KinoSearchMergeModel

Synopsis

The Kino{{`Search merge model is an algorithm for merging inverted documents, originally developed by Marvin Humphrey for an early version of the search engine library Kino}}`Search. It differs from the Java Lucene 2.0 merge model in two main ways.

- Create segments many documents at a time rather than one at a time
- Merge serialized strings rather than objects

The algorithm is particularly well-suited for use with interpreted-language implementations of Lucene, so Lucy will use it.

Algorithm

```
initialize a segment
create a sort pool (an object which performs
  external sorting)

for each document {
  invert document
  write stored fields and term vectors
  add serialized postings to sort pool
}

for each merge-worthy segment {
  write stored fields and term vectors to current
    segment, skipping deletions
  add serialized postings to sort pool, mapping to
    new doc numbers
}

sort the sort pool
write the term dictionary, iterating through the
  items in the sort pool

commit changes
```

Serialized Posting Format

1. Field name encoded as UTF-8 2. null byte 3. Term text as UTF-8 4. null byte 5. document number encoded as big-endian 32-bit integer 6. auxiliary data (not used for sorting): positions, start offsets,
   end offsets, field number, term text length

External Sorter

KinoSearch's external sorter implements an N-way merge sort. The C function memcmp is used to determine sort order.

The external sorter, in the form of a "sortex" object, feeds serialized postings into a memory cache. The memory consumed by the cache is tracked, and whenever it exceeds a user-settable threshold, the cache is sorted and written to disk, producing a "run".

When the sorter is told to "sort" its entire sort pool, it sorts the cache and flushes to disk one last time, then enters "fetch" mode. A small read buffer is created for each run, and items from those buffers are merged into a single, sorted stream.

Rationale

The KinoSearch merge model offers several advantages over the Lucene 2.0 merge model:

1. Adherence to the principle of MinimizingObjectOverhead
   In Lucene 2.0, each document fed into an Index{{`Writer gets turned into its own miniature inverted index, a process that requires that each document get its own Document}}`Writer, TermInfosWriter, FieldsWriter, Field`Infos, and so on. In Java (or C, C++, etc), these objects are cheap. In an interpreted language such as Perl, they are costly by comparison, and the overhead of so much object creation and destruction has a very significant impact on execution speed.

2. Stored fields and TermVectors are only written once, minimizing disk i/o
In Lucene 2.0, segments may be merged multiple times during an indexing session, and the stored fields and term vectors must move each time. Under the KinoSearch merge model, stored fields and term vectors for documents added and segments merged away during the current session are written only once.

3. Lower RAM requirements
The indexing execution speed of Lucene 2.0 depends heavily on how many documents can be buffered in RAM. Higher is better, because it increases the number of segment merges which happen entirely in memory, rather than off disk. In contrast, the impact of using a sort cache larger than around 16 MB with the KinoSearch merge model does not seem to be that great.

4. RAM ceiling set in bytes rather than documents
RAM consumption by the indexer is largely a factor of how large the sort pool's cache is set, rather than maxBufferedDocs.

5. Single commit per indexing session
Since only one segment is written per session, there is only one commit. This makes it easier to recover from a crashed indexing pass, because it's clear which documents were successfully added: either all, or none.