ImproveIndexingSpeed

How to make indexing faster

Here are some things to try to speed up the indexing speed of your Lucene application. Please see ImproveSearchingSpeed for how to speed up searching.

- **Be sure you really need to speed things up.**
  Many of the ideas here are simple to try, but others will necessarily add some complexity to your application. So be sure your indexing speed is indeed too slow and the slowness is indeed within Lucene.
- **Make sure you are using the latest version of Lucene.**
- **Use a local filesystem.**
  Remote filesystems are typically quite a bit slower for indexing. If your index needs to be on the remote fileysystem, consider building it first on the local filesystem and then copying it up to the remote filesystem.
- **Get faster hardware, especially a faster IO system.**
  If possible, use a solid-state disk (SSD). These devices have come down substantially in price recently, and much lower cost of seeking can be a very sizable speedup in cases where the index cannot fit entirely in the OS's IO cache.
- **Open a single writer and re-use it for the duration of your indexing session.**
- **Flush by RAM usage instead of document count.**
  For Lucene <= 2.2: call writer.ramSizeInBytes() after every added doc then call flush() when it's using too much RAM. This is especially good if you have small docs or highly variable doc sizes. You need to first set maxBufferedDocs large enough to prevent the writer from flushing based on document count. However, don't set it too large otherwise you may hit LUCENE-845. Somewhere around 2-3X your "typical" flush count should be OK.
  For Lucene >= 2.3: IndexWriter can flush according to RAM usage itself. Call writer.setRAMBufferSizeMB() to set the buffer size. Be sure you don't also have any leftover calls to setMaxBufferedDocs since the writer will flush "either or" (whichever comes first).
- **Use as much RAM as you can afford.**
  More RAM before flushing means Lucene writes larger segments to begin with which means less merging later. Testing in LUCENE-843 found that around 48 MB is the sweet spot for that content set, but, your application could have a different sweet spot.
- **Turn off compound file format.**
  Call setUseCompoundFile(false). Building the compound file format takes time during indexing (7-33% in testing for LUCENE-888). However, note that doing this will greatly increase the number of file descriptors used by indexing and by searching, so you could run out of file descriptors if mergeFactor is also large.
- **Re-use Document and Field instances**
  As of Lucene 2.3 there are new setValue(...) methods that allow you to change the value of a Field. This allows you to re-use a single Field instance across many added documents, which can save substantial GC cost.
  It's best to create a single Document instance, then add multiple Field instances to it, but hold onto these Field instances and re-use them by changing their values for each added document. For example you might have an idField, bodyField, nameField, storedField1, etc. After the document is added, you then directly change the Field values (idField.setValue(...), etc), and then re-add your Document instance.
  Note that you cannot re-use a single Field instance within a Document, and, you should not change a Field's value until the Document containing that Field has been added to the index. See Field for details.
- **Always add fields in the same order to your Document, when using stored fields or term vectors**
  Lucene's merging has an optimization whereby stored fields and term vectors can be bulk-byte-copied, but the optimization only applies if the field name -> number mapping is the same across segments. Future Lucene versions may attempt to assign the same mapping automatically (see LUCENE-1737), but until then the only way to get the same mapping is to always add the same fields in the same order to each document you index.
- **Re-use a single Token instance in your analyzer**
  Analyzers often create a new Token for each term in sequence that needs to be indexed from a Field. You can save substantial GC cost by re-using a single Token instance instead.

- **Use the char[] API in Token instead of the String API to represent token Text**
  As of Lucene 2.3, a Token can represent its text as a slice into a char array, which saves the GC cost of new'ing and then reclaiming String instances. By re-using a single Token instance and using the char[] API you can avoid new'ing any objects for each term. See Token for details.

- **Use autoCommit=false when you open your IndexWriter**
  In Lucene 2.3 there are substantial optimizations for Documents that use stored fields and term vectors, to save merging of these very large index files. You should see the best gains by using autoCommit=false for a single long-running session of IndexWriter. Note however that searchers will not see any of the changes flushed by this IndexWriter until it is closed; if that is important you should stick with autoCommit=true instead or periodically close and re-open the writer.
- **Instead of indexing many small text fields, aggregate the text into a single "contents" field and index only that (you can still store the other fields).**
- **Increase mergeFactor, but not too much.**
  Larger mergeFactors defers merging of segments until later, thus speeding up indexing because merging is a large part of indexing. However, this will slow down searching, and, you will run out of file descriptors if you make it too large. Values that are too large may even slow down indexing since merging more segments at once means much more seeking for the hard drives.
- **Turn off any features you are not in fact using.**
  If you are storing fields but not using them at query time, don't store them. Likewise for term vectors. If you are indexing many fields, turning off norms for those fields may help performance.
- **Use a faster analyzer.**
  Sometimes analysis of a document takes alot of time. For example, StandardAnalyzer is quite time consuming, especially in Lucene version <= 2.2. If you can get by with a simpler analyzer, then try it.
- **Speed up document construction.**
  Often the process of retrieving a document from somewhere external (database, filesystem, crawled from a Web site, etc.) is very time consuming.
- **Don't optimize... ever.**

- **Use multiple threads with one IndexWriter.**
  Modern hardware is highly concurrent (multi-core CPUs, multi-channel memory architectures, native command queuing in hard drives, etc.) so using more than one thread to add documents can give good gains overall. Even on older machines there is often still concurrency to be gained between IO and CPU. Test the number of threads to find the best performance point.
- **Index into separate indices then merge.**
  If you have a very large amount of content to index then you can break your content into N "silos", index each silo on a separate machine, then use the writer.addIndexesNoOptimize to merge them all into one final index.
- **Run a Java profiler.**
  If all else fails, profile your application to figure out where the time is going. I've had success with a very simple profiler called JMP. There are many others. Often you will be pleasantly surprised to find some silly, unexpected method is taking far too much time.