is in memory and any disk/remote access introduces huge penalties, thus Sling and Oak rely on a bunch of in-memory caches to deliver the desired performance.
Understanding where and how this is used is a very important know-how for operating large scale deployments, especially when you need to address sizing, optimising and vertical scaling. This session takes a holistic approach to understanding RAM consumption by offering a detailed split-view of all the various flavours of a system's memory used by Sling.

adaptTo()

MEMORY

how to avoid OOM?

MEMORY

how much page cache?

MEMORY

how much heap?

MEMORY

how much RAM?

MEMORY

Speaker notes

Memory is a wide and generic subject on which one could speak for days, depending on the perspective and the level of details that are discussed. To scope the presentation and to have some practical takeaways at the end, I'm focusing on a few questions I've often received when talking about this. The questions are usually centred around sizing:

* how much RAM do I need for my deployment?
* how do I configure the JVM to optimally distribute the available memory?

Answering these questions will not only ensure high performance, but also stability by avoiding crashing the java process and potentially losing data.

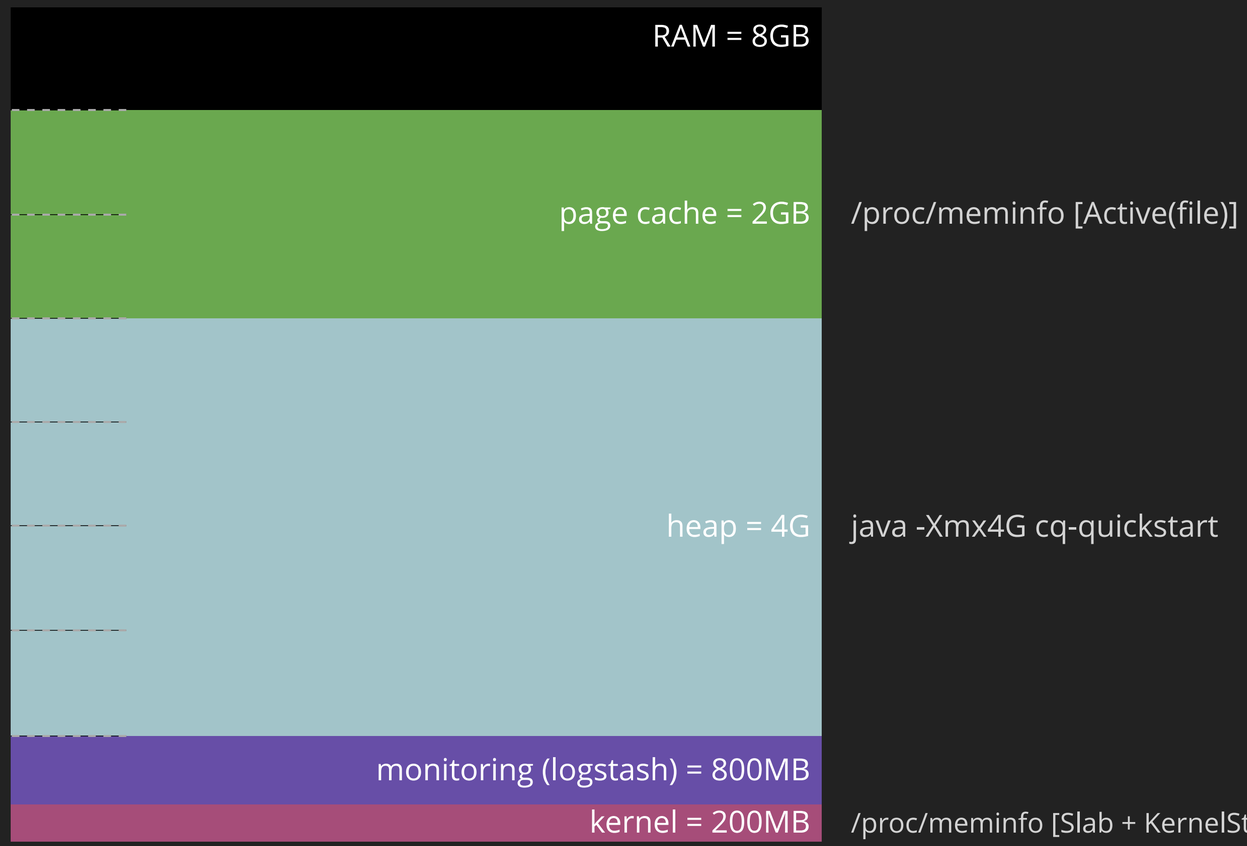
adaptTo()

Full system dissection

Speaker notes

To address the subject from a holistic perspective, I'm going to take a top-down approach: first take a look at the overall system and identify the biggest components and then zoom in each of them and see how they break down further.
For illustrating the concepts with some real-world numbers, I've used a typical AEM deployment for extracting the exact numbers. The system was running in a Linux VM, with a default AEM installation and some constant incoming traffic for keeping the internals warmed up.

adaptTo()



Speaker notes

For identifying which processes have pages committed in the physical memory, I've extensively used standard linux tools, such as `ps`, `pmap` and the `/proc/meminfo` virtual file. The exact commands are gathered in a script at https://github.com/volteanu/sling-memory-deep-dive/blob/master/scripts/cqmeminfo.sh which you can download and adapt for inspecting your own system.

Running the above script on my test system, which has 8GB of RAM, yielded the following consumers:
- Linux kernel and assisting processes (e.g. monitoring apps) occupy about 1GB of the physical memory
- AEM java process committed 4GB, which corresponds exactly to the max heap size parameter (-Xmx) passed to the java command
- The page cache, managed by the linux kernel and intensively used by AEM for optimising file access, especially for caching segment tar files
- Linux kernel always keeps some free RAM at hand in case new processes are started and need to allocate some memory

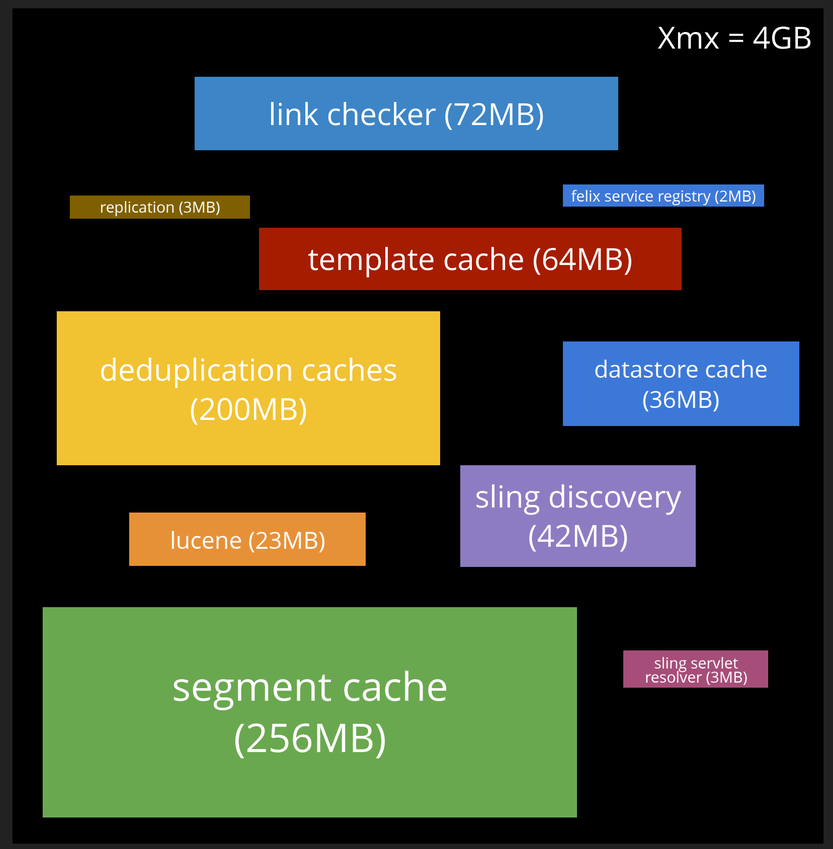
Finding the right balance between heap and page cache is necessary to operate the application at the highest performance while keeping the cost under control. In this case, for the given load and content size, allocating 4GB to the heap and leaving 2GB for the page cache proved to be the right choice to have the application running without running out of (heap) memory and with a sufficiently big cache to ensure small response times.

adaptTo()

Heap dissection

Speaker notes

The content of the heap can be easily inspected by taking a heap dump of a running process (using for example `jhat`) and then loading the generated file in a specialised program. For this exercise, I've used the Eclipse Memory Analyser (https://www.eclipse.org/mat/), which helped me identify the biggest modules in my application from the heap footprint point of view.



Speaker notes

As mentioned before, the memory footprint for the heap was precisely equal to the max heap parameter. This is expected, since the JVM is always committing the maximum allocated amount of memory and then manages it by itself. So, even if you're not always using all the heap, the java process will keep all of it reserved in case it's needed at some point.

The first thing to notice after taking the heap dump of live objects is that the generated file size is smaller (sometimes much smaller) than the total heap size. This is explained by the dynamic nature of java memory management, which means a part of the heap is not accessible and can be garbage-collected anytime.

After grouping the accessible objects into "components" (based on the package), I've been able to identify the following, in descending order by size:
- segment cache takes exactly 256MB (by design/configuration)
- deduplication cache - used by Online Revision Cleanup - takes 200MB (also by design)
- link checker has some sort of cache that keeps 72MB of objects in this usecase
- template cache, also used by the Oak segment module, allocates 64MB
- Sling discovery holds a Json Factory reference that takes 42MB
- Datastore manages a cache of about 36MB
- Lucene requires about 32MB for index caching
- Other parts worth mentioning: replication, service registry and sling servlet resolver are also present, but with smaller amounts (around 2-3MB)

One important characteristic of these modules is how the memory usage changes over time. There are two categories in which we can split them:
1. Fixed size: usually size-bound caches, such as the segment cache, deduplication caches
2. Variable size: basically all the components that allocate memory proportionally to the number of concurrent requests handled by the system. While some components are in my example quite small in heap footprint, given the relatively low system load, they can quickly grow and put a lot of stress on the heap if the concurrent traffic increases.

The takeaway of this introspection should be: Sling and Oak require a fixed, minimum amount of heap, on top of which a dynamic part must to be allocated depending on the load. Also, custom bundles will add some pressure, so don't forget to take them into consideration when sizing your instance.

adaptTo()

Page cache dissection

Speaker notes

Although not Sling or Oak specific, the Linux page cache is a critical mechanism for ensuring a smoothly running application. Sling uses it as a transparent cache for frequently used files, such as bundles files, binaries etc. Out of these, the most frequently accessed files are, by far, the tars used to store the node store. Because these have a considerable size, and the application (JVM) doesn't have any control on the cache, the system administrator needs to ensure proper conditions for this cache to perform.

During normal operations

https://asciinema.org/a/200219

Speaker notes

To help understand how this cache is exercised, I've focused on the segmentstore and tried to visualise which parts of the repository are loaded into memory in a dynamic recording. For this, I've used https://github.com/hoytech/vmtouch to list the resident pages of each tar file at regular time intervals and create a time-lapse animation.

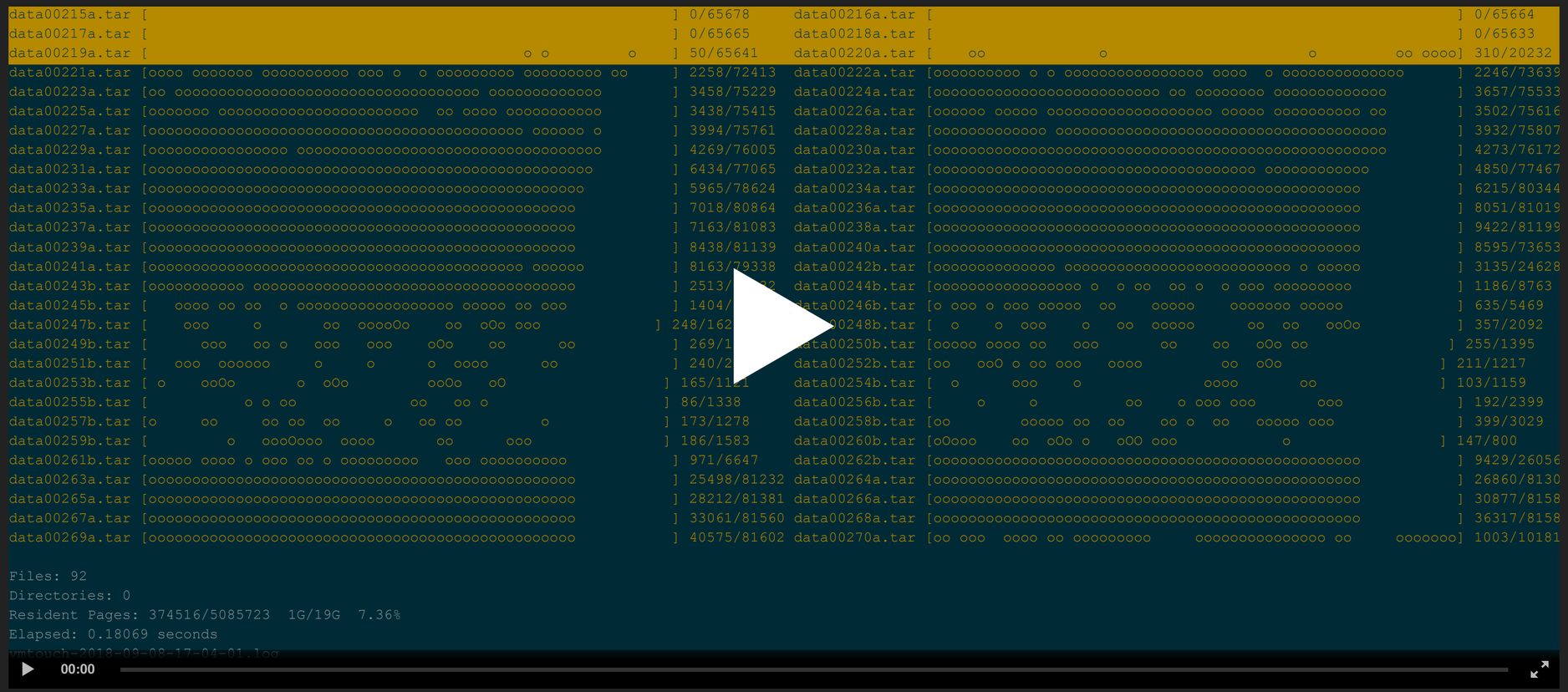
vmtouch outputs, for each file, the number of resident pages out of the total number of pages in the file. Besides it also illustrates the exact area of the file that resides in memory, giving a nice overview of which areas in the segmentstore are being cached at a certain timestamp.

For better understanding the output, I've also colour-coded the tars, depending on the (OnRC) generation: files with the same colour belong to the same generation and there are usually two generations in the segmentstore (old and current). Inside the same generation, the files can be further split between tars generated by the Online Revision Cleanup process (coloured background) and tars generated by the application (coloured text).

During normal operations (that is during the day, when no maintenance task is running and the system runs without problems), the following is expected:
- the old generation is not loaded into memory; these files are not used anymore since they are superseded by the newer generation and will be deleted at the next revision cleanup
- from the current generation, there are files that are frequently accessed so these are cached, but there will always be a part that is not accessed; that can be linked to two causes:
  1. there are newer revisions of the nodes, so some segments holding the previous revisions are not accessible anymore, or
  2. there's dormant content in the repository that is not accessed by the application (e.g. versions, audit logs, disabled indices)

Over time, in normal conditions, the set of cached tars should be quite stable, so the disk is not accessed frequently. Also, the set of cached files is usually called the "working set" and is used for properly sizing the RAM you should let available for the operating system to be allocated for the page cache.

When "trashing"



https://asciinema.org/a/200570

Speaker notes

When pages are constantly evicted and then reloaded, the system enters a state of "trashing": almost every request results in a disk access, drastically impacting the overall performance. This is visible in the animation, where many pages change the state from one minute to the other. If that's the case for your system, you should consider resizing your instance to bring it to a more stable state.

During Online Revision Cleanup

https://asciinema.org/a/200223

Speaker notes

The Online Revision Cleanup (OnRC) is a special phase in which this cache is going through an intensive refresh.
As noted above, before OnRC, the situation looks like this:
- generation n-1 (old) is not memory mapped since it's not accessed anymore
- generation n (current) is partially loaded, defining the current "working set"

When OnRC starts:
- generation n is fully traversed; this is visible in the animation as a rolling loading/unloading of the tar files from this generation
- generation n+1 is created as a compact replica of generation n; you will notice the fast creation of new files
- generation n-1 is safely deleted at the end

After OnRC:
- parts of generation n (which is now old) are still memory mapped because they are accessed by long-running tasks in the application that hold old references; as time passes, the tasks complete and generation n is progressively evicted until no page from these tars is present in memory
- generation n+1 becomes the current generation and is loaded according to the usage; at the end we're reaching the same state as before OnRC, completing the daily cycle

How much?

It depends!

Heap

- on your system load (traffic)

Page cache

- on your content size (repository)
- on your working set

Speaker notes

No notes on this slide.

slides and goodies available at

https://github.com/volteanu/sling-memory-deep-dive/

Speaker notes

No notes on this slide.