Optimizing Multilingual Search With Solr

Recall, Precision and Accuracy
INTRODUCTION

Today's search application users expect search engines to just work seamlessly across multiple languages. They should be able to issue queries in any language against a document corpus that spans multiple languages, and get quality results back.

This whitepaper offers the search application engineer concrete, step-by-step advice on how to customize Apache™ Solr for multilingual search applications. After a brief look at international text processing issues and how they affect precision and recall, we will discuss open source and commercial tools that address these issues. Finally, we will discuss the pros and cons of three possible architectures for a multilingual search engine to deliver a scalable solution that maximizes precision and recall.

ISSUES IN MULTILINGUAL SEARCH

The basic data structure behind a search engine is the inverted index: a mapping from words to the documents containing them. Precision and recall of a search application therefore hinge on the quality of the natural language processing (NLP) pipeline that is applied to text at both index and query time. Multilingual search is hard from a linguistics point of view because the algorithms within the NLP pipeline aren’t always straightforward. (Some NLP subprocesses are particularly difficult for certain languages.) This whitepaper focuses more on the execution of multilingual search in Solr than the underlying natural language processing algorithms, but it’s important to survey this space in order to understand why certain approaches fall short.

Multilingual search is hard from a linguistics point of view because the algorithms within the NLP pipeline aren’t always straightforward.

Language Identification

Identifying the language of a document is often a required first step of the NLP pipeline, as subsequent NLP subprocesses depend on the language of the text. Language identification is usually only performed at index time, because the accuracy of language identification algorithms on short text strings at query time tends to be significantly worse than on longer documents being indexed.
Tokenization

Tokenization refers to dividing a string of written language into words. English is relatively easy to tokenize—using whitespace as word boundaries gets you most of the way there, although there are exceptions, e.g., tokenizing “can’t” into its component words “can” and “not”. It’s not always so easy. In Chinese and Japanese, there are no spaces between words, and in Korean, spaces delineate words inconsistently and with variation among writers.

Word Form Normalization

Once a piece of text has been segmented into words, a word form normalization algorithm is applied to each word. Word form normalization refers to an NLP process that maps a word to some canonical form (e.g., “are”, “is”, and “being” all map to “be”). Performing this normalization at both index time and query time, improves the recall of search engines because queries for one form of a word match documents containing any form of the word.

Stemming and lemmatization are the two primary approaches that search engines use to normalize word variations:

- **Stemming**: a simple rules-based approach that removes characters from the end of a word in order to reduce it to its root form.
- **Lemmatization**: the identification of the dictionary form of a word based on its context.

Lemmatization is preferable to a simplistic stemming approach as it improves both precision and recall. For example, a user searching for “President Obama speaking on healthcare” likely expects to find documents containing “President Obama spoke on healthcare.” By lemmatizing “speaking,” “spoke,” and “has spoken” to the lemma “speak,” at both index and query time, the search engine can recall all relevant results. Stemming would fail to derive “speak” from “spoke,” which may yield imprecise search results about bicycle parts. For European languages, lemmatization is particularly important due to the large amount of irregular inflection and word form variation.

Decompounding

Many languages—German, Dutch, Korean, and Scandinavian languages—freely use compound words that must be broken up into their decompound parts in order to increase recall. The German compound word “Fließbandproduktionsleiter”, meaning “assembly line production manager”, is a good example. It’s reasonable to assume that a person looking in a job board for a manager (Leiter) position should be able to query with the single word “Leiter” and have documents containing the compound word returned (and vice versa).
HOW TO MAKE SOLR MULTILINGUAL

So, how does one apply techniques for addressing the complexities of multilingual search within a Solr environment? Unfortunately, there isn’t a single approach for all use cases. In this paper, we’ll review a handful of the options for configuring Solr for multilingual search, and discuss the associated tradeoffs.

Solr Analysis: How Solr Handles Languages

Analysis is a concept central to Solr that describes the process by which tokens are derived from field text, both at index time and query time. It typically consists of a single tokenizer (which breaks field text into lexical units) and filters, which examine the token stream and keep, transform, discard, or create new tokens from the tokens output by the tokenizer. This series of tokenizer and filters is referred to as the analysis chain for a particular field type. A field type often applies the same analysis chain at index and query time, though this isn’t a requirement.

Language-Specific Analysis Chains in Solr

Generally we want each language to have a distinct analysis chain—with specific rules for segmentation, lemmatization/stemming, decompounding, and character normalization, not to mention synonym and stopword filtering.

For example, using Solr’s freely available stemmers, you might define a field type for German text as follows:

```xml
<fieldType name="text_deu" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.SnowballPorterFilterFactory" language="German"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
</fieldType>
```

The SnowballPorterFilterFactory filter applies stemming in order to achieve word form normalization at index and query time. As discussed previously, stemming yields worse precision and recall compared to lemmatization. This is a limitation of using analysis chains comprised of freely available plugins, as there are no lemmatizers available in Solr out of the box.
Language Identification in Solr

If you do not know the languages of documents in your corpus up front, Solr offers two language detectors out of the box, one based on Tika’s language detection implementation, and the other based on LangDetect.

Language detection is performed within an UpdateRequestProcessor so that all document fields can be viewed and new fields can be written. This is particularly useful in the field-per-language approach to multilingual search discussed below.

If you add the following UpdateRequestProcessor to your requestHandler that applies to document updates, Tika’s language identifier will detect the languages based on the value of the “content” field, the language will be put into a “language” field, and the content field will then be mapped to either “content_eng” or “content_spa”.

```
<processor class="org.apache.solr.update.processor.TikaLanguageIdentifierUpdateProcessorFactory">
  <bool name="langid">true</bool>
  <str name="langid.fl">content</str>
  <str name="langid.langField">language</str>
  <str name="langid.whitelist">en,es</str>
  <bool name="langid.map">true</bool>
  <str name="langid.map.lcmap">en:eng es:spa</str>
</processor>
```
With the language identifier configured to map to sub-fields in this way, you must then define the fields and types that will be used for downstream analysis. For example:

```xml
<dynamicField name="*_eng" type="text_eng" indexed="true" stored="false" multiValued="false"/>
<dynamicField name="*_spa" type="text_spa" indexed="true" stored="false" multiValued="false"/>

<fieldType name="text_eng" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.SnowballPorterFilterFactory" language="English"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
</fieldType>

<fieldType name="text_spa" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.SnowballPorterFilterFactory" language="Spanish"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
</fieldType>
```

**TWO BASIC APPROACHES TO MULTILINGUAL SEARCH IN SOLR**

Once we have defined field types per language, we still have some flexibility in terms of how documents are indexed and queried. For this exploration, we assume that your document corpus is multilingual but that any particular document is in a single language. We focus on two approaches:

1. a separate field per language
2. a separate Solr core per language
Fields Per Language

The field per language Solr setup is straightforward to implement. Whether you define your text field types with freely available tokenizers and filters or use commercial plugins like those packaged in Rosette for Solr, the approach is to create a separate field for every language you care about. For example, to configure your Solr schema to index English and Czech book titles and contents (and only store the former), your schema.xml might contain:

```xml
<field name="book_title_ces" type="text_ces" indexed="true" stored="true" multiValued="false"/>
<field name="book_contents_ces" type="text_ces" indexed="true" stored="false" multiValued="false"/>
<field name="book_title_eng" type="text_eng" indexed="true" stored="true" multiValued="false"/>
<field name="book_contents_eng" type="text_eng" indexed="true" stored="false" multiValued="false"/>
```

How would the search application developer then query against this index? It depends on whether individual queries can be issued for one language or multiple languages.

An example application where queries can be issued within one language is a news search website where users supply a language preference – they’ll only query within that language and they only want results in that language. Querying is simply done against a particular language’s field(s), e.g.:

http://localhost:8983/solr/books/select?q=title_eng:dog

Sometimes queries should be issued across multiple languages. For example, in the electronic document discovery use case, recall is paramount, and users might want to query for all documents containing a particular name, regardless of document language. In this case, Boolean logic can be used in the query:


or the DisMax / Extended DisMax query parsers can be used to query across the per-language fields:

http://localhost:8983/solr/collection1/select?q=Bill&defType=edismax&qf=title_eng%20title_spa
Core Per Language
Another way to index multilingual data is to create a separate Solr core per language. You can use the same field names across cores but tie different field types to them in each core. This introduces more backend complexity compared to the previous approach. Why would we do it then? The primary reason is for performance at scale for queries across multiple languages – the subqueries hitting each core can be executed in parallel. An additional benefit is that queries are simpler because all cores can use the same field name(s). There’s no need to construct bloated Boolean or DisMax queries across per-language fields.

In order to implement this method first you must define your cores. Let’s create `collection_eng` and `collection_spa` cores. The only difference between the two cores is the core name in core.properties and the field type that you tie to your field(s). Let’s say we want a single “text” field. In the collection_eng core’s schema.xml, you would tie the corresponding field type to the “text” field:

```xml
<field name="text" type="text" indexed="true" stored="true" multiValued="true"/>
```

The collection_spa core’s schema.xml would contain the same field specification, and the only difference would be the “text” type would contain the analysis chain for Spanish text.

When you start Solr, you can direct documents at either core. Using Solr’s post.jar indexing tool, this is done via:

```
```

Querying within a language is straightforward—simply direct the query to the appropriate core:

```
```

Querying across multiple per-language cores is done using Solr’s distributed search capability:

```
```
Note that the “q” portion of the query is simpler than it was in the previous approach. However, all language cores that you care about must be tacked onto the query in the “shards” parameter. If your queries are always issued over all language cores, an alternative approach that will simplify queries is to define `requestHandler` with a default shards parameter so that queries don’t have to explicitly specify them:

```xml
<requestHandler name="search" class="solr.SearchHandler" default="true">
  <lst name="defaults">
    <str name="shards"><shard 1 URL>, <shard 2 URL>, ...</str>
  </lst>
</requestHandler>
```

**Comparing Approaches**

The following table summarizes the pros and cons of field per language v. core per language.

<table>
<thead>
<tr>
<th></th>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields per language</td>
<td>- Simplest to implement</td>
<td>- Query performance degrades when many languages are searched</td>
</tr>
<tr>
<td></td>
<td>- Compatible with Solr’s language detection URPs</td>
<td></td>
</tr>
<tr>
<td>Core per language</td>
<td>- Improved performance of queries across languages since queries can be parallelized across cores</td>
<td>- Increased administrative overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Documents must be sharded by language, so not easily tied in with language detection URPs compared to field per language approach</td>
</tr>
</tbody>
</table>
AN ALTERNATIVE APPROACH

Single Multilingual Field

An alternative approach that Trey Grainger from CareerBuilder has developed and uses in production is to create a single field capable of processing content in all languages that you care about. This approach combines the desirable qualities of the previous approaches: you get the administrative simplicity of not having to manage multiple cores, while at the same time avoiding the expense of a large dismax query in the case where you want to search across many languages. An added benefit is that this approach makes handling mixed language documents easier than the other approaches, which all require sending data to multiple fields or cores.

The key to this approach is a new meta-field type that defines a mapping from language to concrete field type, e.g. eng → text_eng, spa → text_spa, etc. A custom tokenizer then inspects field content for prepended language code(s) (e.g., “eng|Solr is the popular, blazing fast open source...”), and applies the concrete field type analyzer(s) corresponding to the language code(s) to the non-prepended text. The final step is to merge and deduplicate all tokens.

The primary downside of this approach is that it requires you to develop a nontrivial Solr plugin to perform the steps just described.

OVERCOMING LINGUISTIC LIMITATIONS WITH ROSETTE

Up to this point, we’ve looked at example Solr configurations comprised of freely available tokenizers and filters. These analyzer components are limited in terms of their linguistic capabilities in the following ways:

- For most languages, there are freely available stemmers but not lemmatizers (the Kuromoji lemmatizer for Japanese being the exception). Compared to lemmatization, stemming yields worse precision (“arsenal” and “arsenic” have the same stem) and recall (“spoke” and “speak” do not have the same stem).
- Solr’s tokenizer for CJK (Chinese, Japanese, Korean) does simple bigramming, which can hurt precision (as single character words aren’t indexed) and can bloat the search index.
- The decompounding filter available in Solr is dictionary-based, and thus not fit to use on languages with widespread use of compounding.

Rosette for Solr addresses these limitations by making the linguistic tools developed at Basis Technology available as Solr plugins.
As an example, the German text field type that incorporates FST-based decompounding as well as lemmatization rather than stemming for word form normalization is defined as follows:

```xml
<fieldType name="text_deu" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer
      class="com.basistech.rosette.lucene.BaseLinguisticsTokenizerFactory"
      rootDirectory="<rootDirectory>
      language="deu"
    />
    <filter
      class="com.basistech.rosette.lucene.BaseLinguisticsTokenFilterFactory"
      rootDirectory="<rootDirectory>
      language="deu"
    />
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer
      class="com.basistech.rosette.lucene.BaseLinguisticsTokenizerFactory"
      rootDirectory="<rootDirectory>
      language="deu"
    />
    <filter
      class="com.basistech.rosette.lucene.BaseLinguisticsTokenFilterFactory"
      rootDirectory="<rootDirectory>
      language="deu"
      query="true"
    />
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

There are field types defined for over **40 languages** in Rosette for Solr.
ADVANCES SEARCH FUNCTIONALITY

Beyond language identification, lemmatization, and decompounding, search engines can employ NLP tools in order to enhance the search experience around named entities in documents. Named entities can be problematic for search (and thus provide an opportunity to differentiate an organization’s search offering) because:

- Entities can be mentioned in multiple languages
- Entity names tend to be misspelled more often than non-entity strings
- The same entity can be referred to in many ways. This calls for the linguistic analogue to lemmatization to be applied to entities (e.g., a search for “Abe Lincoln” should match documents containing “Abraham Lincoln”)

Rosette provides additional software tools that can help enhance multilingual search applications:

**ROSETTE Entity Extractor**

If your corpus consists of raw text documents, entity extraction is used to find the named entities so that tools like name matching and name translation can be applied strictly to the named entities in your documents.

**ROSETTE Name Indexer**

Given a name, how can we match it across different variations and languages, and be robust with respect to misspellings and phonetic guesses? Name matching against a list of canonical names enables such functionality.

**ROSETTE Name Translator**

If your corpus is multilingual or contains named entities in non-English surface forms, translating names to a desired target language can be used to automatically expand user queries across names as they are written in multiple languages.

**ROSETTE Entity Resolver**

Does the “George Bush” in your document refer to the 41st or 43rd president of the United States? Entity Resolution resolves entity mentions in your documents to real world entities, enabling faceted search across the things in your documents as well as enrichment of search results with data from an associated entity knowledge base.
ADVANCES SEARCH FUNCTIONALITY

When extending Solr to search across multiple languages, the engineer has to address four major issues:

1. when and how to perform language identification;
2. the need for tokenization of non-space delimited languages such as CJK;
3. how to normalize word forms—via stemming or the superior lemmatization; and
4. how to decompound words.

The engineer must also carefully select the system architecture of the multilingual system. We considered the pros and cons of the two most common approaches. One, creating a separate field per language has the advantage of implementation simplicity, but degrades query performance as the number of supported languages increases. Two, using a separate Solr core for each language improves performance of querying across languages, but adds administrative overhead and complexity as documents must be sharded by language. A third approach used by CareerBuilder requires development of a nontrivial Solr plugin, but avoids the cons of the two basic approaches, and furthermore enables handling mixed language documents.

International text processing components are available via open source software (language identification, n-gram tokenization, stemming) to get multilingual Solr up and running. For applications with higher requirements of precision and recall, Basis Technology’s Rosette provides tokenization, lemmatization, and decompounding in addition to advanced search functionality in many languages as Solr plugins.

To see how Rosette can help your organization achieve fast, accurate, scalable multilingual search, request a free product evaluation of Rosette for Solr at [http://www.basistech.com/solutions/search/solr/](http://www.basistech.com/solutions/search/solr/).

__________________________

1. We call the keys of the index words for the purposes of illustration, but really they can be any string. As an example of index keys that aren’t strictly words, autocomplete functionality is typically implemented in a search index by indexing all n-grams that start at the beginning of a document token (“hello” → “h”, “he”, “hel”, “hell”, “hello”).

2. Recall, also known as sensitivity, is an information retrieval metric calculated as the fraction of relevant results that are returned in a search.