FlexibleIndexing

The goal of this page is to design/describe a more flexible indexing scheme for Lucene, as first described in #11 of Lucene2Whiteboard.

Since this is a significant change to Lucene and will require some planning, this page is intended to be the starting point of the design until some patches can be worked out.

There are currently two main areas of interest in regards to flexible indexing:

1. API for flexible Posting (indexing) options. Preliminary options, as detailed in ConversationsBetweenDougMarvinAndGrant, are:

1. <doc>+

b. <doc, boost>+

c. <doc, freq, <position>+ >+

d. <doc, freq, <position, boost>+ >+

These suggest the following booleans per field:

1. freq

2. document boost

3. position (requires freq)

4. position boost (requires position)

2. Storing Index level metadata. This can be useful for storing information about the index, such as display name, internal name, name of analyzer used (if appropriate), collection statistics outside the scope of the index itself.

Planning

Related Information

ConversationsBetweenDougMarvinAndGrant

Flexible indexing implemented in 4.0-dev (trunk)

This section describes the changes committed to 4.0-dev under LUCENE-1458 and LUCENE-2111.

The overall goal is to make Lucene extensible, even at its lowest levels, on what it records into the index, and how. Your app should be able to easily store new things into the index, or, alter how existing things (doc IDs, positions, payloads, etc.) are encoded. To accomplish this, a `Codec` class was introduced. The `Codec` currently covers the postings API (fields, terms, docs, positions+payloads enumerators); other elements in the index (norms, deleted docs, stored docs/fields, term vectors) are not covered.

Changes in how postings are consumed

The first big change in flexible indexing is the consumption of the postings enumerators APIs:

- A term is now an arbitrary `byte[]`, represented by a `BytesRef` (which references an offset + length "slice" into an existing `byte[]`). By default terms will be UTF8 encoded character string, created during indexing, but your analysis chain can produce a term that is not UTF8 bytes.
- Fields are separately enumerated (via `FieldsEnum`) from term text. Consumers of the flex API no longer need to check `Term.field()` on each `.next()` call; instead, they obtain a `TermsEnum` for the specific field they need and iterate it until exhaustion.
- `TermsEnum` iterates and seeks to all terms (returned as `BytesRef`) in the index. A `TermsEnum` is optionally able to seek to the ordinal (long) for the term, and return the ordinal for the current term. `SegmentReader` implements this but `MultiReader` does not because working with ords is far too costly (requires merging).
- Deleted documents are no longer implicitly filtered by `DocsEnum` (previously `TermDocs`). Instead, you provide an arbitrary `skipDocs` bit set (`Bits`) stating which documents should be skipped during enumeration. For example, this could be used with a cached filter to enforce your own deletions. `IndexReader.getDeletedDocs` returns a Bits for the current deleted docs of this reader.
- Seeking to a term is no longer done by the docs/positions enums; instead, you must use `TermsEnum.seek` and then `TermsEnum.docs` or `.docsAndPositions` to obtain the enumerator (there are also sugar APIs to accomplish this). `TermsEnum`'s seek method has three return values: `FOUND` (the exact term matched), `NOT_FOUND` (another term matched) and `END` (you seek'd past the end of the enum).
- Composite readers (currently `MultiReader` or `DirectoryReader`) are not able to provide these postings enumerators directly; instead, one must use the static methods on `MultiFields` to obtain the enumerators.

Codec/CodecProvider

The second big change in flexible indexing is the `Codec` and `CodecProvider` APIs that enables apps to plug in different implementations for writing and reading postings data in the index. When you obtain an `IndexWriter` or `IndexReader`, you can optionally pass in a `CodecProvider`, which knows 1) which `Codec` should be used when writing a new segment, and 2) how to resolve the codec name (`String`) to a `Codec` instance, when reading from the index.

The default codec is `StandardCodec`, whose format is similar to the pre-4.0 index format, but introduces sizable improvements to how the terms index is stored. In particular the RAM required by the terms index when reading a segment has been substantially reduced. The on-disk format of the .tis/.tii files is also slightly smaller.

There are some experimental core codecs:

- `PulsingCodec` stores rare terms directly into the terms dicts. This is an excellent match for primary key fields (see here for details), and should also help even "normal" fields (so this may become the default codec at some point).
- `SepCodec` and `IntBlockCodec` are available for block-based codecs. These codes are not useful themselves, rather, they serve as the base for block-based codecs. These codecs separately store the docs, freqs, positions and payloads data, allowing for int block codecs to encode the docs, freqs and positions.

LUCENE-1410 has a prototype PforDelta codec, an int block codec using PFOR-DELTA encoding.

Apps can also create custom {{Codec}}s. Please report back if you do! All of these APIs are very new and need some good baking in time.