SolrRelevancyCookbook

Solr Relevancy Cookbook

Relevancy and Case Matching

Lucene does not currently have low level support for inexact case matches. Most people use an analyzer that lowercases all terms in order to match terms regardless of case. Two approaches may be taken to match terms regardless of case, while giving higher scores for exact case:

LowercaseDuplicateTokenFilter (not yet developed)

Use a TokenFilter that outputs two tokens (one original and one lowercased) for each input token.

For queries, the client would need to expand any search terms containing upper case characters to two terms, one lowercased and one original. The original search term may be given a boost, although it may not be necessary given that a match on both terms will produce a higher score.

```
text:iPod ==> (text:iPod OR text:ipod)
text:NeXT ==> (text:NeXT^10 OR text:next)
```

Index the field multiple times

Index the field twice, using different field types in the schema. One field would employ lowercasing and stemming for maximum matching capabilities, and the other field would employ minimal transformations enabling more "exact match" semantics.
For queries in which "exact case scores higher" is desired, a client may query both fields. The exact case field may be given an extra boost, which may not be necessary given that a match on both fields produces a higher score.

```
text:iPOd ==> (text:iPOd OR text_exact:iPOd)
text:"NeXT-3000A" ==> (text:"NeXT-3000A" OR text_exact:"NeXT-3000A"^3)
```

Note that while this approach requires an additional field, it's more flexible than the LowercaseDuplicateTokenFilter since the field types are completely independent and may be used for optional exact punctuation matching, soundex matching, and synonym matching.

The most flexible way of indexing the same field multiple times is via the copyField mechanism as it allows complete isolation from the indexing client.

Ranking Terms

"Ranking Terms" is a concept other search applications (like AltaVista) use as a way of placing greater importance on certain query terms by specifying an ordered list of important terms to be applied to the results of a query. The ranking terms will only change the sorting order, not the number of documents found by a query. All documents containing a ranking term will be sorted before all documents without it.

Boosting Ranking Terms

The most general way of implementing this is via Lucene's boosting mechanism which allows one to boost the importance of any part of a query relative to the other parts. If absolute ordering is desired, a very high boost may be used.

For example, if you want to find all documents containing both "widescreen" and "HDTV" terms, but you want to give extra importance to those with "HDTV", then simply boost that term query.

```
widescreen AND HDTV^3
```

If you want all documents with HDTV to sort before all documents without it, use a very high boost:

```
widescreen AND HDTV^100
```

A general approach of for building Solr/Lucene queries with ranking terms is to create a boolean query with the main query part marked as mandatory, and the ranking terms marked as optional with high boost.

```
+(basequery) rankingterm1^100
+(basequery) rankingterm1^10000 rankingterm2^100
```

Sorting

If the ranking term is a single valued field, then the Solr sorting mechanism may be used to sort all documents containing that field first (or last if desired).

For example, if the "popular" field is either missing, or set to "true", then one can move all results containing popular:true to the top of the search results with the following query and sort:

```
widescreen AND HDTV^2; popular desc, score desc;
```

Term Proximity

It may be desirable to boost the score of documents with query terms that appear closer together. This is not done by default in Lucene, but there are Lucene Span queries that do this. Unfortunately, these queries are relatively new and don't have any support in the query parser (only a Java API currently exists).

One way to get term proximity effects with the current query parser is to use a phrase query with a very large slop. Phrase queries with slop will score higher when the terms are closer together.

without term proximity	term proximity using phrase queries
foo AND bar	"foo bar"~1000000
foo OR bar	foo OR bar OR "foo bar"~1000000^10
"foo bar" "baz bing"	+("foo bar" "baz bing") "foo baz" ~1000000

Intra-Word Delimiters

It may be desirable to treat certain non alph-anumeric characters as word delimiters and be able to query for the words concatenated as well as split apart. Case changes and alpha-numeric transitions may also be treated as intra-word delimiters.

source document text	desirable matching search-box text
wi-fi	wi-fi, wifi, wi+fi, wi fi
PowerShot	PowerShot, Power-Shot, Power Shot
Canon SD500	Canon SD500, Canon SD 500, CanonSD500, Canon-SD-500, CanonSD 500

Approach 1: Index Expansion: index all combinations of groupings

A limited form of index expansion may be performed with Solr's WordDelimiterFilter.

If all possible groupings of subword elements are indexed, the query side may remain relatively simple. The number of terms indexed using this algorithm will be approximately n(n/2+1/2) where n is the number of sub-word elements.

```
Assume that a,b,c,d are non-number subword elements.
SOURCE: a-b      INDEX: a b, ab
SOURCE: a-b-c    INDEX: a b c, ab c, a bc, abc
SOURCE: a-b-c-d  INDEX: a b c d, ab c d, a bc d, a b cd, abc d, a bcd, abcd

SEARCHBOX: a+b-c d  QUERY: "a+b-c d"  PARSED_QUERY:  "abc d"
SEARCHBOX: a-b cd   QUERY: "a-b cd"   PARSED_QUERY:  "ab cd"
```

Term Positions

Term positions may overlap to allow phrase searches of any combination of subwords to match.

Expanded Term Positions of a 2 segment Compound Word	
1	**2**
a	b
	a b

	Expanded Term Positions of a 3 segment Compound Word	
1	**2**	**3**
a	b	c
	ab	
	bc	abc

	Expanded Term Positions of a 4 segment Compound Word		
1	**2**	**3**	**4**
a	b	c	d
	ab	bc	
	a	abc	
		cd	abcd

Advantages:

- Simple query construction and execution... just remove delimiters
- The source documents are assumed to be of higher quality than the search box text. This allows easier expansion on the indexing or server side than on the client/query side.
    - Example: if the user enters "i hate my think pad", we really don't want to try all word groupings: "ihate my think pad" | "ihatemy think pad" | "ihatemythink pad", etc

Disadvantages:

- If words with many sub-words are prevalent, the index explodes in size
- pollution of index frequency statistics
- scoring doesn't make a distinction of closeness to original word (but a separate field could be used to boost scores of exact matches).
- destruction of original word spacing for that field (exact phrase queries for words after the expanded term won't work).
    - Example: PhraseQuery("wifi card") without any slop won't match an index containing the terms wifi/wi, fi , card (but now "wi fi card" would match).
- increases probability of false matches
- exact phrase queries for some groupings of subwords won't match... slop must be used.

Handling Numbers

Subword elements that are numbers may be handled differently. If concatenation of two numbers is not needed or undesirble, then numbers need not be indexed catenated with any other subword elements. The reason for this is that alpha-numeric transitions will never be obscured, so numbers may always be indexed alone because the same transformation can always be done reliably during query parsing.

```
SOURCE: SD500     INDEX: SD 500
SOURCE: SD500-XL  INDEX: SD 500 XL

SEARCHBOX: SD-500   QUERY: "SD-500"    PARSED_QUERY:  "SD 500"
SEARCHBOX: SD 500XL QUERY: "SD 500XL"  PARSED_QUERY:  "SD 500 XL"
```

Advantages to handling numbers differently:

- makes the index much smaller for product names containing numbers
    - PowerShotSD200DigitalELPH would be expanded to 6*5 or 30 terms if number catenation is allowed, but just 5 terms without.

Disadvantages to handling numbers differently:

- may make matches on serial numbers worse

Variations:

- A concatenated version of all adjacent numbers may be indexed, and that would allow the non-matching "SN 100200300" in the following table to match.

	Cases Handled by Index Expansion w/o Number catenation	
source document text	**matching raw search-box queries**	**nonmatching queries**
wi-fi	wi-fi, wifi, wi+fi, wi fi	
wi fi	wi-fi, wi+fi, wi fi	wifi
PowerShot	PowerShot, Power-Shot, Power Shot	
powershot	PowerShot, Power-Shot	Power Shot
sd500	sd500, sd 500, sd-500	sd-50-0
sd 500	sd500, sd 500, sd-500	sd-50-0
SN100-200-300	SN-100/200/300, SN 100 200 300	SN 100200300

Query and Analysis Debugging

Query Parser Results

If you go to the "FULL INTERFACE" part of the query section of the admin page, and check the "Debug: query info" checkbox when issuing a query to the standard query handler, you can see how the query parser parsed your query (which includes the analysis phase). This sets the query parameter "debugQuery=on", and results in the return of an extra list named "debug" containing the parsed query.

Example: If you type in a query such as "Running Cows", you should see the following fragment at the end of the XML response (details depend on the exact schema):

```
<lst name="debug">
  <str name="querystring">Running Cows</str>
  <str name="parsedquery">text:run text:cow</str>
</lst>
```

Analyzer Tools

⚠️ :TODO: ⚠️ fill this in refrencing analysis.jsp