LexEVS Text Match Algorithm Audit

Document Information

Author: Craig Stancl, Scott Bauer, Cory Endle
Email: Stancl.craig@mayo.edu, bauer.scott@mayo.edu, endle.cory@mayo.edu
Team: LexEVS
Contract: S13-500 MOD4
Client: NCI CBIIT
National Institutes of Heath
US Department of Health and Human Services

Table of Contents

Overview

The extent of text match algorithms in LexEVS has grown quite a lot over the decade the application has been in existence. Many matching algorithms overlap in their functionality and dependencies. We've created a review of each of these algorithms with notes on their index dependencies and search focus with an eye towards simplifying and updating the search functionality. NCI should review and decide if any of these can be removed or updated.

Current Text Matches

- Lucene Query
- phrase
- contains
- leading and trailing wild card
- exact match
- substring
- spelling error tolerant substring match
- stemmed Lucene query
- literal contains
- starts with
- non leading wild card literal substring
- literal
- Weighted double metaphone Lucene query
- literal substring
- Double metaphone Lucene query
- Regular expression

Text Match Breakdown:

Lucene Query

Search with the Lucene query syntax. http://lucene.apache.org/core/5_0_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package_description

Phrase

Searches for a Phrase in text using the regular Lucene query parser. The only addition is a set of escaped quotation marks at the beginning and end of the phrase. It could be done in the regular Lucene Query by the user. No special indexing.

Contains

Equivalent to ' term* ' - in other words - a trailing wildcard on a term (but no leading wild card) and the term can appear at any position. Searches on property value only.

Leading and Trailing Wild Card

Equivalent to '*term*' This should be a very poor performing search and is not recommended especially when entering a phrase.

Exact Match

Exact match (case insensitive). Requires it's own indexed value, a lower case, untokenized property value.

SubString

Search based on a \"*some sub-string here*\". Functions much like the Java String.indexOf method. This requires two indexed fields to manage this without significant overhead. One field is the tokenized property value which causes no extra indexing, the other is reversed which requires an extra indexed field.

Spelling Error Tolerant Substring Match

Adds Spelling-error tolerance to 'subString' search. This makes use of the double metaphone indexed value as well as literal property values. Since it shares these with other algorithms it probably doesn't add much to the index. However this doesn't look like a high performing search.

Stemmed Lucene Query

Search with the Lucene query syntax, using stemmed terms. A search for 'trees' will get a hit on 'tree' This requires an extra indexed field when it is enabled in the load.

Literal Contains

Works the same as contains but uses the literal property value enabling searches on special characters.

Starts With

Equivalent to 'term*' (case insensitive) This runs against the same indexed property value as exactMatch so no extra indexing is needed. The query may require increased overhead however.

Non Leading Wild Card Literal Substring

Search based on a \"*some sub-string here*\" Functions much like the Java String.indexOf method. Single term searches will match '*term' and 'term*' but not '*term*'. This is because leading wildcards are very inefficient. Special characters are included. This seems to be very similar to the literal contains, but makes use of the reverse index.

Literal

All special characters are taken literally. Since we usually normalize we can search on a string with colons, parentheses and other special characters using this search.

Weighted Double Metaphone Lucene Query

Search with the Lucene query syntax, using a 'sounds like' algorithm. A search for 'atack' will get a hit on 'attack' Also, the exact user-entered text is taken into account -- so correct spelling will override the 'sounds like' algorithm. Searches on the same indexed property value as the other double metaphone search. Does not add anything more to the index, but does add more overhead to the search.

Literal Substring

The same as the substring search but with special characters enabled. However this doesn't seem to make use of the optimized reverse string. It may be slower as a result.

Double Metaphone Lucene Query

Search with the Lucene query syntax, using a 'sounds like' algorithm. A search for 'atack' will get a hit on 'attack' Searches on a property value that has been double metaphone enabled.

Regular Expression

A Regular Expression query. Searches against the lowercased text, so a regular expression that specifies an uppercase character will never return a match. Additionally, this searches against the entire string as a single token, rather than the tokenized string - so write your regular expression accordingly. This is the apache implementation of Regular Expression so follow their documentation as needed. This is from the old Jakarta project. We may want to update to the latest Lucene supported version of this. This runs against the same indexed value as exact match so it doesn't cost any more in terms of indexed data.