SolrPerformanceFactors

See also: SolrPerformanceData, BenchmarkingSolr, SolrPerformanceProblems

See also: Lucene's ImproveSearchingSpeed and ImproveIndexingSpeed pages.

- Schema Design Considerations
  - indexed fields
  - Stored fields
- Configuration Considerations
  - mergeFactor
    - mergeFactor Tradeoffs
  - HashDocSet Max Size Considerations
  - Cache autoWarm Count Considerations
  - Cache hit rate
  - Explicit Warming of Sort Fields
- Optimization Considerations
- Updates and Commit Frequency Tradeoffs
- Query Response Compression
- Indexing Performance
- RAM Usage Considerations
  - OutOfMemoryErrors
  - Memory allocated to the Java VM
  - Factors affecting memory usage

Schema Design Considerations

indexed fields

The number of indexed fields greatly increases the following:

- Memory usage during indexing
- Segment merge time
- Optimization times
- Index size

These effects can be reduced by the use of `omitNorms="true"`

Stored fields

Retrieving the stored fields of a query result can be a significant expense. This cost is affected largely by the number of bytes stored per document--the higher byte count, the sparser the documents will be distributed on disk and more I/O is necessary to retrieve the fields (usually this is a concern when storing large fields, like the entire contents of a document).

Consider storing large fields outside of Solr. If you feel inclined to do so regardless, consider using compressed fields, which increase the CPU cost of storing and retrieving the fields, but lowers the I/O burden and CPU usage.

If you aren't always using all the stored fields, then enabling lazy field loading can be a huge boon, especially if compressed fields are used.

Configuration Considerations

mergeFactor

The mergeFactor roughly determines the number of segments.

The mergeFactor value tells Lucene how many segments of equal size to build before merging them into a single segment. It can be thought of as the base of a number system.

For example, if you set mergeFactor to 10, a new segment will be created on the disk for every 1000 (or maxBufferedDocs) documents added to the index. When the 10th segment of size 1000 is added, all 10 will be merged into a single segment of size 10,000. When 10 such segments of size 10,000 have been added, they will be merged into a single segment containing 100,000 documents, and so on. Therefore, at any time, there will be no more than 9 segments in each index size.

These values are set in the *mainIndex* section of solrconfig.xml (disregard the indexDefaults section):

mergeFactor Tradeoffs

High value merge factor (e.g., 25):

- Pro: Generally improves indexing speed
- Con: Less frequent merges, resulting in a collection with more index files which may slow searching

Low value merge factor (e.g., 2):

- Pro: Smaller number of index files, which speeds up searching.
- Con: More segment merges slow down indexing.

HashDocSet Max Size Considerations

The hashDocSet is an optimization specified in the solrconfig.xml that enables an int hash representation for filters (docSets) when the number of items in the set is less than maxSize. For smaller sets, this representation is more memory efficient, more efficient to iterate, and faster to take intersections.

The hashDocSet max size should be based primarliy on the number of documents in the collection – the larger the number of documents, the larger the hashDocSet max size. You will have to do a bit of trial-and-error to arrive at the optimal number:

1. Calulate 0.005 of the total number of documents that you are going to store.
2. Try values on either 'side' of that value to arrive at the best query times.
3. When query times seem to plateau, and performance doesn't show much difference between the higher number and the lower, use the higher.

Note: hashDocSet is no longer part of Solr as of version 1.4.0, see SOLR-1169.

Cache autoWarm Count Considerations

When a new searcher is opened, its caches may be prepopulated or "autowarmed" with cached object from caches in the old searcher. `autowarmCount` is the number of cached items that will be copied into the new searcher. You will proably want to base the autowarmCount setting on how long it takes to autowarm. You must consider the trade-off – time-to-autowarm versus how warm (i.e., autowarmCount) you want the cache to be. The autowarm parameter is set for the caches in solrconfig.xml.

See also the Solr Caching page.

Cache hit rate

Monitor the cache statistics from Solr's admin! Raising Solr's cache size is often the best way to improve performance, especially if you notice many evictions for a particular cache type. Pay particular attention to the `filterCache`, which is also used internally by Solr for facetting. See also SolrCaching and this FAQ entry.

Explicit Warming of Sort Fields

If you do a lot of field based sorting, it is advantageous to add explicitly warming queries to the "newSearcher" and "firstSearcher" event listeners in your solrconfig which sort on those fields, so the FieldCache is populated prior to any queries being executed by your users.

Optimization Considerations

You may want to optimize an index in certain situations – ie: if you build your index once, and then never modify it.

If you have a rapidly changing index, rather than optimizing, you likely simply want to use a lower merge factor. Optimizing is very expensive, and if the index is constantly changing, the slight performance boost will not last long. The tradeoff is not often worth it for a non static index.

In a master slave setup, sometimes you may also want to optimize on the master so that slaves serve from a single segment index. This will can greatly increase the time to replicate the index though, so this is often not desirable either.

Updates and Commit Frequency Tradeoffs

If slaves receive new collections too frequently their performance will suffer. In order to avoid this type of degradation you must understand how a slave receives a collection update so that you can know how to best adjust the relevant parameters (number/frequency of commits, snappullers, and autowarming/autocount) so that new collections do not get installed on slaves too frequently.

1. A snapshot of the collection is taken every time a client runs a commit, or an optimization is run depending on whether `postCommit` or `postOptimize` hooks are used on the master.
2. Snappullers on the slaves running on a cron'd basis check the master for new snapshots. If the snappullers find a new collection version the slaves pull it down and snapinstall it.
3. Every time a new index searcher is opened, some autowarming of the cache occurs before Solr hands queries over to that version of the collection. It is crucial to individual query latency that queries have warmed caches.

The three relevant parameters:

- The **number/frequency of snapshots** is completely up to the indexing client. Therefore, the number of versions of the collection is determined by the client's activity.
- The **snappullers** are cron'd. They could run every second, once a day, or anything in between. When they run, they will retrieve only the most recent collection that they do not have.
- **Cache autowarming** is configured for each cache in solrconfig.xml.

If you desire frequent new collections in order for your most recent changes to appear "live online", you must have both frequent commits/snapshots and frequent snappulls. The most frequently you can distribute index changes and maintain good performance is probably in the range of 1 to 5 minutes, depending on your reliance on caching for good query times, and the time it takes to autowarm those caches.

Cache autowarming may be crucial to performance. On one hand a new cache version must be populated with enough entries so that subsequent queries will be served from the cache after the system switches to the new version of the collection. On the other hand, autowarming (populating) a new collection could take a lot of time, especially since it uses only one thread and one CPU. If your settings fire off snapinstaller too frequently, then a Solr slave could be in the undesirable condition of handing-off queries to one (old) collection, and, while warming a new collection, a second “new” one could be snapped and begin warming!

If we attempted to solve such a situation, we would have to invalidate the first “new” collection in order to use the second one, then when a “third” new collection would be snapped and warmed, we would have to invalidate the “second” new collection, and so on ad infinitum. A completely warmed collection would never make it to full term before it was aborted. This can be prevented with a properly tuned configuration so new collections do not get installed too rapidly.

Query Response Compression

Compressing the Solr XML response before it is sent back to the client is worthwhile in some circumstances. If responses are very large, and NIC I/O limits are encroached, *and* Gigabit ethernet is not an option, using compression is a way out.

Compression increases CPU use and since Solr is typically a CPU-bound service, compression *diminishes* query performance. Compression attempts to reduce files to 1/6th original size, and network packets to 1/3rd original size. (We're not taking the time right now to figure out if the big gap between bits and packets makes sense or not, but suffice it to say it's a nice reduction.) Query performance is impacted ~15% on the Solr server.

Consult the documentation for the application server you are using (ie: Tomcat, Resin, Jetty, etc...) for more information on how to configure page compression.

Indexing Performance

In general, adding many documents per update request is faster than one per update request.

For bulk updating from a Java client, (in 3.X) consider using the StreamingUpdateSolrServer.java which streams updates over multiple connections using multiple threads. **N.B.** In 4.X, the StreamingSolrServer has been deprecated in favour of the ConcurrentUpdateSolrServer, and for SolrCloud use CloudSolr Server.

Reducing the frequency of automatic commits or disabling them entirely may speed indexing. Beware that this can lead to increased memory usage, which can cause performance issues of its own, such as excessive swapping or garbage collection.

RAM Usage Considerations

OutOfMemoryErrors

If your Solr instance doesn't have enough memory allocated to it, the Java virtual machine will sometimes throw a Java OutOfMemoryError. There is no danger of data corruption when this occurs, and Solr will attempt to recover gracefully. Any adds/deletes/commits in progress when the error was thrown are not likely to succeed, however. Other adverse effects may arise. For instance, if the SimpleFSLock locking mechanism is in use (as is the case in Solr 1.2), an ill-timed OutOfMemoryError can potentially cause Solr to lose its lock on the index. If this happens, further attempts to modify the index will result in

```
SEVERE: Exception during commit/optimize:java.io.IOException: Lock obtain timed out: SimpleFSLock@/tmp/lucene-5d12dd782520964674beb001c4877b36-write.lock
```

errors.

If you want to see the heap when OOM occurs set "-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/the/dump" (courtesy of Bill Au)

Memory allocated to the Java VM

The easiest way to fight this error, assuming the Java virtual machine isn't already using all your physical memory, is to increase the amount of memory allocated to the Java virtual machine running Solr. To do this for the example/ in the Solr distribution, if you're running the standard Sun virtual machine, you can use the -Xms and -Xmx command-line parameters:

```
java -Xms512M -Xmx1024M -jar start.jar
```

To see how big different JVM memory pools are and how much they are utilized use tools like jconsole, visualvm, or SPM for Solr performance monitoring for long-term monitoring.

Factors affecting memory usage

You may also wish to try to actually reduce Solr's memory usage.

One factor is the size of the input document:

When processing an "add" command for a document, the standard XML update handler has two limitations:

- All of the document's fields must simultaneously fit into memory. (Technically, it's actually the sum of min(<the actual field value's length>, maxFieldLength). As such, adjusting maxFieldLength may be of some help.)
    - (*I'm assuming that fields are truncated to maxFieldLength before being added to the relevant document object. If that's not true, then maxFieldLength won't help here. --ChrisHarris*)
- Each individual <field>...</field> tag in the input XML must fit into memory, regardless of maxFieldLength.

Note that several different "add" commands can be running simultaneously (in different threads). The more threads, the greater the memory usage.

When indexing, memory usage will grow with the number of documents indexed until a commit is performed. A commit (including a soft commit) will free up almost all heap memory. To avoid very large heaps and associated garbage collection pauses during indexing, perform a manual (soft) commit periodically, or consider enabling `autoCommit` (or `autoSoftCommit`) in solrconfig.xml.