ScalabilityIssues1.4

Information Below pertains to DSpace 1.4 and previous

The below scalability issues pertain to DSpace 1.4.x and previous versions. **This information is out-of-date as of DSpace 1.5**

As our DSpace deployments get bigger and bigger, we inevitably run into scaling issues. Please add any you find here, or discuss how to address them.

Hint: When trying to find what's causing a problem (e.g. long processing time), you can find out a lot by upping your DSpace's logging level to DEUG. This increases the amount of information saved to dspace/log/dspace.log. See TechnicalFaq - "Setting logging level up to DEBUG".See also: HowToPerformanceTuneForDspace.

Contents

In-memory object cacheorg.dspace.core.Context contains an 'object cache'. Every org.dspace.content object instantiated gets placed in this cache. Two reasons for this:* To make sure the same object isn't instantiated twice. e.g. if one part of the code instantiates item hdl:123.456/789 and modified it, and another part instantiates the same object, it will not see that update.

- Efficiency in a similar scenario (two parts of the code instantiating same object).

However during certain operations, particularly those iterating over a large number of objects, this cache can get huge and cause memory problems, since it is never cleaned out. Note that this cache exists only for the lifetime of the Context object, which in general is a single HTTP transaction or batch import. One way to alleviate the problem a little is to use a WeakHashMap instead of a regular HashMap, as theoretically the garbarge collector will clear out objects which no longer have outside references. However anecdotal evidence suggests this is not very effective.

Notes on this issue

As of DSpace 1.5 there is a Context.clearCache() method available to developers. While this is not utilised by the existing code base, it can be used by new developments which require large turn over of un-reused data to minimise the memory usage of this mechanism by calling this method regularly.

Loading 'sub-objects' into memory

In certain cases, classes in org.dspace.content (org.dspace.content.Itemorg.dspace.eperson), when instantiated, also instantiate and read in from the database 'sub-objects'. For example, when you instantiate an object, it in turn instantiates the org.dspace.content.Bundle objects within that, which in turn instantiates the appropriate org.dspace.content.Bitstream objects. This amounts to a fair amount of database reads and memory use (combined with above object cache issue), which can impact performance when iterating over large numbers of objects.

The main reasons for this were:

- Convenience to callers.
- So that one can instantiate an

```
Item
```

  and pass it to a JSP, which can safely invoke

```
getBundles
```

on that without having to worry about trapping an

```
SQLException
```

. i.e. to fit in with an MVC style of interaction.
It was a case-by-case judgement call as to which object constructors did this, based on a guess of potential performance vs. convenience trade-off. e.g.

```
Collection
```

objects don't instantiate all `Item` objects within them. This would clearly cause performance problems.

Notes on this issue

Development work to clean up the Item (and other content) object data access work means that it will soon be possible to work with Proxy classes for these objects which behave in more sensible ways. Therefore, if the operation is for a JSP then a fully loaded Item might be appropriate, but for other operations (such as browsing, bulk importin, etc) different implementations of Item might be appropriate. This development work should be available in 1.5 or 1.6

History system

Whenever an object is modified (e.g. a

```
Collection
```

), the history system appears to iterate over every object within that collection. (Compounded by above sub-object problem.)

Community/collection page

With a large number of communities and collections, the Community/collection page takes a long time to load. This is probably caused in part by 'sub-object' problem above: Each

```
Collection
```

object instantiated loads in logo bitstream, any metadata template (for new submissions), and associated e-person groups (workflow steps, submitters, admins).

From DSpace@Cambridge

Our DSpace@Cambridge archive reached over 100,000 items last week, and we're running into serious performance problems (actually, we've seen performance degrade since we had a couple of tens of thousands of items). I'll give an overview of some of the issues, and where possible, the workarounds we devised for them.
A lot of these problems stem from the design of the database, some seem to result from leaks in the code of rather inefficient database queries being run. We've run PostgreSQL on bigger databases than this, and are quite certain that it isn't the cause.
(I'd already run an analysis of the database queries performed by DSpace a while back, and all indices that might help have already been created.)

Memory/DB pool leak

System running out of memory and/or the DCP Commons database pool being exhausted - typically happens while being indexed by search engine crawlers from Google/MSN/Yahoo/etc.
This happened to us well before we reached 100,000 items. The reason seems to be in the design of the browse pages: we suspect that there's a database pool connection leak here, but these pages also perform a lot of queries, and/or are in se uncacheable. For example, if you look at an item, then go to the "browse by author" page, the author of the item you last looked at will be highlighted in the browse page. To search engines, that means that these browse pages are different from the ones they've cached when following the "browse by author" links from the DSpace front page, essentially creating a new set of browse pages for every item. As you can imagine, this causes a huge load on the system.

Step 1 of the workaround: create a robots.txt with the following in it:

```
User-agent: *
Disallow: /browse-*
Disallow: /items-by-author*
```

This will stop search engine crawlers from going over browse pages (in our experience, most crawlers are well enough behaved to honour robots.txt, although MSN's still occassionally ignores it). But in order to allow your DSpace instance to get indexed, however, you want to give search engine crawlers a list of the items in your archive. We've solved this by creating a little Perl script that gets all the items from the database, and dumps them to a html file which we link to from the front page.
See http://www.dspace.cam.ac.uk/ - at the bottom there's a link for "All content" (added to layour/footer-default.html). Writing some Perl to dump all handles from your database to a file such as this is trivial, and could be done in Java or shell script just as easily and is "left as an exercise for the reader"... (a plain
"

```
SELECT handle_id FROM handle
```

" dumped with some html around
the result rows will tend to be Good Enough to do the trick).

Notes on this issue

This may be out of date. DSpace 1.4.2 does not exhibit obvious database leaks.

OAI harvester

OAI PMH virtually hangs the system when being asked anything. We're pretty sure this is caused by the database design, which lacks the structure for the database engine to efficientely perform joins etc. For now we've disabled the OAI harvesting interface completely.

Batch imports

Batch imports have become excruciatingly slow, and large batches (several thousand items) can't be run at all, because somewhere there's a memory leak. We've pretty much stopped importing items for now, since the importer takes over 8 seconds per item, and that's unworkable (we have at least 200,000 more items lined up to be imported).
No workaround (yet).

Browse pages

Browse pages have become very slow - in part because the database spends quite a bit of time calculating the "showing items 1-20 of FOO" text at the top (see earlier notes about database design).
No workaround (yet).

Notes on this issue

As of DSpace 1.5 there have been significant improvements to the end user experience for browsing large datasets. At Imperial College we have seen smooth performance of the front end browse using this code for up to 122,000 records. The down side is that the indexing process has an extremely non-linear response to archive size, and may become unusable as the archive expands.

From DSpace 1.41 @ PoisonCentre.be

42 thousands items imported from PubMed: 175 thousands references to authors, one million references to subjects.

The hierarchical tree of 44 thousands MESH (Medical Subjects Headings) terms has been defined as a controlled vocabulary.

Postgresql Vacuum/Analyse solves many performance after batch import. Probably Analyse is the key factor because Postgres can then better optimize the execution of queries.

Browse is too slow but also, in general, the application workflow is not well suited for searching in a big database: a design-a-thon around this would be necessary. We will provide you with the feedback of our users.