FieldSelectorPerformance

Lazy loading when there are many stored (but not indexed) fields in documents can significantly improve performance. In this application, we need to return summary information for up to 500 documents in response to searches. Only when the user drills down into a specific document do we need to get very much of the possibly compressed data.

See then end of this e-mail for design notes

DISCLAIMER: Your results may vary. Once I figured out the speed-up I got by using FieldSelector, I stopped looking for further improvements or refining my test harness since we're now getting better than 3 times the design performance target. So, while I'm quite confident I'm seeing a *very* significant improvement, these numbers aren't all that precise.

I wrote a little test harness that creates a configurable number of threads firing queries off at my search engine with no delays, first firing off a warm-up query before starting any of the threads. It's a fairly simple measurement, but the results are pretty consistent and way better than "one-one thousand, two - one thousand"...

This particular application returns lots summaries at a time as a result of a search, the default is 500. This is summary information, so I only return 6 fields from each document. I'm using a TopDocs to assemble results.

Baseline QPS for returning 1,000 results 0.9 or so queries per second (QPS), before any tuning. This is not acceptable in our application.....

So I started by asking:
*What happens if I retrieve only one doc?
*What happens if I retrieve 100 docs?
*What happens if I retrieve 1000 docs?

All the above require the same search effort, including sorting, so the fact that my results were as follows lead me to scratch my head since I expected the time to be spent in searching and sorting. Note that these numbers are with default (relevance) sorting. Sorting on other fields costs about 0.2 QPS, so I'll ignore them.
*returning 1 doc, 33 qps
*returning 100 docs, 4.34 qps
*returning 1000 docs, 0.88 qps (ZERO.88. Less than 1)

Hmmmm, sez I. This is surprising. So I commented out the document fetch and kludged in hard-coded responses for the data I would have gotten from the loaded document and got 11 QPS. So then I uncommented the document fetch (without FieldSelector) but still used fake field data and was back to 0.89 QPS. Men have been hung on flimsier evidence.

So, I poked around and found FieldSelector, which has been mentioned several times on the mailing list, but I hadn't found reason to use it yet. It took about 1/2 hour to implement and run my first test. Then I spent another hour realizing that I had foolishly excluded a couple of compressed un-indexed fields that could be loaded. If a field can be loaded the usual way, it can be loaded with a FieldSelector. Sheeeesh...

Anyway, here's the results of using FieldSelector to load only the fields I need.
*returning 1,000 docs 12.5 QPS excluding the 2 compressed fields. (just skipping them)
*returning 1,000 docs 7.14 QPS including loading the compressed fields

So, I regenerated the index without compressing those two fields, and the result is
*returning 1,000 docs, all necessary fields, none compressed: 9 QPS

The regenerated index has two fields (one an integer and one the title of the book) that were stored compressed and not indexed in the 7.14 QPS case, and stored and indexed UN_TOKENIZED in the 9 QPS case. No, don't ask me what I was thinking when I compressed a 4 digit field. I plead advancing senility.

And the little moral here, one I return to repeatedly. The preliminary test took me maybe 3 hours to write and get the first set of anomalous results, which pointed me in a completely different direction than I expected. There's no substitute for data when performance tuning.

Design notes:

I strongly suspect that the meta-data heavy design of this index is the main reason for the differences I'm finding when I use IndexReader.document(doc, FieldSelector) rather than IndexReader.document(doc). I doubt (but have no evidence) that an index with no meta-data would get this kind of performance improvement.

My particular application indexes 20,000+ books, some of them quite large (i.e. over 7,000 pages). The index approaches 4G. I designed it to avoid needing a database, so I store a LOT of data I don't search. Some of it is compressed and the meta-data is not indexed. The point is that in this particular application there may be as much data stored as indexed for each book. And extracting it, particularly the compressed fields (which may be quite large) turns out to be expensive. I haven't calculated an exact ratio of stored to indexed data. And, far and away the largest amount of meta-data (I'm guessing 90%) is irrelevant to the search results I'm concentrating on here. So avoiding the overhead of loading the unneeded meta-data is where the savings is coming from I believe.

The underpinnings of this design is that I need to search lots of page text, but only when displaying a specific book do I care about things like how many pages are in each chapter, the start and end page of each chapter, the size of the image corresponding to each page, etc. I never have to search the meta-data so I store it but don't index it. This allows me to avoid connecting to a database, simplifying the application considerably.