NearRealtimeSearchTuning

Original text of this wiki page is from Peter Sturge and can be found in this thread

This page discusses tuning the Solr 1.4 and 3.x branches. NearRealtimeSearch is a separate project in Solr4.0.

Solr with frequent commits

Example environment: commit every 30secs, large index (>20 mio docs), heavy use of facets, solr 1.4.1 or branch_3x

Tuning an index where you are adding new data, never changing existing data.

Symptoms

If anyone has used faceting in searches where you are also performing frequent commits, you've likely encountered the dreaded OutOfMemory or GC Overhead Exeeded errors. In high commit rate environments, this is almost always due to multiple 'onDeck' searchers and autowarming - i.e. new searchers don't finish autowarming their caches before the next commit() comes along and invalidates them. Once this starts happening on a regular basis, it is likely your Solr's JVM will run out of memory eventually, as the number of searchers (and their cache arrays) will keep growing until the JVM dies of thirst. To check if your Solr environment is suffering from this, turn on INFO level logging, and look for: 'PERFORMANCE WARNING: Overlapping onDeckSearchers=x'. In tests, we've only ever seen this problem when using faceting, and facet.method=fc.

Some solutions to this are:

- Reduce the commit rate to allow searchers to fully warm before the

next commit

- Reduce or eliminate the autowarming in caches Both of the above

The trouble is, if you're doing NRT commits, you likely have a good reason for it, and reducing/elimintating autowarming will very significantly impact search performance in high commit rate environments.

Solution

Here are some setup steps we've used that allow lots of faceting (we typically search with at least 20-35 different facet fields, and date faceting/sorting) on large indexes, and still keep decent search performance:

1. Play with different facet methods: facet.method=enum or facet.method=fc
   Taken from Erick Erickson: *"I'd stick with fc for the time being and think about enum if you have a good idea of what the number of unique terms is or you start to need to finely tune your speed."*
   ⚠ per segment faceting] Or even use [http://lucene-eurocon.org/slides/Solr-15-and-Beyond_Yonik-Seely.pdf with facet.method=fcs
2. Secondly, we've found that LRUCache is faster at autowarming than FastLRUCache - in our tests, about 20% faster. Maybe this is just our environment - your mileage may vary. So, our filterCache section in solrconfig.xml looks like this: <filterCache class="solr.LRUCache" size="3600" initialSize="1400" autowarmCount="3600"/>
3. For a 28GB index, running in a quad-core x64 VMWare instance, 30 warmed facet fields, Solr is running at ~4GB. Stats filterCache size shows usually in the region of ~2400.
4. It's also a good idea to have some sort of firstSearcher/newSearcher event listener queries to allow new data to populate the caches. Of course, what you put in these is dependent on the facets you need/use. We've found a good combination is a firstSearcher with as many facets in the search as your environment can handle, then a subset of the most common facets for the newSearcher.
5. We also set: <useColdSearcher>true</useColdSearcher> just in case.
6. Another key area for search performance with high commits is to use 2 Solr instances - one for the high commit rate indexing, and one for searching. The read-only searching instance can be a remote replica, or a local read-only instance that reads the same core as the indexing instance (for the latter, you'll need something that periodically refreshes - i.e. runs commit()). This way, you can tune the indexing instance for writing performance and the searching instance as above for max read performance.

Using the setup above, we get fantastic searching speed for small facet sets (well under 1sec), and really good searching for large facet sets (a couple of secs depending on index size, number of facets, unique terms etc. etc.), even when searching against largeish indexes (>20million docs). We have yet to see any OOM or GC errors using the techniques above, even in low memory conditions.

Notes

1. Regarding the point of two solr instances: one for the high commit rate indexing, and one for searching (read only == RO)
   - You can run multiple Solr instances in separate JVMs, with both having their solr.xml configured to use the same index folder. You need to be careful that one and only one of these instances will ever update the index at a time. The best way to ensure this is to use one for writing only, and the other is RO and never writes to the index. This RO instance is the one to use for tuning for high search performance. Even though the RO instance doesn't write to the index, it still needs periodic (albeit empty) commits to kick off autowarming/cache refresh.
   - Depending on your needs, you might not need to have 2 separate instances. We need it because the 'write' instance is also doing a lot of metadata pre-write operations in the same jvm as Solr, and so has its own memory requirements.
   - We use sharding all the time, and it works just fine with this scenario, as the RO instance is simply another shard in the pack.
2. The first one to note is that the techniques/setup described in this thread don't fix the underlying potential for OutOfMemory errors - there can always be an index large enough to ask of its JVM more memory than is available for cache. These techniques, however, mitigate the risk, and provide an efficient balance between memory use and search performance. There are some interesting discussions going on for both Lucene and

Solr regarding the '2 pounds of baloney into a 1 pound bag' issue of unbounded caches, with a number of interesting strategies. One strategy that I like, but haven't found in discussion lists is auto-limiting cache size/warming based on available resources (similar to the way file system caches use free memory). This would allow caches to adjust to their memory environment as indexes grow.

3. use 'simple' for lockType
4. maxWarmingSearchers = 1 as a way of minimizing the number of onDeckSearchers