ImproveSearchingSpeed

How to make searching faster

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

- **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 searching 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 searching. If the index must be remote, try to mount the remote filesystem as a "readonly" mount. In some cases this could improve performance.
- **Get faster hardware, especially a faster IO system.**
  Flash-based Solid State Drives works very well for Lucene searches. As seek-times for SSD's are about 100 times faster than traditional platter-based harddrives, the usual penalty for seeking is virtually eliminated. This means that SSD-equipped machines need less RAM for file caching and that searchers require less warm-up time before they respond quickly.
- **Tune the OS**
  One tunable that stands out on Linux is swappiness (http://kerneltrap.org/node/3000), which controls how aggressively the OS will swap out RAM used by processes in favor of the IO Cache. Most Linux distros default this to a highish number (meaning, aggressive) but this can easily cause horrible search latency, especially if you are searching a large index with a low query rate. Experiment by turning swappiness down or off entirely (by setting it to 0). Windows also has a checkbox, under My Computer -> Properties -> Advanced -> Performance Settings -> Advanced -> Memory Usage, that lets you favor Programs or System Cache, that's likely doing something similar.
- **Open the IndexReader with readOnly=true.**
  This makes a big difference when multiple threads are sharing the same reader, as it removes certain sources of thread contention.
- **On non-Windows platform, using NIOFSDirectory instead of FSDirectory.**
  This also removes sources of contention when accessing the underlying files. Unfortunately, due to a longstanding bug on Windows in Sun's JRE (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6265734 – feel particularly free to go vote for it), NIOFSDirectory gets poor performance on Windows.
- **Add RAM to your hardware and/or increase the heap size for the JVM.**
  For a large index, searching can use alot of RAM. If you don't have enough RAM or your JVM is not running with a large enough HEAP size then the JVM can hit swapping and thrashing at which point everything will run slowly.
- **Use one instance of IndexSearcher.**
  Share a single IndexSearcher across queries and across threads in your application.
- **When measuring performance, disregard the first query.**
  The first query to a searcher pays the price of initializing caches (especially when sorting by fields) and thus will skew your results (assuming you re-use the searcher for many queries). On the other hand, if you re-run the same query again and again, results won't be realistic either, because the operating system will use its cache to speed up IO operations. On Linux (kernel 2.6.16 and later) you can clean the disk cache using `sync ; echo 3 > /proc/sys/vm/drop_caches`. See http://linux-mm.org/Drop_Caches for details.
- **Re-open the IndexSearcher only when necessary.**
  You must re-open the IndexSearcher in order to make newly committed changes visible to searching. However, re-opening the searcher has a certain overhead (noticeable mostly with large indexes and with sorting turned on) and should thus be minimized. Consider using a so called warming technique which allows the searcher to warm up its caches before the first query hits.
- **Decrease mergeFactor.**
  Smaller mergeFactors mean fewer segments and searching will be faster. However, this will slow down indexing speed, so you should test values to strike an appropriate balance for your application.
- **Limit usage of stored fields and term vectors.**
  Retrieving these from the index is quite costly. Typically you should only retrieve these for the current "page" the user will see, not for all documents in the full result set. For each document retrieved, Lucene must seek to a different location in various files. Try sorting the documents you need to retrieve by docID order first.
- **Use FieldSelector to carefully pick which fields are loaded, and how they are loaded, when you retrieve a document.**
- **Don't iterate over more hits than needed.**
  Iterating over all hits is slow for two reasons. Firstly, the search() method that returns a Hits object re-executes the search internally when you need more than 100 hits. Solution: use the search method that takes a HitCollector instead. Secondly, the hits will probably be spread over the disk so accessing them all requires much I/O activity. This cannot easily be avoided unless the index is small enough to be loaded into RAM. If you don't need the complete documents but only one (small) field you could also use the FieldCache class to cache that one field and have fast access to it.
- **When using fuzzy queries use a minimum prefix length.**
  Fuzzy queries perform CPU-intensive string comparisons - avoid comparing all unique terms with the user input by only examining terms starting with the first "N" characters. This prefix length is a property on both QueryParser and FuzzyQuery - default is zero so ALL terms are compared.
- **Consider using filters.**
  It can be much more efficient to restrict results to a part of the index using a cached bit set filter rather than using a query clause. This is especially true for restrictions that match a great number of documents of a large index. Filters are typically used to restrict the results to a category but could in many cases be used to replace any query clause. One difference between using a Query and a Filter is that the Query has an impact on the score while a Filter does not.
- **Find the bottleneck.**
  Complex query analysis or heavy post-processing of results are examples of hidden bottlenecks for searches. Profiling with at tool such as Visual VM helps locating the problem.