ParallelIncrementalIndexing

Parallel-incremental indexing

This is a proposal for a new major feature in Lucene.

⚠️ :TODO: Discuss implementation details. ⚠️

Problem

Today in Lucene it is only possible to update an entire document by doing a delete/add operation. An often upcoming usecase (e.g. one of the most popular feature requests in Solr) is it to update only certain fields in a large number of documents. It might also often be desired to add/remove terms to /from a given index.

This can currently only be achived by re-indexing the entire documents, but of course this is very expensive. And the work of reprocessing the entire documents is often unnecessary if only one field needs to be changed.

One partial solution in Lucene is the ParallelReader. It is a IndexReader implementation that, as the name says, can read from multiple parallel indexes. What are parallel indexes? In Lucene's terminology multiple indexes are considered parallel when their low-level docIDs match, i.e. when all indexes contain the same documents which have the same docIDs assigned.

However, it is currently not easy to create parallel indexes with Lucene. The reason is that docIDs are assigned internally by Lucene's IndexWriter (DocumentsWriter) and are mutable. When documents are deleted from an index, then their docIDs will be removed in the next segment merge. That in turn leads to remapping the docIDs of any subsequent documents.

Therefore the requirement is that the parallel indexes have to perform the exact same flush/delete/merge operations. This is easily possible if the IndexWriters are used with settings that enforce flushes and merges based on document counts. This was Lucene's default behavior until release 2.2. A big improvement in Lucene was it to switch to flushing and merging based on document sizes, which increases performance and memory consumption significantly.

With flushing/merging by size it is practically impossible to create parallel indexes and keep them in sync on a docID level. This is the problem this new proposed features is targeted to solve.

Parallel-incremental indexing

The idea here is to "link" multiple IndexWriters in a way, so that one
`IndexWriter acts as the only master writer and the others as the slave writers. Only the master decides when to flush and merge. The slave writers use a SlaveMergePolicy and SlaveMergeScheduler that does not decide on its own when it is time to flush or merge. Instead it listens to the MasterMergePolicy and MasterMergeScheduler when and which segments to merge. The MasterMergePolicy can wrap any other MergePolicy, so a free choice of the actual merge strategy is still possible.

This approach enforces all parallel indexes to always perform the same flush and merge operations. Now the only missing piece is to make sure that document deletions also happen on all indexes. If the different parallel indexes contain different fields, then it is not possible to perform the same delete-by-term and delete-by-query operations on all indexes, unless they all share the terms deletions will be performed on. This would be a severe limitation of this design.

A solution for the deletion problem is to have the master and slave writers share the same .del files. Then deletions performed on any index will take immediate effect in all parallel indexes. An obious implementation for this approach is consequently a master/slave directory implementation. The master IndexWriter writes into a master directory, which on disk contains all files a Lucene index contains today. The slave directory "knows" in addition to its own the master directory. If a consumer of the slave directory wants to read or write a file, then it checks if that file is a master file or not. If it is, then it reads /writes from/to the master directory, otherwise from/to the slave directory. We would define the segments file and the deletion files as master files. Then only the master directory would actually have the segments files and delete files.

This has another advantage: the transaction semantics the IndexWriter currently enforces would still work the same way: before a new segment is committed, the master MergeScheduler will trigger the same flush or merge operation on the slave writers. After that is done, the master writer commits the segment. If something goes wrong, then the master writer will rollback and not write a new segments file. We will never run into the problem of having non-matching segments files in the different parallel indexes, they only exist in a single directory!

For merging we will need something like a 2-dimensional MergePolicy. One dimension is the one we have today: merges in the document direction. The second dimension will allow merging in field/term direction, i.e. multiple parallel indexes into a single one.

Usecases:

- Adding a new term or field to an existing index

We would provide a class, e.g. ParallelIndexBuilder, that can create a new index which then has the exact same segments structure as a given one. So to add a new field to an index, we would create a new parallel index and store the data of the new field in it. When that's done you can open a ParallelReader on the existing and new indexes; the app using the new reader will see an index containing the old and new fields - no reindexing of already indexes data was necessary.

Note that using a ParallelReader adds almost no overhead to using a single IndexReader on a single index containing all fields. To open a posting list only one additional hashmap-lookup is necessary to determine in which index to look for the requested term.

- Changing contents of a field

When you design your index layout you need to decide upfront which fields you'll want to update individually. You would put them into a separate parallel index. Then when you need to reindex this field you can simply create a new generation of this parallel index and fill it with the new values. When that's done you create a new ParallelReader using the new index generation and you can delete the old one.