Apache Solr Reference Guide

For Solr 0.0

Table of Contents

Licenses

Introduction to Document Analysis

This section introduces tokenization, analyzers and filters.

Understanding Analyzers, Tokenizers, and Filters

The following sections describe how Solr breaks down and works with textual data. There are three main concepts to understand: analyzers, tokenizers, and filters.

Field analyzers are used both during ingestion, when a document is indexed, and at query time. An analyzer examines the text of fields and generates a token stream. Analyzers may be a single class or they may be composed of a series of tokenizer and filter classes.

Tokenizers break field data into lexical units, or *tokens*.

Filters examine a stream of tokens and keep them, transform or discard them, or create new ones. Tokenizers and filters may be combined to form pipelines, or *chains*, where the output of one is input to the next. Such a sequence of tokenizers and filters is called an *analyzer* and the resulting output of an analyzer is used to match query results or build indices.

Using Analyzers, Tokenizers, and Filters

Although the analysis process is used for both indexing and querying, the same analysis process need not be used for both operations. For indexing, you often want to simplify, or normalize, words. For example, setting all letters to lowercase, eliminating punctuation and accents, mapping words to their stems, and so on. Doing so can increase recall because, for example, "ram", "Ram" and "RAM" would all match a query for "ram". To increase query-time precision, a filter could be employed to narrow the matches by, for example, ignoring all-cap acronyms if you're interested in male sheep, but not Random Access Memory.

The tokens output by the analysis process define the values, or *terms*, of that field and are used either to build an index of those terms when a new document is added, or to identify which documents contain the terms you are querying for.

For More Information

These sections will show you how to configure field analyzers and also serves as a reference for the details of configuring each of the available tokenizer and filter classes. It also serves as a guide so that you can configure your own analysis classes if you have special needs that cannot be met with the included filters or tokenizers.

For Analyzers, see:

- Analyzers: Detailed conceptual information about Solr analyzers.
- Running Your Analyzer: Detailed information about testing and running your Solr analyzer.

For Tokenizers, see:

- About Tokenizers: Detailed conceptual information about Solr tokenizers.

- Tokenizers: Information about configuring tokenizers, and about the tokenizer factory classes included in this distribution of Solr.

For Filters, see:

- About Filters: Detailed conceptual information about Solr filters.
- Filter Descriptions: Information about configuring filters, and about the filter factory classes included in this distribution of Solr.
- CharFilterFactories: Information about filters for pre-processing input characters.

To find out how to use Tokenizers and Filters with various languages, see:

- Language Analysis: Information about tokenizers and filters for character set conversion or for use with specific languages.

Analyzers

An analyzer examines the text of fields and generates a token stream. Analyzers are specified as a child of the `<fieldType>` element in the `schema.xml` configuration file (in the same `conf/` directory as `solrconfig.xml`).

In normal usage, only fields of type `solr.TextField` will specify an analyzer. The simplest way to configure an analyzer is with a single `<analyzer>` element whose class attribute is a fully qualified Java class name. The named class must derive from `org.apache.lucene.analysis.Analyzer`. For example:

```
<fieldType name="nametext" class="solr.TextField">
  <analyzer class="org.apache.lucene.analysis.WhitespaceAnalyzer"/>
</fieldType>
```

In this case a single class, `WhitespaceAnalyzer`, is responsible for analyzing the content of the named text field and emitting the corresponding tokens. For simple cases, such as plain English prose, a single analyzer class like this may be sufficient. But it's often necessary to do more complex analysis of the field content.

Even the most complex analysis requirements can usually be decomposed into a series of discrete, relatively simple processing steps. As you will soon discover, the Solr distribution comes with a large selection of tokenizers and filters that covers most scenarios you are likely to encounter. Setting up an analyzer chain is very straightforward; you specify a simple `<analyzer>` element (no class attribute) with child elements that name factory classes for the tokenizer and filters to use, in the order you want them to run.

For example:

```
<fieldType name="nametext" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory"/>
    <filter class="solr.EnglishPorterFilterFactory"/>
  </analyzer>
</fieldType>
```

Note that classes in the `org.apache.solr.analysis` package may be referred to here with the shorthand `solr.` prefix.

In this case, no Analyzer class was specified on the `<analyzer>` element. Rather, a sequence of more specialized classes are wired together and collectively act as the Analyzer for the field. The text of the field is passed to the first item in the list (`solr.StandardTokenizerFactory`), and the tokens that emerge from the last one (`solr.EnglishPorterFilterFactory`) are the terms that are used for indexing or querying any fields that use the "nametext" `fieldType`.

Field Values versus Indexed Terms

The output of an Analyzer affects the *terms* indexed in a given field (and the terms used when parsing queries against those fields) but it has no impact on the *stored* value for the fields.

For example, an analyzer might split "Brown Cow" into two indexed terms "brown" and "cow", but the stored value will still be a single string, "Brown Cow".

Analysis Phases

Analysis takes place in two contexts. At index time, when a field is being created, the token stream that results from analysis is added to an index and defines the set of terms (including positions, sizes, and so on) for the field. At query time, the values being searched for are analyzed and the terms that result are matched against those that are stored in the field's index.

In many cases, the same analysis should be applied to both phases. This is desirable when you want to query for exact string matches, possibly with case-insensitivity, for example. In other cases, you may want to apply slightly different analysis steps during indexing than those used at query time.

If you provide a simple `<analyzer>` definition for a field type, as in the examples above, then it will be used for both indexing and queries. If you want distinct analyzers for each phase, you may include two `<analyzer>` definitions distinguished with a type attribute. For example:

```xml
<fieldType name="nametext" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"/>
    <filter class="solr.SynonymFilterFactory" synonyms="syns.txt"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

In this theoretical example, at index time the text is tokenized, the tokens are set to lowercase, any that are not listed in `keepwords.txt` are discarded and those that remain are mapped to alternate values as defined by the synonym rules in the file `syns.txt`. This essentially builds an index from a restricted set of possible values and then normalizes them to values that may not even occur in the original text.

At query time, the only normalization that happens is to convert the query terms to lowercase. The filtering and mapping steps that occur at index time are not applied to the query terms. Queries must then, in this example, be very precise, using only the normalized terms that were stored at index time.

About Tokenizers

The job of a tokenizer is to break up a stream of text into tokens, where each token is (usually) a sub-sequence of the characters in the text. An analyzer is aware of the field it is configured for, but a tokenizer is not. Tokenizers read from a character stream (a Reader) and produce a sequence of Token objects (a TokenStream).

Characters in the input stream may be discarded, such as whitespace or other delimiters. They may also be added to or replaced, such as mapping aliases or abbreviations to normalized forms. A token contains various metadata in addition to its text value, such as the location at which the token occurs in the field. Because a tokenizer may produce tokens that diverge from the input text, you should not assume that the text of the token is the same text that occurs in the field, or that its length is the same as the original text. It's also possible for more than one token to have the same position or refer to the same offset in the original text. Keep this in mind if you use token metadata for things like highlighting search results in the field text.

```
<fieldType name="text" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
  </analyzer>
</fieldType>
```

The class named in the tokenizer element is not the actual tokenizer, but rather a class that implements the `org.apache.solr.analysis.TokenizerFactory` interface. This factory class will be called upon to create new tokenizer instances as needed. Objects created by the factory must derive from `org.apache.lucene.analysis.TokenStream`, which indicates that they produce sequences of tokens. If the tokenizer produces tokens that are usable as is, it may be the only component of the analyzer. Otherwise, the tokenizer's output tokens will serve as input to the first filter stage in the pipeline.

A `TypeTokenFilterFactory` is available that creates a `TypeTokenFilter` that filters tokens based on their TypeAttribute, which is set in `factory.getStopTypes`.

For a complete list of the available TokenFilters, see the section Tokenizers.

When To use a CharFilter vs. a TokenFilter

There are several pairs of CharFilters and TokenFilters that have related (ie: `MappingCharFilter` and `ASCIIFoldingFilter`) or nearly identical (ie: `PatternReplaceCharFilterFactory` and `PatternReplaceFilterFactory`) functionality and it may not always be obvious which is the best choice.

The decision about which to use depends largely on which Tokenizer you are using, and whether you need to preprocess the stream of characters.

For example, suppose you have a tokenizer such as `StandardTokenizer` and although you are pretty happy with how it works overall, you want to customize how some specific characters behave. You could modify the rules and re-build your own tokenizer with JFlex, but it might be easier to simply map some of the characters before tokenization with a `CharFilter`.

About Filters

Like tokenizers, filters consume input and produce a stream of tokens. Filters also derive from `org.apache.lucene.analysis.TokenStream`. Unlike tokenizers, a filter's input is another TokenStream. The job of a filter is usually easier than that of a tokenizer since in most cases a filter looks at each token in the stream sequentially and decides whether to pass it along, replace it or discard it.

A filter may also do more complex analysis by looking ahead to consider multiple tokens at once, although this is less common. One hypothetical use for such a filter might be to normalize state names that would be tokenized as two words. For example, the single token "california" would be replaced with "CA", while the token pair "rhode" followed by "island" would become the single token "RI".

Because filters consume one `TokenStream` and produce a new `TokenStream`, they can be chained one after another indefinitely. Each filter in the chain in turn processes the tokens produced by its predecessor. The order in which you specify the filters is therefore significant. Typically, the most general filtering is done first, and later filtering stages are more specialized.

```
<fieldType name="text" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPorterFilterFactory"/>
  </analyzer>
</fieldType>
```

This example starts with Solr's standard tokenizer, which breaks the field's text into tokens. Those tokens then pass through Solr's standard filter, which removes dots from acronyms, and performs a few other common operations. All the tokens are then set to lowercase, which will facilitate case-insensitive matching at query time.

The last filter in the above example is a stemmer filter that uses the Porter stemming algorithm. A stemmer is basically a set of mapping rules that maps the various forms of a word back to the base, or *stem*, word from which they derive. For example, in English the words "hugs", "hugging" and "hugged" are all forms of the stem word "hug". The stemmer will replace all of these terms with "hug", which is what will be indexed. This means that a query for "hug" will match the term "hugged", but not "huge".

Conversely, applying a stemmer to your query terms will allow queries containing non stem terms, like "hugging", to match documents with different variations of the same stem word, such as "hugged". This works because both the indexer and the query will map to the same stem ("hug").

Word stemming is, obviously, very language specific. Solr includes several language-specific stemmers created by the Snowball generator that are based on the Porter stemming algorithm. The generic Snowball Porter Stemmer Filter can be used to configure any of these language stemmers.

Solr also includes a convenience wrapper for the English Snowball stemmer. There are also several purpose-built stemmers for non-English languages. These stemmers are described in Language Analysis.

Tokenizers

You configure the tokenizer for a text field type in `schema.xml` with a `<tokenizer>` element, as a child of `<analyzer>`:

```
<fieldType name="text" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
  </analyzer>
</fieldType>
```

The class attribute names a factory class that will instantiate a tokenizer object when needed. Tokenizer factory classes implement the `org.apache.solr.analysis.TokenizerFactory`. A TokenizerFactory's `create()` method accepts a Reader and returns a TokenStream. When Solr creates the tokenizer it passes a Reader object that provides the content of the text field.

Arguments may be passed to tokenizer factories by setting attributes on the `<tokenizer>` element.

```
<fieldType name="semicolonDelimited" class="solr.TextField">
  <analyzer type="query">
    <tokenizer class="solr.PatternTokenizerFactory" pattern="; "/>
  </analyzer>
</fieldType>
```

Tokenizer Types

The following sections describe the tokenizer factory classes included in this release of Solr.

For more information about Solr's tokenizers, see http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters.

Standard Tokenizer

This tokenizer splits the text field into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded, with the following exceptions:

- Periods (dots) that are not followed by whitespace are kept as part of the token, including Internet domain names.
- The "@" character is among the set of token-splitting punctuation, so email addresses are **not** preserved as single tokens.

Note that words are split at hyphens.

The Standard Tokenizer supports Unicode standard annex UAX#29 word boundaries with the following token types: `<ALPHANUM>`, `<NUM>`, `<SOUTHEAST_ASIAN>`, `<IDEOGRAPHIC>`, and `<HIRAGANA>`.

Factory class: `solr.StandardTokenizerFactory`

Arguments:

`maxTokenLength`: (integer, default 255) Solr ignores tokens that exceed the number of characters specified by `maxTokenLength`.

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
</analyzer>
```

In: "Please, email john.doe@foo.com by 03-09, re: m37-xq."

Out: "Please", "email", "john.doe", "foo.com", "by", "03", "09", "re", "m37", "xq"

Classic Tokenizer

The Classic Tokenizer preserves the same behavior as the Standard Tokenizer of Solr versions 3.1 and previous. It does not use the Unicode standard annex UAX#29 word boundary rules that the Standard Tokenizer uses. This tokenizer splits the text field into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded, with the following exceptions:

- Periods (dots) that are not followed by whitespace are kept as part of the token.
- Words are split at hyphens, unless there is a number in the word, in which case the token is not split and the numbers and hyphen(s) are preserved.
- Recognizes Internet domain names and email addresses and preserves them as a single token.

Factory class: `solr.ClassicTokenizerFactory`

Arguments:

`maxTokenLength`: (integer, default 255) Solr ignores tokens that exceed the number of characters specified by `maxTokenLength`.

Example:

```
<analyzer>
  <tokenizer class="solr.ClassicTokenizerFactory"/>
</analyzer>
```

In: "Please, email john.doe@foo.com by 03-09, re: m37-xq."

Out: "Please", "email", "john.doe@foo.com", "by", "03-09", "re", "m37-xq"

Keyword Tokenizer

This tokenizer treats the entire text field as a single token.

Factory class: `solr.KeywordTokenizerFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.KeywordTokenizerFactory"/>
</analyzer>
```

In: "Please, email john.doe@foo.com by 03-09, re: m37-xq."

Out: "Please, email john.doe@foo.com by 03-09, re: m37-xq."

Letter Tokenizer

This tokenizer creates tokens from strings of contiguous letters, discarding all non-letter characters.

Factory class: `solr.LetterTokenizerFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.LetterTokenizerFactory"/>
</analyzer>
```

In: "I can’t."

Out: "I", "can", "t"

Lower Case Tokenizer

Tokenizes the input stream by delimiting at non-letters and then converting all letters to lowercase. Whitespace and non-letters are discarded.

Factory class: `solr.LowerCaseTokenizerFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.LowerCaseTokenizerFactory"/>
</analyzer>
```

In: "I just **LOVE** my iPhone!"

Out: "i", "just", "love", "my", "iphone"

N-Gram Tokenizer

Reads the field text and generates n-gram tokens of sizes in the given range.

Factory class: `solr.NGramTokenizerFactory`

Arguments:

`minGramSize`: (integer, default 1) The minimum n-gram size, must be > 0.

`maxGramSize`: (integer, default 2) The maximum n-gram size, must be >= `minGramSize`.

Example:

Default behavior. Note that this tokenizer operates over the whole field. It does not break the field at whitespace. As a result, the space character is included in the encoding.

```
<analyzer>
  <tokenizer class="solr.NGramTokenizerFactory"/>
</analyzer>
```

In: "hey man"

Out: "h", "e", "y", " ", "m", "a", "n", "he", "ey", "y ", " m", "ma", "an"

Example:

With an n-gram size range of 4 to 5:

```
<analyzer>
  <tokenizer class="solr.NGramTokenizerFactory" minGramSize="4" maxGramSize="5"/>
</analyzer>
```

In: "bicycle"

Out: "bicy", "bicyc", "icyc", "icycl", "cycl", "cycle", "ycle"

Edge N-Gram Tokenizer

Reads the field text and generates edge n-gram tokens of sizes in the given range.

Factory class: `solr.EdgeNGramTokenizerFactory`

Arguments:

`minGramSize`: (integer, default is 1) The minimum n-gram size, must be > 0.

`maxGramSize`: (integer, default is 1) The maximum n-gram size, must be >= `minGramSize`.

`side`: ("front" or "back", default is "front") Whether to compute the n-grams from the beginning (front) of the text or from the end (back).

Example:

Default behavior (min and max default to 1):

```
<analyzer>
  <tokenizer class="solr.EdgeNGramTokenizerFactory"/>
</analyzer>
```

In: "babaloo"

Out: "b"

Example:

Edge n-gram range of 2 to 5

```
<analyzer>
  <tokenizer class="solr.EdgeNGramTokenizerFactory" minGramSize="2" maxGramSize="5"/>
</analyzer>
```

In: "babaloo"

Out:"ba", "bab", "baba", "babal"

Example:

Edge n-gram range of 2 to 5, from the back side:

```
<analyzer>
  <tokenizer class="solr.EdgeNGramTokenizerFactory" minGramSize="2" maxGramSize="5"
side="back"/>
</analyzer>
```

In: "babaloo"

Out: "oo", "loo", "aloo", "baloo"

ICU Tokenizer

This tokenizer processes multilingual text and tokenizes it appropriately based on its script attribute.

You can customize this tokenizer’s behavior by specifying per-script rule files. To add per-script rules, add a `rulefiles` argument, which should contain a comma-separated list of `code:rulefile` pairs in the following format: four-letter ISO 15924 script code, followed by a colon, then a resource path. For example, to specify rules for Latin (script code "Latn") and Cyrillic (script code "Cyrl"), you would enter `Latn:my.Latin.rules.rbbi,Cyrl:my.Cyrillic.rules.rbbi`.

The default `solr.ICUTokenizerFactory` provides UAX#29 word break rules tokenization (like `solr.StandardTokenizer`), but also includes custom tailorings for Hebrew (specializing handling of double and single quotation marks), and for syllable tokenization for Khmer, Lao, and Myanmar.

Factory class: `solr.ICUTokenizerFactory`

Arguments:

`rulefile`: a comma-separated list of `code:rulefile` pairs in the following format: four-letter ISO 15924 script code, followed by a colon, then a resource path.

Example:

```
  <analyzer>
    <!-- no customization -->
    <tokenizer class="solr.ICUTokenizerFactory"/>
  </analyzer>
```

```
  <analyzer>
    <tokenizer class="solr.ICUTokenizerFactory"
    rulefiles="Latn:my.Latin.rules.rbbi,Cyrl:my.Cyrillic.rules.rbbi"
  />
  </analyzer>
```

Path Hierarchy Tokenizer

This tokenizer creates synonyms from file path hierarchies.

Factory class: `solr.PathHierarchyTokenizerFactory`

Arguments:

`delimiter`: (character, no default) You can specify the file path delimiter and replace it with a delimiter you provide. This can be useful for working with backslash delimiters.

`replace`: (character, no default) Specifies the delimiter character Solr uses in the tokenized output.

Example:

```
<fieldType name="text_path" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.PathHierarchyTokenizerFactory" delimiter="\" replace="/"/>
  </analyzer>
</fieldType>
```

In: "c:\usr\local\apache"

Out: "c:", "c:/usr", "c:/usr/local", "c:/usr/local/apache"

Regular Expression Pattern Tokenizer

This tokenizer uses a Java regular expression to break the input text stream into tokens. The expression provided by the pattern argument can be interpreted either as a delimiter that separates tokens, or to match patterns that should be extracted from the text as tokens.

See the Javadocs for `java.util.regex.Pattern` for more information on Java regular expression syntax.

Factory class: `solr.PatternTokenizerFactory`

Arguments:

`pattern`: (Required) The regular expression, as defined by in `java.util.regex.Pattern`.

`group`: (Optional, default -1) Specifies which regex group to extract as the token(s). The value -1 means the regex should be treated as a delimiter that separates tokens. Non-negative group numbers (>= 0) indicate that character sequences matching that regex group should be converted to tokens. Group zero refers to the entire regex, groups greater than zero refer to parenthesized sub-expressions of the regex, counted from left to right.

Example:

A comma separated list. Tokens are separated by a sequence of zero or more spaces, a comma, and zero or more spaces.

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory" pattern="\s*,\s*"/>
</analyzer>
```

In: "fee,fie, foe , fum, foo"

Out: "fee", "fie", "foe", "fum", "foo"

Example:

Extract simple, capitalized words. A sequence of at least one capital letter followed by zero or more letters of either case is extracted as a token.

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory" pattern="[A-Z][A-Za-z]*" group="0"/>
</analyzer>
```

In: "Hello. My name is Inigo Montoya. You killed my father. Prepare to die."

Out: "Hello", "My", "Inigo", "Montoya", "You", "Prepare"

Example:

Extract part numbers which are preceded by "SKU", "Part" or "Part Number", case sensitive, with an optional semi-colon separator. Part numbers must be all numeric digits, with an optional hyphen. Regex capture groups are numbered by counting left parenthesis from left to right. Group 3 is the subexpression "[0-9-]+", which matches one or more digits or hyphens.

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory" pattern=
"(SKU|Part(\sNumber)?):?\s(\[0-9-\]+)" group="3"/>
</analyzer>
```

In: "SKU: 1234, Part Number 5678, Part: 126-987"

Out: "1234", "5678", "126-987"

UAX29 URL Email Tokenizer

This tokenizer splits the text field into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded, with the following exceptions:

- Periods (dots) that are not followed by whitespace are kept as part of the token.
- Words are split at hyphens, unless there is a number in the word, in which case the token is not split and the numbers and hyphen(s) are preserved.
- Recognizes top-level Internet domain names (validated against the white list in the IANA Root Zone Database when the tokenizer was generated); email addresses; `file://`, `http(s)://`, and `ftp://` addresses; IPv4 and IPv6 addresses; and preserves them as a single token.

The UAX29 URL Email Tokenizer supports Unicode standard annex UAX#29 word boundaries with the following token types: `<ALPHANUM>`, `<NUM>`, `<URL>`, `<EMAIL>`, `<SOUTHEAST_ASIAN>`, `<IDEOGRAPHIC>`, and `<HIRAGANA>`.

Factory class: `solr.UAX29URLEmailTokenizerFactory`

Arguments:

`maxTokenLength`: (integer, default 255) Solr ignores tokens that exceed the number of characters specified by `maxTokenLength`.

Example:

```
<analyzer>
  <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
</analyzer>
```

In: "Visit http://accarol.com/contact.htm?from=external&a=10 or e-mail bob.cratchet@accarol.com"

Out: "Visit", "http://accarol.com/contact.htm?from=external&a=10", "or", "e", "mail", "bob.cratchet@accarol.com"

White Space Tokenizer

Simple tokenizer that splits the text stream on whitespace and returns sequences of non-whitespace characters as tokens. Note that any punctuation *will* be included in the tokenization.

Factory class: `solr.WhitespaceTokenizerFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
</analyzer>
```

In: "To be, or what?"

Out: "To", "be,", "or", "what?"

Filter Descriptions

You configure each filter with a `<filter>` element in `schema.xml` as a child of `<analyzer>`, following the `<tokenizer>` element. Filter definitions should follow a tokenizer or another filter definition because they take a `TokenStream` as input. For example.

```
<fieldType name="text" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>...
  </analyzer>
</fieldType>
```

The class attribute names a factory class that will instantiate a filter object as needed. Filter factory classes must implement the `org.apache.solr.analysis.TokenFilterFactory` interface. Like tokenizers, filters are also instances of TokenStream and thus are producers of tokens. Unlike tokenizers, filters also consume tokens from a TokenStream. This allows you to mix and match filters, in any order you prefer, downstream of a tokenizer.

Arguments may be passed to tokenizer factories to modify their behavior by setting attributes on the `<filter>` element. For example:

```
<fieldType name="semicolonDelimited" class="solr.TextField">
  <analyzer type="query">
    <tokenizer class="solr.PatternTokenizerFactory" pattern="; " />
    <filter class="solr.LengthFilterFactory" min="2" max="7"/>
  </analyzer>
</fieldType>
```

The following sections describe the filter factories that are included in this release of Solr.

Filters discussed in this section:

ASCII Folding Filter

This filter converts alphabetic, numeric, and symbolic Unicode characters which are not in the Basic Latin Unicode block (the first 127 ASCII characters) to their ASCII equivalents, if one exists. This filter converts characters from the following Unicode blocks:

- C1 Controls and Latin-1 Supplement (PDF)
- Latin Extended-A (PDF)
- Latin Extended-B (PDF)
- Latin Extended Additional (PDF)

- Latin Extended-C (PDF)
- Latin Extended-D (PDF)
- IPA Extensions (PDF)
- Phonetic Extensions (PDF)
- Phonetic Extensions Supplement (PDF)
- General Punctuation (PDF)
- Superscripts and Subscripts (PDF)
- Enclosed Alphanumerics (PDF)
- Dingbats (PDF)
- Supplemental Punctuation (PDF)
- Alphabetic Presentation Forms (PDF)
- Halfwidth and Fullwidth Forms (PDF)

Factory class: `solr.ASCIIFoldingFilterFactory`

Arguments: None

Example:

```
<analyzer>
  <filter class="solr.ASCIIFoldingFilterFactory"/>
</analyzer>
```

In: "á" (Unicode character 00E1)

Out: "a" (ASCII character 97)

Beider-Morse Filter

Implements the Beider-Morse Phonetic Matching (BMPM) algorithm, which allows identification of similar names, even if they are spelled differently or in different languages. More information about how this works is available in the section on Phonetic Matching.

BeiderMorseFilter changed its behavior in Solr 5.0 (version 3.04 of the BMPM algorithm is implemented, while previous Solr versions implemented BMPM version 3.00 - see http://stevemorse.org/phoneticinfo.htm), so any index built using this filter with earlier versions of Solr will need to be rebuilt.

Factory class: `solr.BeiderMorseFilterFactory`

Arguments:

`nameType`: Types of names. Valid values are GENERIC, ASHKENAZI, or SEPHARDIC. If not processing Ashkenazi or Sephardic names, use GENERIC.

`ruleType`: Types of rules to apply. Valid values are APPROX or EXACT.

`concat`: Defines if multiple possible matches should be combined with a pipe ("|").

`languageSet`: The language set to use. The value "auto" will allow the Filter to identify the language, or a comma-separated list can be supplied.

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.BeiderMorseFilterFactory" nameType="GENERIC" ruleType="APPROX"
          concat="true" languageSet="auto">
  </filter>
</analyzer>
```

Classic Filter

This filter takes the output of the Classic Tokenizer and strips periods from acronyms and "'s" from possessives.

Factory class: `solr.ClassicFilterFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.ClassicTokenizerFactory"/>
  <filter class="solr.ClassicFilterFactory"/>
</analyzer>
```

In: "I.B.M. cat’s can’t"

Tokenizer to Filter: "I.B.M", "cat’s", "can’t"

Out: "IBM", "cat", "can’t"

Common Grams Filter

This filter creates word shingles by combining common tokens such as stop words with regular tokens. This is useful for creating phrase queries containing common words, such as "the cat." Solr normally ignores stop words in queried phrases, so searching for "the cat" would return all matches for the word "cat."

Factory class: `solr.CommonGramsFilterFactory`

Arguments:

`words`: (a common word file in .txt format) Provide the name of a common word file, such as `stopwords.txt`.

`format`: (optional) If the stopwords list has been formatted for Snowball, you can specify `format="snowball"` so Solr can read the stopwords file.

`ignoreCase`: (boolean) If true, the filter ignores the case of words when comparing them to the common word file. The default is false.

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.CommonGramsFilterFactory" words="stopwords.txt" ignoreCase="true
"/>
</analyzer>
```

In: "the Cat"

Tokenizer to Filter: "the", "Cat"

Out: "the_cat"

Collation Key Filter

Collation allows sorting of text in a language-sensitive way. It is usually used for sorting, but can also be used with advanced searches. We've covered this in much more detail in the section on Unicode Collation.

Daitch-Mokotoff Soundex Filter

Implements the Daitch-Mokotoff Soundex algorithm, which allows identification of similar names, even if they are spelled differently. More information about how this works is available in the section on Phonetic Matching.

Factory class: `solr.DaitchMokotoffSoundexFilterFactory`

Arguments:

`inject` : (true/false) If true (the default), then new phonetic tokens are added to the stream. Otherwise, tokens are replaced with the phonetic equivalent. Setting this to false will enable phonetic matching, but the exact spelling of the target word may not match.

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.DaitchMokotoffSoundexFilterFactory" inject="true"/>
</analyzer>
```

Double Metaphone Filter

This filter creates tokens using the ` DoubleMetaphone ` encoding algorithm from commons-codec. For more information, see the Phonetic Matching section.

Factory class: `solr.DoubleMetaphoneFilterFactory`

Arguments:

`inject`: (true/false) If true (the default), then new phonetic tokens are added to the stream. Otherwise, tokens are replaced with the phonetic equivalent. Setting this to false will enable phonetic matching, but the exact spelling of the target word may not match.

`maxCodeLength`: (integer) The maximum length of the code to be generated.

Example:

Default behavior for inject (true): keep the original token and add phonetic token(s) at the same position.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.DoubleMetaphoneFilterFactory"/>
</analyzer>
```

In: "four score and Kuczewski"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "Kuczewski"(4)

Out: "four"(1), "FR"(1), "score"(2), "SKR"(2), "and"(3), "ANT"(3), "Kuczewski"(4), "KSSK"(4), "KXFS"(4)

The phonetic tokens have a position increment of 0, which indicates that they are at the same position as the token they were derived from (immediately preceding). Note that "Kuczewski" has two encodings, which are added at the same position.

Example:

Discard original token (`inject="false"`).

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.DoubleMetaphoneFilterFactory" inject="false"/>
</analyzer>
```

In: "four score and Kuczewski"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "Kuczewski"(4)

Out: "FR"(1), "SKR"(2), "ANT"(3), "KSSK"(4), "KXFS"(4)

Note that "Kuczewski" has two encodings, which are added at the same position.

Edge N-Gram Filter

This filter generates edge n-gram tokens of sizes within the given range.

Factory class: `solr.EdgeNGramFilterFactory`

Arguments:

`minGramSize`: (integer, default 1) The minimum gram size.

`maxGramSize`: (integer, default 1) The maximum gram size.

Example:

Default behavior.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.EdgeNGramFilterFactory"/>
</analyzer>
```

In: "four score and twenty"

Tokenizer to Filter: "four", "score", "and", "twenty"

Out: "f", "s", "a", "t"

Example:

A range of 1 to 4.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.EdgeNGramFilterFactory" minGramSize="1" maxGramSize="4"/>
</analyzer>
```

In: "four score"

Tokenizer to Filter: "four", "score"

Out: "f", "fo", "fou", "four", "s", "sc", "sco", "scor"

Example:

A range of 4 to 6.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.EdgeNGramFilterFactory" minGramSize="4" maxGramSize="6"/>
</analyzer>
```

In: "four score and twenty"

Tokenizer to Filter: "four", "score", "and", "twenty"

Out: "four", "scor", "score", "twen", "twent", "twenty"

English Minimal Stem Filter

This filter stems plural English words to their singular form.

Factory class: `solr.EnglishMinimalStemFilterFactory`

Arguments: None

Example:

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory "/>
  <filter class="solr.EnglishMinimalStemFilterFactory"/>
</analyzer>
```

In: "dogs cats"

Tokenizer to Filter: "dogs", "cats"

Out: "dog", "cat"

Hunspell Stem Filter

The Hunspell Stem Filter provides support for several languages. You must provide the dictionary (`.dic`) and rules (`.aff`) files for each language you wish to use with the Hunspell Stem Filter. You can download those language files here.

Be aware that your results will vary widely based on the quality of the provided dictionary and rules files. For example, some languages have only a minimal word list with no morphological information. On the other hand, for languages that have no stemmer but do have an extensive dictionary file, the Hunspell stemmer may be a good choice.

Factory class: `solr.HunspellStemFilterFactory`

Arguments:

`dictionary`: (required) The path of a dictionary file.

`affix`: (required) The path of a rules file. `ignoreCase`: (boolean) controls whether matching is case sensitive or not. The default is false.

`strictAffixParsing`: (boolean) controls whether the affix parsing is strict or not. If true, an error while reading an affix rule causes a ParseException, otherwise is ignored. The default is true.

Example:

```
<analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.HunspellStemFilterFactory"
    dictionary="en_GB.dic"
    affix="en_GB.aff"
    ignoreCase="true"
    strictAffixParsing="true" />
</analyzer>
```

In: "jump jumping jumped"

Tokenizer to Filter: "jump", "jumping", "jumped"

Out: "jump", "jump", "jump"

Hyphenated Words Filter

This filter reconstructs hyphenated words that have been tokenized as two tokens because of a line break or other intervening whitespace in the field test. If a token ends with a hyphen, it is joined with the following token and the hyphen is discarded. Note that for this filter to work properly, the upstream tokenizer must not remove trailing hyphen characters. This filter is generally only useful at index time.

Factory class: `solr.HyphenatedWordsFilterFactory`

Arguments: None

Example:

```
<analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.HyphenatedWordsFilterFactory"/>
</analyzer>
```

In: "A hyphen- ated word"

Tokenizer to Filter: "A", "hyphen-", "ated", "word"

Out: "A", "hyphenated", "word"

ICU Folding Filter

This filter is a custom Unicode normalization form that applies the foldings specified in Unicode Technical Report 30 in addition to the `NFKC_Casefold` normalization form as described in ICU Normalizer 2 Filter. This filter is a better substitute for the combined behavior of the ASCII Folding Filter, Lower Case Filter, and ICU Normalizer 2 Filter.

To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

Factory class: `solr.ICUFoldingFilterFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ICUFoldingFilterFactory"/>
</analyzer>
```

For detailed information on this normalization form, see http://www.unicode.org/reports/tr30/tr30-4.html.

ICU Normalizer 2 Filter

This filter factory normalizes text according to one of five Unicode Normalization Forms as described in Unicode Standard Annex #15:

- NFC: (name="nfc" mode="compose") Normalization Form C, canonical decomposition
- NFD: (name="nfc" mode="decompose") Normalization Form D, canonical decomposition, followed by canonical composition
- NFKC: (name="nfkc" mode="compose") Normalization Form KC, compatibility decomposition
- NFKD: (name="nfkc" mode="decompose") Normalization Form KD, compatibility decomposition, followed by canonical composition
- NFKC_Casefold: (name="nfkc_cf" mode="compose") Normalization Form KC, with additional Unicode case folding. Using the ICU Normalizer 2 Filter is a better-performing substitution for the Lower Case Filter and NFKC normalization.

Factory class: `solr.ICUNormalizer2FilterFactory`

Arguments:

`name`: (string) The name of the normalization form; `nfc`, `nfd`, `nfkc`, `nfkd`, `nfkc_cf`

`mode`: (string) The mode of Unicode character composition and decomposition; `compose` or `decompose`

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ICUNormalizer2FilterFactory" name="nkc_cf" mode="compose"/>
</analyzer>
```

For detailed information about these Unicode Normalization Forms, see http://unicode.org/reports/tr15/.

To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

ICU Transform Filter

This filter applies ICU Tranforms to text. This filter supports only ICU System Transforms. Custom rule sets are not supported.

Factory class: `solr.ICUTransformFilterFactory`

Arguments:

`id`: (string) The identifier for the ICU System Transform you wish to apply with this filter. For a full list of ICU System Transforms, see http://demo.icu-project.org/icu-bin/translit?TEMPLATE_FILE=data/translit_rule_main.html.

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ICUTransformFilterFactory" id="Traditional-Simplified"/>
</analyzer>
```

For detailed information about ICU Transforms, see http://userguide.icu-project.org/transforms/general.

To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

Keep Words Filter

This filter discards all tokens except those that are listed in the given word list. This is the inverse of the Stop Words Filter. This filter can be useful for building specialized indices for a constrained set of terms.

Factory class: `solr.KeepWordFilterFactory`

Arguments:

`words`: (required) Path of a text file containing the list of keep words, one per line. Blank lines and lines that begin with "#" are ignored. This may be an absolute path, or a simple filename in the Solr config directory.

`ignoreCase`: (true/false) If **true** then comparisons are done case-insensitively. If this argument is true, then the words file is assumed to contain only lowercase words. The default is **false**.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

Example:

Where `keepwords.txt` contains:

`happy`

`funny`

`silly`

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"/>
</analyzer>
```

In: "Happy, sad or funny"

Tokenizer to Filter: "Happy", "sad", "or", "funny"

Out: "funny"

Example:

Same `keepwords.txt`, case insensitive:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.KeepWordFilterFactory" words="keepwords.txt" ignoreCase="true"/>
</analyzer>
```

In: "Happy, sad or funny"

Tokenizer to Filter: "Happy", "sad", "or", "funny"

Out: "Happy", "funny"

Example:

Using LowerCaseFilterFactory before filtering for keep words, no `ignoreCase` flag.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"/>
</analyzer>
```

In: "Happy, sad or funny"

Tokenizer to Filter: "Happy", "sad", "or", "funny"

Filter to Filter: "happy", "sad", "or", "funny"

Out: "happy", "funny"

KStem Filter

KStem is an alternative to the Porter Stem Filter for developers looking for a less aggressive stemmer. KStem was written by Bob Krovetz, ported to Lucene by Sergio Guzman-Lara (UMASS Amherst). This stemmer is only appropriate for English language text.

Factory class: `solr.KStemFilterFactory`

Arguments: None

Example:

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory "/>
  <filter class="solr.KStemFilterFactory"/>
</analyzer>
```

In: "jump jumping jumped"

Tokenizer to Filter: "jump", "jumping", "jumped"

Out: "jump", "jump", "jump"

Length Filter

This filter passes tokens whose length falls within the min/max limit specified. All other tokens are discarded.

Factory class: `solr.LengthFilterFactory`

Arguments:

`min`: (integer, required) Minimum token length. Tokens shorter than this are discarded.

`max`: (integer, required, must be >= min) Maximum token length. Tokens longer than this are discarded.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens.

This argument is invalid if `luceneMatchVersion` is `5.0` or later.

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LengthFilterFactory" min="3" max="7"/>
</analyzer>
```

In: "turn right at Albuquerque"

Tokenizer to Filter: "turn", "right", "at", "Albuquerque"

Out: "turn", "right"

Lower Case Filter

Converts any uppercase letters in a token to the equivalent lowercase token. All other characters are left unchanged.

Factory class: `solr.LowerCaseFilterFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
```

In: "Down With CamelCase"

Tokenizer to Filter: "Down", "With", "CamelCase"

Out: "down", "with", "camelcase"

Managed Stop Filter

This is specialized version of the Stop Words Filter Factory that uses a set of stop words that are managed from a REST API.

Arguments:

`managed`: The name that should be used for this set of stop words in the managed REST API.

Example:

With this configuration the set of words is named "english" and can be managed via `/solr/collection_name/schema/analysis/stopwords/english`

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ManagedStopFilterFactory" managed="english"/>
</analyzer>
```

See Stop Filter for example input/output.

Managed Synonym Filter

This is specialized version of the Synonym Filter Factory that uses a mapping on synonyms that is managed from a REST API.

Arguments:

`managed`: The name that should be used for this mapping on synonyms in the managed REST API.

Example:

With this configuration the set of mappings is named "english" and can be managed via `/solr/collection_name/schema/analysis/synonyms/english`

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ManagedSynonymFilterFactory" managed="english"/>
</analyzer>
```

See Synonym Filter for example input/output.

N-Gram Filter

Generates n-gram tokens of sizes in the given range. Note that tokens are ordered by position and then by gram size.

Factory class: `solr.NGramFilterFactory`

Arguments:

`minGramSize`: (integer, default 1) The minimum gram size.

`maxGramSize`: (integer, default 2) The maximum gram size.

Example:

Default behavior.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.NGramFilterFactory"/>
</analyzer>
```

In: "four score"

Tokenizer to Filter: "four", "score"

Out: "f", "o", "u", "r", "fo", "ou", "ur", "s", "c", "o", "r", "e", "sc", "co", "or", "re"

Example:

A range of 1 to 4.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.NGramFilterFactory" minGramSize="1" maxGramSize="4"/>
</analyzer>
```

In: "four score"

Tokenizer to Filter: "four", "score"

Out: "f", "fo", "fou", "four", "s", "sc", "sco", "scor"

Example:

A range of 3 to 5.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.NGramFilterFactory" minGramSize="3" maxGramSize="5"/>
</analyzer>
```

In: "four score"

Tokenizer to Filter: "four", "score"

Out: "fou", "four", "our", "sco", "scor", "score", "cor", "core", "ore"

Numeric Payload Token Filter

This filter adds a numeric floating point payload value to tokens that match a given type. Refer to the Javadoc for the `org.apache.lucene.analysis.Token` class for more information about token types and payloads.

Factory class: `solr.NumericPayloadTokenFilterFactory`

Arguments:

`payload`: (required) A floating point value that will be added to all matching tokens.

`typeMatch`: (required) A token type name string. Tokens with a matching type name will have their payload set to the above floating point value.

Example:

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.NumericPayloadTokenFilterFactory" payload="0.75" typeMatch="word
"/>
</analyzer>
```

In: "bing bang boom"

Tokenizer to Filter: "bing", "bang", "boom"

Out: "bing"[0.75], "bang"[0.75], "boom"[0.75]

Pattern Replace Filter

This filter applies a regular expression to each token and, for those that match, substitutes the given replacement string in place of the matched pattern. Tokens which do not match are passed though unchanged.

Factory class: `solr.PatternReplaceFilterFactory`

Arguments:

`pattern`: (required) The regular expression to test against each token, as per `java.util.regex.Pattern`.

`replacement`: (required) A string to substitute in place of the matched pattern. This string may contain references to capture groups in the regex pattern. See the Javadoc for `java.util.regex.Matcher`.

`replace`: ("all" or "first", default "all") Indicates whether all occurrences of the pattern in the token should be replaced, or only the first.

Example:

Simple string replace:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.PatternReplaceFilterFactory" pattern="cat" replacement="dog"/>
</analyzer>
```

In: "cat concatenate catycat"

Tokenizer to Filter: "cat", "concatenate", "catycat"

Out: "dog", "condogenate", "dogydog"

Example:

String replacement, first occurrence only:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.PatternReplaceFilterFactory" pattern="cat" replacement="dog"
replace="first"/>
</analyzer>
```

In: "cat concatenate catycat"

Tokenizer to Filter: "cat", "concatenate", "catycat"

Out: "dog", "condogenate", "dogycat"

Example:

More complex pattern with capture group reference in the replacement. Tokens that start with non-numeric characters and end with digits will have an underscore inserted before the numbers. Otherwise the token is passed through.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.PatternReplaceFilterFactory" pattern="(\D+)(\d+)$" replacement=
"$1_$2"/>
</analyzer>
```

In: "cat foo1234 9987 blah1234foo"

Tokenizer to Filter: "cat", "foo1234", "9987", "blah1234foo"

Out: "cat", "foo_1234", "9987", "blah1234foo"

Phonetic Filter

This filter creates tokens using one of the phonetic encoding algorithms in the `org.apache.commons.codec.language` package. For more information, see the section on Phonetic Matching.

Factory class: `solr.PhoneticFilterFactory`

Arguments:

`encoder`: (required) The name of the encoder to use. The encoder name must be one of the following (case insensitive): DoubleMetaphone, Metaphone, Soundex, RefinedSoundex, Caverphone (v2.0), ColognePhonetic, or Nysiis.

`inject`: (true/false) If true (the default), then new phonetic tokens are added to the stream. Otherwise, tokens are replaced with the phonetic equivalent. Setting this to false will enable phonetic matching, but the exact spelling of the target word may not match.

`maxCodeLength`: (integer) The maximum length of the code to be generated by the Metaphone or Double Metaphone encoders.

Example:

Default behavior for DoubleMetaphone encoding.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.PhoneticFilterFactory" encoder="DoubleMetaphone"/>
</analyzer>
```

In: "four score and twenty"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "twenty"(4)

Out: "four"(1), "FR"(1), "score"(2), "SKR"(2), "and"(3), "ANT"(3), "twenty"(4), "TNT"(4)

The phonetic tokens have a position increment of 0, which indicates that they are at the same position as the token they were derived from (immediately preceding).

Example:

Discard original token.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.PhoneticFilterFactory" encoder="DoubleMetaphone" inject="false"/>
</analyzer>
```

In: "four score and twenty"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "twenty"(4)

Out: "FR"(1), "SKR"(2), "ANT"(3), "TWNT"(4)

Example:

Default Soundex encoder.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.PhoneticFilterFactory" encoder="Soundex"/>
</analyzer>
```

In: "four score and twenty"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "twenty"(4)

Out: "four"(1), "F600"(1), "score"(2), "S600"(2), "and"(3), "A530"(3), "twenty"(4), "T530"(4)

Porter Stem Filter

This filter applies the Porter Stemming Algorithm for English. The results are similar to using the Snowball Porter Stemmer with the `language="English"` argument. But this stemmer is coded directly in Java and is not based on Snowball. It does not accept a list of protected words and is only appropriate for English language text. However, it has been benchmarked as four times faster than the English Snowball stemmer, so can provide a performance enhancement.

Factory class: `solr.PorterStemFilterFactory`

Arguments: None

Example:

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory "/>
  <filter class="solr.PorterStemFilterFactory"/>
</analyzer>
```

In: "jump jumping jumped"

Tokenizer to Filter: "jump", "jumping", "jumped"

Out: "jump", "jump", "jump"

Remove Duplicates Token Filter

The filter removes duplicate tokens in the stream. Tokens are considered to be duplicates if they have the same text and position values.

Factory class: `solr.RemoveDuplicatesTokenFilterFactory`

Arguments: None

Example:

One example of where `RemoveDuplicatesTokenFilterFactory` is in situations where a synonym file is being used in conjuntion with a stemmer causes some synonyms to be reduced to the same stem. Consider the following entry from a `synonyms.txt` file:

```
Television, Televisions, TV, TVs
```

When used in the following configuration:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"/>
  <filter class="solr.EnglishMinimalStemFilterFactory"/>
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer>
```

In: "Watch TV"

Tokenizer to Synonym Filter: "Watch"(1) "TV"(2)

Synonym Filter to Stem Filter: "Watch"(1) "Television"(2) "Televisions"(2) "TV"(2) "TVs"(2)

Stem Filter to Remove Dups Filter: "Watch"(1) "Television"(2) "Television"(2) "TV"(2) "TV"(2)

Out: "Watch"(1) "Television"(2) "TV"(2)

Reversed Wildcard Filter

This filter reverses tokens to provide faster leading wildcard and prefix queries. Tokens without wildcards are not reversed.

Factory class: `solr.ReversedWildcardFilterFactory`

Arguments:

`withOriginal` (boolean) If true, the filter produces both original and reversed tokens at the same positions. If false, produces only reversed tokens.

`maxPosAsterisk` (integer, default = 2) The maximum position of the asterisk wildcard ('*') that triggers the reversal of the query term. Terms with asterisks at positions above this value are not reversed.

`maxPosQuestion` (integer, default = 1) The maximum position of the question mark wildcard ('?') that triggers the reversal of query term. To reverse only pure suffix queries (queries with a single leading asterisk), set this to 0 and `maxPosAsterisk` to 1.

`maxFractionAsterisk` (float, default = 0.0) An additional parameter that triggers the reversal if asterisk ('*') position is less than this fraction of the query token length.

`minTrailing` (integer, default = 2) The minimum number of trailing characters in a query token after the last wildcard character. For good performance this should be set to a value larger than 1.

Example:

```
<analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.ReversedWildcardFilterFactory" withOriginal="true"
    maxPosAsterisk="2" maxPosQuestion="1" minTrailing="2" maxFractionAsterisk="0"/>
</analyzer>
```

In: "*foo *bar"

Tokenizer to Filter: "*foo", "*bar"

Out: "oof*", "rab*"

Shingle Filter

This filter constructs shingles, which are token n-grams, from the token stream. It combines runs of tokens into a single token.

Factory class: `solr.ShingleFilterFactory`

Arguments:

`minShingleSize`: (integer, default 2) The minimum number of tokens per shingle.

`maxShingleSize`: (integer, must be >= 2, default 2) The maximum number of tokens per shingle.

`outputUnigrams`: (true/false) If true (the default), then each individual token is also included at its original position.

`outputUnigramsIfNoShingles`: (true/false) If false (the default), then individual tokens will be output if no shingles are possible.

`tokenSeparator`: (string, default is " ") The default string to use when joining adjacent tokens to form a shingle.

Example:

Default behavior.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ShingleFilterFactory"/>
</analyzer>
```

In: "To be, or what?"

Tokenizer to Filter: "To"(1), "be"(2), "or"(3), "what"(4)

Out: "To"(1), "To be"(1), "be"(2), "be or"(2), "or"(3), "or what"(3), "what"(4)

Example:

A shingle size of four, do not include original token.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ShingleFilterFactory" maxShingleSize="4" outputUnigrams="false"/>
</analyzer>
```

In: "To be, or not to be."

Tokenizer to Filter: "To"(1), "be"(2), "or"(3), "not"(4), "to"(5), "be"(6)

Out: "To be"(1), "To be or"(1), "To be or not"(1), "be or"(2), "be or not"(2), "be or not to"(2), "or not"(3), "or not to"(3), "or not to be"(3), "not to"(4), "not to be"(4), "to be"(5)

Snowball Porter Stemmer Filter

This filter factory instantiates a language-specific stemmer generated by Snowball. Snowball is a software package that generates pattern-based word stemmers. This type of stemmer is not as accurate as a table-based stemmer, but is faster and less complex. Table-driven stemmers are labor intensive to create and maintain and so are typically commercial products.

Solr contains Snowball stemmers for Armenian, Basque, Catalan, Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish and Turkish. For more information on Snowball, visit http://snowball.tartarus.org/.

`StopFilterFactory`, `CommonGramsFilterFactory`, and `CommonGramsQueryFilterFactory` can optionally read stopwords in Snowball format (specify `format="snowball"` in the configuration of those FilterFactories).

Factory class: `solr.SnowballPorterFilterFactory`

Arguments:

`language`: (default "English") The name of a language, used to select the appropriate Porter stemmer to use. Case is significant. This string is used to select a package name in the "org.tartarus.snowball.ext" class hierarchy.

`protected`: Path of a text file containing a list of protected words, one per line. Protected words will not be stemmed. Blank lines and lines that begin with "#" are ignored. This may be an absolute path, or a simple file name in the Solr config directory.

Example:

Default behavior:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SnowballPorterFilterFactory"/>
</analyzer>
```

In: "flip flipped flipping"

Tokenizer to Filter: "flip", "flipped", "flipping"

Out: "flip", "flip", "flip"

Example:

French stemmer, English words:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="French"/>
</analyzer>
```

In: "flip flipped flipping"

Tokenizer to Filter: "flip", "flipped", "flipping"

Out: "flip", "flipped", "flipping"

Example:

Spanish stemmer, Spanish words:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Spanish"/>
</analyzer>
```

In: "cante canta"

Tokenizer to Filter: "cante", "canta"

Out: "cant", "cant"

Standard Filter

This filter removes dots from acronyms and the substring "'s" from the end of tokens. This filter depends on the tokens being tagged with the appropriate term-type to recognize acronyms and words with apostrophes.

Factory class: `solr.StandardFilterFactory`

Arguments: None

Note:

This filter is no longer operational in Solr when the `luceneMatchVersion` (in `solrconfig.xml`) is higher than "3.1".

Stop Filter

This filter discards, or *stops* analysis of, tokens that are on the given stop words list. A standard stop words list is included in the Solr config directory, named `stopwords.txt`, which is appropriate for typical English language text.

Factory class: `solr.StopFilterFactory`

Arguments:

`words`: (optional) The path to a file that contains a list of stop words, one per line. Blank lines and lines that begin with "#" are ignored. This may be an absolute path, or path relative to the Solr config directory.

`format`: (optional) If the stopwords list has been formatted for Snowball, you can specify `format="snowball"` so Solr can read the stopwords file.

`ignoreCase`: (true/false, default false) Ignore case when testing for stop words. If true, the stop list should contain lowercase words.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.4` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

Example:

Case-sensitive matching, capitalized words not stopped. Token positions skip stopped words.

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
</analyzer>
```

In: "To be or what?"

Tokenizer to Filter: "To"(1), "be"(2), "or"(3), "what"(4)

Out: "To"(1), "what"(4)

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
</analyzer>
```

In: "To be or what?"

Tokenizer to Filter: "To"(1), "be"(2), "or"(3), "what"(4)

Out: "what"(4)

Suggest Stop Filter

Like Stop Filter, this filter discards, or *stops* analysis of, tokens that are on the given stop words list. Suggest Stop Filter differs from Stop Filter in that it will not remove the last token unless it is followed by a token separator. For example, a query “find the” would preserve the ‘the’ since it was not followed by a space, punctuation etc., and mark it as a `KEYWORD` so that following filters will not change or remove it. By contrast, a query like “find the popsicle” would remove “the” as a stopword, since it’s followed by a space. When using one of the analyzing suggesters, you would normally use the ordinary `StopFilterFactory` in your index analyzer and then SuggestStopFilter in your query analyzer.

Factory class: `solr.SuggestStopFilterFactory`

Arguments:

`words`: (optional; default: ` StopAnalyzer#ENGLISH_STOP_WORDS_SET `) The name of a stopwords file to parse.

`format`: (optional; default: `wordset`) Defines how the words file will be parsed. If `words` is not specified, then `format` must not be specified. The valid values for the format option are:

- `wordset`: This is the default format, which supports one word per line (including any intra-word whitespace) and allows whole line comments begining with the “#” character. Blank lines are ignored.
- `snowball`: This format allows for multiple words specified on each line, and trailing comments may be specified using the vertical line (“|”). Blank lines are ignored.

`ignoreCase`: (optional; default: `false`) If `true`, matching is case-insensitive.

Example:

```
<analyzer type="query">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SuggestStopFilterFactory" ignoreCase="true"
          words="stopwords.txt" format="wordset"/>
</analyzer>
```

In: "The The"

Tokenizer to Filter: "the"(1), "the"(2)

Out: "the"(2)

Synonym Filter

This filter does synonym mapping. Each token is looked up in the list of synonyms and if a match is found, then the synonym is emitted in place of the token. The position value of the new tokens are set such they all occur at the same position as the original token.

Factory class: `solr.SynonymFilterFactory`

Arguments:

`synonyms`: (required) The path of a file that contains a list of synonyms, one per line. In the (default) `solr` format - see the `format` argument below for alternatives - blank lines and lines that begin with "#" are ignored. This may be an absolute path, or path relative to the Solr config directory. There are two ways to specify synonym mappings:

- A comma-separated list of words. If the token matches any of the words, then all the words in the list are substituted, which will include the original token.
- Two comma-separated lists of words with the symbol "⇒" between them. If the token matches any word on the left, then the list on the right is substituted. The original token will not be included unless it is also in the list on the right.

`ignoreCase`: (optional; default: `false`) If `true`, synonyms will be matched case-insensitively.

`expand`: (optional; default: `true`) If `true`, a synonym will be expanded to all equivalent synonyms. If `false`, all equivalent synonyms will be reduced to the first in the list.

`format`: (optional; default: `solr`) Controls how the synonyms will be parsed. The short names `solr` (for ` SolrSynonymParser)` and `wordnet` (for ` WordnetSynonymParser `) are supported, or you may alternatively supply the name of your own ` SynonymMap.Builder ` subclass.

`tokenizerFactory`: (optional; default: `WhitespaceTokenizerFactory`) The name of the tokenizer factory to use when parsing the synonyms file. Arguments with the name prefix "tokenizerFactory."` will be supplied as init params to the specified tokenizer factory. Any arguments not consumed by the synonym filter factory, including those without the "`tokenizerFactory." prefix, will also be supplied as init params to the tokenizer factory. If `tokenizerFactory` is specified, then `analyzer` may not be, and vice versa.

`analyzer`: (optional; default: `WhitespaceTokenizerFactory`) The name of the analyzer class to use when parsing the synonyms file. If `analyzer` is specified, then `tokenizerFactory` may not be, and vice versa.

For the following examples, assume a synonyms file named `mysynonyms.txt`:

```
couch,sofa,divan
teh => the
huge,ginormous,humungous => large
small => tiny,teeny,weeny
```

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SynonymFilterFactory" synonyms="mysynonyms.txt"/>
</analyzer>
```

In: "teh small couch"

Tokenizer to Filter: "teh"(1), "small"(2), "couch"(3)

Out: "the"(1), "tiny"(2), "teeny"(2), "weeny"(2), "couch"(3), "sofa"(3), "divan"(3)

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory "/>
  <filter class="solr.SynonymFilterFactory" synonyms="mysynonyms.txt"/>
</analyzer>
```

In: "teh ginormous, humungous sofa"

Tokenizer to Filter: "teh"(1), "ginormous"(2), "humungous"(3), "sofa"(4)

Out: "the"(1), "large"(2), "large"(3), "couch"(4), "sofa"(4), "divan"(4)

Token Offset Payload Filter

This filter adds the numeric character offsets of the token as a payload value for that token.

Factory class: `solr.TokenOffsetPayloadTokenFilterFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.TokenOffsetPayloadTokenFilterFactory"/>
</analyzer>
```

In: "bing bang boom"

Tokenizer to Filter: "bing", "bang", "boom"

Out: "bing"[0,4], "bang"[5,9], "boom"[10,14]

Trim Filter

This filter trims leading and/or trailing whitespace from tokens. Most tokenizers break tokens at whitespace, so this filter is most often used for special situations.

Factory class: `solr.TrimFilterFactory`

Arguments:

`updateOffsets`: if `luceneMatchVersion` is `4.3` or earlier and `updateOffsets="true"`, trimmed tokens' start and end offsets will be updated to those of the first and last characters (plus one) remaining in the token. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

Example:

The PatternTokenizerFactory configuration used here splits the input on simple commas, it does not remove whitespace.

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory" pattern=","/>
  <filter class="solr.TrimFilterFactory"/>
</analyzer>
```

In: "one, two , three ,four "

Tokenizer to Filter: "one", " two ", " three ", "four "

Out: "one", "two", "three", "four"

Type As Payload Filter

This filter adds the token's type, as an encoded byte sequence, as its payload.

Factory class: `solr.TypeAsPayloadTokenFilterFactory`

Arguments: None

Example:

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.TypeAsPayloadTokenFilterFactory"/>
</analyzer>
```

In: "Pay Bob's I.O.U."

Tokenizer to Filter: "Pay", "Bob's", "I.O.U."

Out: "Pay"[<ALPHANUM>], "Bob's"[<APOSTROPHE>], "I.O.U."[<ACRONYM>]

Type Token Filter

This filter blacklists or whitelists a specified list of token types, assuming the tokens have type metadata associated with them. For example, the UAX29 URL Email Tokenizer emits "<URL>" and "<EMAIL>" typed tokens, as well as other types. This filter would allow you to pull out only e-mail addresses from text as tokens, if you wish.

Factory class: `solr.TypeTokenFilterFactory`

Arguments:

`types`: Defines the location of a file of types to filter.

`useWhitelist`: If **true**, the file defined in `types` should be used as include list. If **false**, or undefined, the file defined in `types` is used as a blacklist.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

Example:

```
<analyzer>
  <filter class="solr.TypeTokenFilterFactory" types="stoptypes.txt" useWhitelist="true
"/>
</analyzer>
```

Word Delimiter Filter

This filter splits tokens at word delimiters. The rules for determining delimiters are determined as follows:

- A change in case within a word: "CamelCase" → "Camel", "Case". This can be disabled by setting `splitOnCaseChange="0"`.
- A transition from alpha to numeric characters or vice versa: "Gonzo5000" → "Gonzo", "5000" "4500XL" → "4500", "XL". This can be disabled by setting `splitOnNumerics="0"`.
- Non-alphanumeric characters (discarded): "hot-spot" → "hot", "spot"
- A trailing "'s" is removed: "O'Reilly's" → "O", "Reilly"
- Any leading or trailing delimiters are discarded: "--hot-spot--" → "hot", "spot"

Factory class: `solr.WordDelimiterFilterFactory`

Arguments:

`generateWordParts`: (integer, default 1) If non-zero, splits words at delimiters. For example:"CamelCase", "hot-spot" → "Camel", "Case", "hot", "spot"

`generateNumberParts`: (integer, default 1) If non-zero, splits numeric strings at delimiters:"1947-32" →"1947", "32"

`splitOnCaseChange`: (integer, default 1) If 0, words are not split on camel-case changes:"BugBlaster-XL" → "BugBlaster", "XL". Example 1 below illustrates the default (non-zero) splitting behavior.

`splitOnNumerics`: (integer, default 1) If 0, don't split words on transitions from alpha to numeric:"FemBot3000" → "Fem", "Bot3000"

`catenateWords`: (integer, default 0) If non-zero, maximal runs of word parts will be joined: "hot-spot-sensor's" → "hotspotsensor"

`catenateNumbers`: (integer, default 0) If non-zero, maximal runs of number parts will be joined: 1947-32" → "194732"

`catenateAll`: (0/1, default 0) If non-zero, runs of word and number parts will be joined: "Zap-Master-9000" → "ZapMaster9000"

`preserveOriginal`: (integer, default 0) If non-zero, the original token is preserved: "Zap-Master-9000" → "Zap-Master-9000", "Zap", "Master", "9000"

`protected`: (optional) The pathname of a file that contains a list of protected words that should be passed through without splitting.

`stemEnglishPossessive`: (integer, default 1) If 1, strips the possessive "'s" from each subword.

Example:

Default behavior. The whitespace tokenizer is used here to preserve non-alphanumeric characters.

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.WordDelimiterFilterFactory"/>
</analyzer>
```

In: "hot-spot RoboBlaster/9000 100XL"

Tokenizer to Filter: "hot-spot", "RoboBlaster/9000", "100XL"

Out: "hot", "spot", "Robo", "Blaster", "9000", "100", "XL"

Example:

Do not split on case changes, and do not generate number parts. Note that by not generating number parts, tokens containing only numeric parts are ultimately discarded.

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.WordDelimiterFilterFactory" generateNumberParts="0"
splitOnCaseChange="0"/>
</analyzer>
```

In: "hot-spot RoboBlaster/9000 100-42"

Tokenizer to Filter: "hot-spot", "RoboBlaster/9000", "100-42"

Out: "hot", "spot", "RoboBlaster", "9000"

Example:

Concatenate word parts and number parts, but not word and number parts that occur in the same token.

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.WordDelimiterFilterFactory" catenateWords="1" catenateNumbers="1
"/>
</analyzer>
```

In: "hot-spot 100+42 XL40"

Tokenizer to Filter: "hot-spot"(1), "100+42"(2), "XL40"(3)

Out: "hot"(1), "spot"(2), "hotspot"(2), "100"(3), "42"(4), "10042"(4), "XL"(5), "40"(6)

Example:

Concatenate all. Word and/or number parts are joined together.

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.WordDelimiterFilterFactory" catenateAll="1"/>
</analyzer>
```

In: "XL-4000/ES"

Tokenizer to Filter: "XL-4000/ES"(1)

Out: "XL"(1), "4000"(2), "ES"(3), "XL4000ES"(3)

Example:

Using a protected words list that contains "AstroBlaster" and "XL-5000" (among others).

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.WordDelimiterFilterFactory" protected="protwords.txt"/>
</analyzer>
```

In: "FooBar AstroBlaster XL-5000 ==ES-34-"

Tokenizer to Filter: "FooBar", "AstroBlaster", "XL-5000", "==ES-34-"

Out: "FooBar", "FooBar", "AstroBlaster", "XL-5000", "ES", "34"

CharFilterFactories

Char Filter is a component that pre-processes input characters. Char Filters can be chained like Token Filters and placed in front of a Tokenizer. Char Filters can add, change, or remove characters while preserving the original character offsets to support features like highlighting.

solr.MappingCharFilterFactory

This filter creates `org.apache.lucene.analysis.MappingCharFilter`, which can be used for changing one string to another (for example, for normalizing `é` to `e`.).

This filter requires specifying a `mapping` argument, which is the path and name of a file containing the mappings to perform.

Example:

```
<analyzer>
  <charFilter class="solr.MappingCharFilterFactory" mapping="mapping-FoldToASCII.txt"/>
  <tokenizer ...>
  [...]
</analyzer>
```

Mapping file syntax:

- Comment lines beginning with a hash mark (`#`), as well as blank lines, are ignored.
- Each non-comment, non-blank line consists of a mapping of the form: `"source" => "target"`
  - Double-quoted source string, optional whitespace, an arrow (`=>`), optional whitespace, double-quoted target string.
- Trailing comments on mapping lines are not allowed.
- The source string must contain at least one character, but the target string may be empty.
- The following character escape sequences are recognized within source and target strings:

Escape Sequence	**Resulting character (ECMA-48 alias)**	**Unicode character**	**Example Mapping Line**
`\\`	`\`	U+005C	`"\\" => "/"`
`\"`	`"`	U+0022	`"\"and\"" => "'and'"`
`\b`	backspace (BS)	U+0008	`"\b" => " "`
`\t`	tab (HT)	U+0009	`"\t" => ","`
`\n`	newline (LF)	U+000A	`"\n" => "<br>"`

Escape Sequence	Resulting character (ECMA-48 alias)	Unicode character	Example Mapping Line
`\f`	form feed (FF)	U+000C	`"\f" ⇒ "\n"`
`\r`	carriage return (CR)	U+000D	`"\r" ⇒ "/carriage-return/"`
`\uXXXX`	Unicode char referenced by the 4 hex digits	U+XXXX	`"\uFEFF" ⇒ ""`

A backslash followed by any other character is interpreted as if the character were present without the backslash.

solr.HTMLStripCharFilterFactory

This filter creates `org.apache.solr.analysis.HTMLStripCharFilter`. This Char Filter strips HTML from the input stream and passes the result to another Char Filter or a Tokenizer.

This filter:

- Removes HTML/XML tags while preserving other content.
- Removes attributes within tags and supports optional attribute quoting.
- Removes XML processing instructions, such as: <?foo bar?>
- Removes XML comments.
- Removes XML elements starting with <!>.
- Removes contents of <script> and <style> elements.
- Handles XML comments inside these elements (normal comment processing will not always work).
- Replaces numeric character entities references like `&#65;` or `&#x7f;` with the corresponding character.
- The terminating ';' is optional if the entity reference at the end of the input; otherwise the terminating ';' is mandatory, to avoid false matches on something like "Alpha&Omega Corp".
- Replaces all named character entity references with the corresponding character.
- `&nbsp;` is replaced with a space instead of the 0xa0 character.
- Newlines are substituted for block-level elements.
- <CDATA> sections are recognized.
- Inline tags, such as `<b>`, `<i>`, or `<span>` will be removed.
- Uppercase character entities like `quot`, `gt`, `lt` and `amp` are recognized and handled as lowercase.

The input need not be an HTML document. The filter removes only constructs that look like HTML. If the input doesn't include anything that looks like HTML, the filter won't remove any input.

The table below presents examples of HTML stripping.

Input	Output
`my <a href="www.foo.bar">link</a>`	my link
`<br>hello<!--comment-→`	hello
`hello<script><!-- f('<!--internal-→</script>'); -→</script>`	hello
`if a<b then print a;`	if a<b then print a;
`hello <td height=22 nowrap align="left">`	hello
`a<b &#65 Alpha&Omega Ω`	a<b A Alpha&Omega Ω

solr.ICUNormalizer2CharFilterFactory

This filter performs pre-tokenization Unicode normalization using ICU4J.

Arguments:

`name`: A Unicode Normalization Form, one of `nfc`, `nfkc`, `nfkc_cf`. Default is `nfkc_cf`.

`mode`: Either `compose` or `decompose`. Default is `compose`. Use `decompose` with `name="nfc"` or `name="nfkc"` to get NFD or NFKD, respectively.

`filter`: A UnicodeSet pattern. Codepoints outside the set are always left unchanged. Default is `[]` (the null set, no filtering - all codepoints are subject to normalization).

Example:

```
<analyzer>
  <charFilter class="solr.ICUNormalizer2CharFilterFactory"/>
  <tokenizer ...>
  [...]
</analyzer>
```

solr.PatternReplaceCharFilterFactory

This filter uses regular expressions to replace or change character patterns.

Arguments:

`pattern`: the regular expression pattern to apply to the incoming text.

`replacement`: the text to use to replace matching patterns.

You can configure this filter in `schema.xml` like this:

```
<analyzer>
  <charFilter class="solr.PatternReplaceCharFilterFactory"
              pattern="([nN][oO]\.)\s*(\d+)" replacement="$1$2"/>
  <tokenizer ...>
  [...]
</analyzer>
```

The table below presents examples of regex-based pattern replacement:

Input	Pattern	Replacement	Output	Description
seeing looking	`(\w+)(ing)`	`$1`	seeing look	Removes "ing" from the end of word.
seeing looking	`(\w+)ing`	`$1`	seeing look	Same as above. 2nd parentheses can be omitted.
No.1 NO. no. 543	`[nN][oO]\.\s*(\d+)`	`#$1`	#1 NO. #543	Replace some string literals
abc=1234=5678	`(\w+)=(\d+)=(\d+)`	`$3=$1=$2`	5678=abc=1234	Change the order of the groups.

Language Analysis

This section contains information about tokenizers and filters related to character set conversion or for use with specific languages.

For the European languages, tokenization is fairly straightforward. Tokens are delimited by white space and/or a relatively small set of punctuation characters. In other languages the tokenization rules are often not so simple. Some European languages may require special tokenization rules as well, such as rules for decompounding German words.

For information about language detection at index time, see Detecting Languages During Indexing.

Topics discussed in this section:

- KeywordMarkerFilterFactory
- StemmerOverrideFilterFactory
- Dictionary Compound Word Token Filter
- Unicode Collation
- ASCII Folding Filter
- Language-Specific Factories

KeywordMarkerFilterFactory

Protects words from being modified by stemmers. A customized protected word list may be specified with the "protected" attribute in the schema. Any words in the protected word list will not be modified by any stemmer in Solr.

A sample Solr `protwords.txt` with comments can be found in the `sample_techproducts_configs` config set directory:

```
<fieldtype name="myfieldtype" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt" />
    <filter class="solr.PorterStemFilterFactory" />
  </analyzer>
</fieldtype>
```

StemmerOverrideFilterFactory

Overrides stemming algorithms by applying a custom mapping, then protecting these terms from being modified by stemmers.

A customized mapping of words to stems, in a tab-separated file, can be specified to the "dictionary" attribute in the schema. Words in this mapping will be stemmed to the stems from the file, and will not be further changed by any stemmer.

A sample stemdict.txt with comments can be found in the Source Repository.

```
<fieldtype name="myfieldtype" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StemmerOverrideFilterFactory" dictionary="stemdict.txt" />
    <filter class="solr.PorterStemFilterFactory" />
  </analyzer>
</fieldtype>
```

Dictionary Compound Word Token Filter

This filter splits, or *decompounds*, compound words into individual words using a dictionary of the component words. Each input token is passed through unchanged. If it can also be decompounded into subwords, each subword is also added to the stream at the same logical position.

Compound words are most commonly found in Germanic languages.

Factory class

`solr.DictionaryCompoundWordTokenFilterFactory`

Arguments

`dictionary`

(required) The path of a file that contains a list of simple words, one per line. Blank lines and lines that begin with "#" are ignored. This path may be an absolute path, or path relative to the Solr config directory.

`minWordSize`

(integer, default 5) Any token shorter than this is not decompounded.

`minSubwordSize`

(integer, default 2) Subwords shorter than this are not emitted as tokens.

`maxSubwordSize`

(integer, default 15) Subwords longer than this are not emitted as tokens.

`onlyLongestMatch`

(true/false) If true (the default), only the longest matching subwords will generate new tokens.

Example

Assume that `germanwords.txt` contains at least the following words: `dumm kopf donau dampf schiff`

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.DictionaryCompoundWordTokenFilterFactory" dictionary=
"germanwords.txt"/>
</analyzer>
```

In

"Donaudampfschiff dummkopf"

Tokenizer to Filter

"Donaudampfschiff"(1), "dummkopf"(2),

Out

"Donaudampfschiff"(1), "Donau"(1), "dampf"(1), "schiff"(1), "dummkopf"(2), "dumm"(2), "kopf"(2)

Unicode Collation

Unicode Collation is a language-sensitive method of sorting text that can also be used for advanced search purposes.

Unicode Collation in Solr is fast, because all the work is done at index time.

Rather than specifying an analyzer within `<fieldtype … class="solr.TextField">`, the `solr.CollationField` and `solr.ICUCollationField` field type classes provide this functionality. `solr.ICUCollationField`, which is backed by the ICU4J library, provides more flexible configuration, has more locales, is significantly faster, and requires less memory and less index space, since its keys are smaller than those produced by the JDK implementation that backs `solr.CollationField`.

`solr.ICUCollationField` is included in the Solr `analysis-extras` contrib - see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `SOLR_HOME/lib` in order to use it.

`solr.ICUCollationField` and `solr.CollationField` fields can be created in two ways:

- Based upon a system collator associated with a Locale.
- Based upon a tailored `RuleBasedCollator` ruleset.

ICUCollationField Arguments

Arguments for `solr.ICUCollationField` are specified as attributes within the `<fieldtype>` element.

Using a System collator:

`locale`

(required) RFC 3066 locale ID. See the ICU locale explorer for a list of supported locales.

`strength`
: Valid values are `primary`, `secondary`, `tertiary`, `quaternary`, or `identical`. See Comparison Levels in ICU Collation Concepts for more information.

`decomposition`
: Valid values are `no` or `canonical`. See Normalization in ICU Collation Concepts for more information.

Using a Tailored ruleset:

`custom`
: (required) Path to a UTF-8 text file containing rules supported by the ICU ` RuleBasedCollator `

`strength`
: Valid values are `primary`, `secondary`, `tertiary`, `quaternary`, or `identical`. See Comparison Levels in ICU Collation Concepts for more information.

`decomposition`
: Valid values are `no` or `canonical`. See Normalization in ICU Collation Concepts for more information.

Expert options:

`alternate`
: Valid values are `shifted` or `non-ignorable`. Can be used to ignore punctuation/whitespace.

`caseLevel`
: (true/false) If true, in combination with `strength="primary"`, accents are ignored but case is taken into account. The default is false. See CaseLevel in ICU Collation Concepts for more information.

`caseFirst`
: Valid values are `lower` or `upper`. Useful to control which is sorted first when case is not ignored.

`numeric`
: (true/false) If true, digits are sorted according to numeric value, e.g. foobar-9 sorts before foobar-10. The default is false.

`variableTop`
: Single character or contraction. Controls what is variable for `alternate`

Sorting Text for a Specific Language

In this example, text is sorted according to the default German rules provided by ICU4J.

Locales are typically defined as a combination of language and country, but you can specify just the language if you want. For example, if you specify "de" as the language, you will get sorting that works well for the German language. If you specify "de" as the language and "CH" as the country, you will get German sorting specifically tailored for Switzerland.

```
<!-- Define a field type for German collation -->
<fieldType name="collatedGERMAN" class="solr.ICUCollationField"
           locale="de"
           strength="primary" />
...
<!-- Define a field to store the German collated manufacturer names. -->
<field name="manuGERMAN" type="collatedGERMAN" indexed="false" stored="false" docValues
="true"/>
...
<!-- Copy the text to this field. We could create French, English, Spanish versions too,
     and sort differently for different users! -->
<copyField source="manu" dest="manuGERMAN"/>
```

In the example above, we defined the strength as "primary". The strength of the collation determines how strict the sort order will be, but it also depends upon the language. For example, in English, "primary" strength ignores differences in case and accents.

Another example:

```
<fieldType name="polishCaseInsensitive" class="solr.ICUCollationField"
           locale="pl_PL"
           strength="secondary" />
...
<field name="city" type="text_general" indexed="true" stored="true"/>
...
<field name="city_sort" type="polishCaseInsensitive" indexed="true" stored="false"/>
...
<copyField source="city" dest="city_sort"/>
```

The type will be used for the fields where the data contains Polish text. The "secondary" strength will ignore case differences, but, unlike "primary" strength, a letter with diacritic(s) will be sorted differently from the same base letter without diacritics.

An example using the "city_sort" field to sort:

```
q=*:*&fl=city&sort=city_sort+asc
```

Sorting Text for Multiple Languages

There are two approaches to supporting multiple languages: if there is a small list of languages you wish to support, consider defining collated fields for each language and using `copyField`. However, adding a large number of sort fields can increase disk and indexing costs. An alternative approach is to use the Unicode `default` collator.

The Unicode `default` or `ROOT` locale has rules that are designed to work well for most languages. To use the `default` locale, simply define the locale as the empty string. This Unicode default sort is still significantly more advanced than the standard Solr sort.

```
<fieldType name="collatedROOT" class="solr.ICUCollationField"
           locale=""
           strength="primary" />
```

Sorting Text with Custom Rules

You can define your own set of sorting rules. It's easiest to take existing rules that are close to what you want and customize them.

In the example below, we create a custom rule set for German called DIN 5007-2. This rule set treats umlauts in German differently: it treats ö as equivalent to oe, ä as equivalent to ae, and ü as equivalent to ue. For more information, see the ICU RuleBasedCollator javadocs.

This example shows how to create a custom rule set for `solr.ICUCollationField` and dump it to a file:

```
// get the default rules for Germany
// these are called DIN 5007-1 sorting
RuleBasedCollator baseCollator = (RuleBasedCollator) Collator.getInstance(new ULocale(
"de", "DE"));

// define some tailorings, to make it DIN 5007-2 sorting.
// For example, this makes ö equivalent to oe
String DIN5007_2_tailorings =
    "& ae , a\u0308 & AE , A\u0308"+
    "& oe , o\u0308 & OE , O\u0308"+
    "& ue , u\u0308 & UE , u\u0308";

// concatenate the default rules to the tailorings, and dump it to a String
RuleBasedCollator tailoredCollator = new RuleBasedCollator(baseCollator.getRules() +
DIN5007_2_tailorings);
String tailoredRules = tailoredCollator.getRules();

// write these to a file, be sure to use UTF-8 encoding!!!
FileOutputStream os = new FileOutputStream(new File("/solr_home/conf/customRules.dat"));
IOUtils.write(tailoredRules, os, "UTF-8");
```

This rule set can now be used for custom collation in Solr:

```
---
<fieldType name="collatedCUSTOM" class="solr.ICUCollationField"
           custom="customRules.dat"
           strength="primary" />
---
```

JDK Collation

As mentioned above, ICU Unicode Collation is better in several ways than JDK Collation, but if you cannot use ICU4J for some reason, you can use `solr.CollationField`.

The principles of JDK Collation are the same as those of ICU Collation; you just specify `language`, `country` and `variant` arguments instead of the combined `locale` argument.

CollationField Arguments

Arguments for `solr.CollationField` are specified as attributes within the `<fieldtype>` element.

Using a System collator (see Oracle's list of locales supported in Java 7):

`language`
: (required) ISO-639 language code

`country`
: ISO-3166 country code

`variant`
: Vendor or browser-specific code

`strength`
: Valid values are `primary`, `secondary`, `tertiary` or `identical`. See Oracle Java 7 Collator javadocs for more information.

`decomposition`
: Valid values are `no`, `canonical`, or `full`. See Oracle Java 7 Collator javadocs for more information.

Using a Tailored ruleset:

`custom`
: (required) Path to a UTF-8 text file containing rules supported by the ` JDK RuleBasedCollator `

`strength`
: Valid values are `primary`, `secondary`, `tertiary` or `identical`. See Oracle Java 7 Collator javadocs for more information.

`decomposition`
: Valid values are `no`, `canonical`, or `full`. See Oracle Java 7 Collator javadocs for more information.

Example solr.CollationField

```
<fieldType name="collatedGERMAN" class="solr.CollationField"
           language="de"
           country="DE"
           strength="primary" /> <!-- ignore Umlauts and letter case when sorting -->
...
<field name="manuGERMAN" type="collatedGERMAN" indexed="false" stored="false" docValues
="true" />
...
<copyField source="manu" dest="manuGERMAN"/>
```

ASCII Folding Filter

This filter converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists. Only those characters with reasonable ASCII alternatives are converted:

This can increase recall by causing more matches. On the other hand, it can reduce precision because language-specific character differences may be lost.

Factory class
`solr.ASCIIFoldingFilterFactory`

Arguments
None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ASCIIFoldingFilterFactory"/>
</analyzer>
```

In
"Björn Ångström"

Tokenizer to Filter
"Björn", "Ångström"

Out
"Bjorn", "Angstrom"

Language-Specific Factories

These factories are each designed to work with specific languages.

Arabic

Solr provides support for the Light-10 (PDF) stemming algorithm, and Lucene includes an example stopword list.

This algorithm defines both character normalization and stemming, so these are split into two filters to provide more flexibility.

Factory classes

`solr.ArabicStemFilterFactory`, `solr.ArabicNormalizationFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ArabicNormalizationFilterFactory"/>
  <filter class="solr.ArabicStemFilterFactory"/>
</analyzer>
```

Brazilian Portuguese

This is a Java filter written specifically for stemming the Brazilian dialect of the Portuguese language. It uses the Lucene class `org.apache.lucene.analysis.br.BrazilianStemmer`. Although that stemmer can be configured to use a list of protected words (which should not be stemmed), this factory does not accept any arguments to specify such a list.

Factory class
`solr.BrazilianStemFilterFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.BrazilianStemFilterFactory"/>
</analyzer>
```

In

"praia praias"

Tokenizer to Filter

"praia", "praias"

Out

"pra", "pra"

Bulgarian

Solr includes a light stemmer for Bulgarian, following this algorithm (PDF), and Lucene includes an example stopword list.

Factory class

`solr.BulgarianStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.BulgarianStemFilterFactory"/>
</analyzer>
```

Catalan

Solr can stem Catalan using the Snowball Porter Stemmer with an argument of `language="Catalan"`. Solr includes a set of contractions for Catalan, which can be stripped using `solr.ElisionFilterFactory`.

Factory class

`solr.SnowballPorterFilterFactory`

Arguments

`language`

(required) stemmer language, "Catalan" in this case

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_ca.txt"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Catalan" />
</analyzer>
```

In

"llengües llengua"

Tokenizer to Filter

"llengües"(1) "llengua"(2),

Out

"llengu"(1), "llengu"(2)

Chinese

Chinese Tokenizer

The Chinese Tokenizer is deprecated as of Solr 3.4. Use the `solr.StandardTokenizerFactory` instead.

Factory class
`solr.ChineseTokenizerFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.ChineseTokenizerFactory"/>
</analyzer>
```

Chinese Filter Factory

The Chinese Filter Factory is deprecated as of Solr 3.4. Use the `solr.StopFilterFactory` instead.

Factory class
`solr.ChineseFilterFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ChineseFilterFactory"/>
</analyzer>
```

Simplified Chinese

For Simplified Chinese, Solr provides support for Chinese sentence and word segmentation with the `solr.HMMChineseTokenizerFactory` in the `analysis-extras` contrib module. This component includes a large dictionary and segments Chinese text into words with the Hidden Markov Model. To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

Factory class
: `solr.HMMChineseTokenizerFactory`

Arguments
: None

Examples
: To use the default setup with fallback to English Porter stemmer for English words, use:

`<analyzer class="org.apache.lucene.analysis.cn.smart.SmartChineseAnalyzer"/>`

Or to configure your own analysis setup, use the `solr.HMMChineseTokenizerFactory` along with your custom filter setup.

```
<analyzer>
  <tokenizer class="solr.HMMChineseTokenizerFactory"/>
  <filter class="solr.StopFilterFactory
          words="org/apache/lucene/analysis/cn/smart/stopwords.txt"/>
  <filter class="solr.PorterStemFilterFactory"/>
</analyzer>
```

CJK

This tokenizer breaks Chinese, Japanese and Korean language text into tokens. These are not whitespace delimited languages. The tokens generated by this tokenizer are "doubles", overlapping pairs of CJK characters found in the field text.

Factory class
: `solr.CJKTokenizerFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.CJKTokenizerFactory"/>
</analyzer>
```

Czech

Solr includes a light stemmer for Czech, following this algorithm, and Lucene includes an example stopword list.

Factory class
`solr.CzechStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.CzechStemFilterFactory"/>
<analyzer>
```

In

"prezidenští, prezidenta, prezidentského"

Tokenizer to Filter

"prezidenští", "prezidenta", "prezidentského"

Out

"preziden", "preziden", "preziden"

Danish

Solr can stem Danish using the Snowball Porter Stemmer with an argument of `language="Danish"`.

Also relevant are the Scandinavian normalization filters.

Factory class
`solr.SnowballPorterFilterFactory`

Arguments

`language`

(required) stemmer language, "Danish" in this case

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Danish" />
</analyzer>
```

In

"undersøg undersøgelse"

Tokenizer to Filter

"undersøg"(1) "undersøgelse"(2),

Out

"undersøg"(1), "undersøg"(2)

Dutch

Solr can stem Dutch using the Snowball Porter Stemmer with an argument of `language="Dutch"`.

Factory class

`solr.SnowballPorterFilterFactory`

Arguments

`language`

(required) stemmer language, "Dutch" in this case

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Dutch"/>
</analyzer>
```

In

"kanaal kanalen"

Tokenizer to Filter

"kanaal", "kanalen"

Out

"kanal", "kanal"

Finnish

Solr includes support for stemming Finnish, and Lucene includes an example stopword list.

Factory class
`solr.FinnishLightStemFilterFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.FinnishLightStemFilterFactory"/>
</analyzer>
```

In

"kala kalat"

Tokenizer to Filter

"kala", "kalat"

Out

"kala", "kala"

French

Elision Filter

Removes article elisions from a token stream. This filter can be useful for languages such as French, Catalan, Italian, and Irish.

Factory class
`solr.ElisionFilterFactory`

Arguments
`articles`
The pathname of a file that contains a list of articles, one per line, to be stripped. Articles are words such as "le", which are commonly abbreviated, such as in *l'avion* (the plane). This file should include the abbreviated form, which precedes the apostrophe. In this case, simply "*l*". If no `articles` attribute is specified, a default set of French articles is used.

`ignoreCase`
: (boolean) If true, the filter ignores the case of words when comparing them to the common word file. Defaults to `false`.

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ElisionFilterFactory"
          ignoreCase="true"
          articles="lang/contractions_fr.txt"/>
</analyzer>
```

In
: "L'histoire d'art"

Tokenizer to Filter
: "L'histoire", "d'art"

Out
: "histoire", "art"

French Light Stem Filter

Solr includes three stemmers for French: one in the `solr.SnowballPorterFilterFactory`, a lighter stemmer called `solr.FrenchLightStemFilterFactory`, and an even less aggressive stemmer called `solr.FrenchMinimalStemFilterFactory`. Lucene includes an example stopword list.

Factory classes
: `solr.FrenchLightStemFilterFactory`, `solr.FrenchMinimalStemFilterFactory`

Arguments
: None

Examples

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_fr.txt"/>
  <filter class="solr.FrenchLightStemFilterFactory"/>
</analyzer>
```

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_fr.txt"/>
  <filter class="solr.FrenchMinimalStemFilterFactory"/>
</analyzer>
```

In

"le chat, les chats"

Tokenizer to Filter

"le", "chat", "les", "chats"

Out

"le", "chat", "le", "chat"

Galician

Solr includes a stemmer for Galician following this algorithm, and Lucene includes an example stopword list.

Factory class
`solr.GalicianStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.GalicianStemFilterFactory"/>
</analyzer>
```

In

"felizmente Luzes"

Tokenizer to Filter

"felizmente", "luzes"

Out

"feliz", "luz"

German

Solr includes four stemmers for German:

- one in the `solr.SnowballPorterFilterFactory language="German"`,
- a stemmer called `solr.GermanStemFilterFactory`,
- a lighter stemmer called `solr.GermanLightStemFilterFactory`,
- and an even less aggressive stemmer called `solr.GermanMinimalStemFilterFactory`.

Lucene includes an example stopword list.

Factory classes::: `solr.GermanStemFilterFactory`, `solr.LightGermanStemFilterFactory`, `solr.MinimalGermanStemFilterFactory`

Arguments

None

Examples

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory "/>
  <filter class="solr.GermanStemFilterFactory"/>
</analyzer>
```

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.GermanLightStemFilterFactory"/>
</analyzer>
```

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory "/>
  <filter class="solr.GermanMinimalStemFilterFactory"/>
</analyzer>
```

In

"haus häuser"

Tokenizer to Filter

"haus", "häuser"

Out

"haus", "haus"

Greek

This filter converts uppercase letters in the Greek character set to the equivalent lowercase character.

Factory class

`solr.GreekLowerCaseFilterFactory`

Arguments

None

Use of custom charsets is not supported as of Solr 3.1. If you need to index text in these encodings, please use Java's character set conversion facilities (`InputStreamReader`, and so on.) during I/O, so that Lucene can analyze this text as Unicode instead.

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.GreekLowerCaseFilterFactory"/>
</analyzer>
```

Hindi

Solr includes support for stemming Hindi following this algorithm (PDF), support for common spelling differences through the `solr.HindiNormalizationFilterFactory`, support for encoding differences through the `solr.IndicNormalizationFilterFactory` following this algorithm, and Lucene includes an example stopword list.

Factory classes

`solr.IndicNormalizationFilterFactory`, `solr.HindiNormalizationFilterFactory`, `solr.HindiStemFilterFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.IndicNormalizationFilterFactory"/>
  <filter class="solr.HindiNormalizationFilterFactory"/>
  <filter class="solr.HindiStemFilterFactory"/>
</analyzer>
```

Indonesian

Solr includes support for stemming Indonesian (Bahasa Indonesia) following this algorithm (PDF), and Lucene includes an example stopword list.

Factory class

`solr.IndonesianStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.IndonesianStemFilterFactory" stemDerivational="true" />
</analyzer>
```

In

"sebagai sebagainya"

Tokenizer to Filter

"sebagai", "sebagainya"

Out

"bagai", "bagai"

Italian

Solr includes two stemmers for Italian: one in the `solr.SnowballPorterFilterFactory language="Italian"`, and a lighter stemmer called `solr.ItalianLightStemFilterFactory`. Lucene includes an example stopword list.

Factory class

`solr.ItalianStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_it.txt"/>
  <filter class="solr.ItalianLightStemFilterFactory"/>
</analyzer>
```

In

"propaga propagare propagamento"

Tokenizer to Filter

"propaga", "propagare", "propagamento"

Out

"propag", "propag", "propag"

Irish

Solr can stem Irish using the Snowball Porter Stemmer with an argument of `language="Irish"`. Solr includes `solr.IrishLowerCaseFilterFactory`, which can handle Irish-specific constructs. Solr also includes a set of contractions for Irish which can be stripped using `solr.ElisionFilterFactory`.

Factory class
`solr.SnowballPorterFilterFactory`

Arguments
`language`
(required) stemmer language, "Irish" in this case

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_ga.txt"/>
  <filter class="solr.IrishLowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Irish" />
</analyzer>
```

In

"siopadóireacht síceapatacha b’fhearr m’athair"

Tokenizer to Filter

"siopadóireacht", "síceapatacha", "b’fhearr", "m’athair"

Out

"siopadóir", "síceapaite", "fearr", "athair"

Japanese

Solr includes support for analyzing Japanese, via the Lucene Kuromoji morphological analyzer, which includes several analysis components - more details on each below:

- `JapaneseIterationMarkCharFilter` normalizes Japanese horizontal iteration marks (odoriji) to their expanded form.
- `JapaneseTokenizer` tokenizes Japanese using morphological analysis, and annotates each term with part-of-speech, base form (a.k.a. lemma), reading and pronunciation.
- `JapaneseBaseFormFilter` replaces original terms with their base forms (a.k.a. lemmas).
- `JapanesePartOfSpeechStopFilter` removes terms that have one of the configured parts-of-speech.
- `JapaneseKatakanaStemFilter` normalizes common katakana spelling variations ending in a long sound character (U+30FC) by removing the long sound character.

Also useful for Japanese analysis, from lucene-analyzers-common:

- `CJKWidthFilter` folds fullwidth ASCII variants into the equivalent Basic Latin forms, and folds halfwidth Katakana variants into their equivalent fullwidth forms.

Japanese Iteration Mark CharFilter

Normalizes horizontal Japanese iteration marks (odoriji) to their expanded form. Vertical iteration marks are not supported.

Factory class

`JapaneseIterationMarkCharFilterFactory`

Arguments

`normalizeKanji`: set to `false` to not normalize kanji iteration marks (default is `true`)

` normalizeKana`: set to `false` to not normalize kana iteration marks (default is `true`)

Japanese Tokenizer

Tokenizer for Japanese that uses morphological analysis, and annotates each term with part-of-speech, base form (a.k.a. lemma), reading and pronunciation.

`JapaneseTokenizer` has a `search` mode (the default) that does segmentation useful for search: a heuristic is used to segment compound terms into their constituent parts while also keeping the original compound terms as synonyms.

Factory class

`solr.JapaneseTokenizerFactory`

Arguments

`mode`

Use `search` mode to get a noun-decompounding effect useful for search. `search` mode improves segmentation for search at the expense of part-of-speech accuracy. Valid values for `mode` are:

- `normal`: default segmentation
- `search`: segmentation useful for search (extra compound splitting)
- `extended`: search mode plus unigramming of unknown words (experimental)

  For some applications it might be good to use `search` mode for indexing and `normal` mode for queries to increase precision and prevent parts of compounds from being matched and highlighted.

`userDictionary`

filename for a user dictionary, which allows overriding the statistical model with your own entries for segmentation, part-of-speech tags and readings without a need to specify weights. See `lang/userdict_ja.txt` for a sample user dictionary file.

`userDictionaryEncoding`

user dictionary encoding (default is UTF-8)

`discardPunctuation`

set to `false` to keep punctuation, `true` to discard (the default)

Japanese Base Form Filter

Replaces original terms' text with the corresponding base form (lemma). (`JapaneseTokenizer` annotates each term with its base form.)

Factory class

`JapaneseBaseFormFilterFactory`

Arguments

None

Japanese Part Of Speech Stop Filter

Removes terms with one of the configured parts-of-speech. `JapaneseTokenizer` annotates terms with parts-of-speech.

Factory class

`JapanesePartOfSpeechStopFilterFactory`

Arguments

`tags`

filename for a list of parts-of-speech for which to remove terms; see `conf/lang/stoptags_ja.txt` in the `sample_techproducts_config` config set for an example.

`enablePositionIncrements`

if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

Japanese Katakana Stem Filter

Normalizes common katakana spelling variations ending in a long sound character (U+30FC) by removing the long sound character.

`CJKWidthFilterFactory` should be specified prior to this filter to normalize half-width katakana to full-width.

Factory class

`JapaneseKatakanaStemFilterFactory`

Arguments

`minimumLength`

terms below this length will not be stemmed. Default is 4, value must be 2 or more.

CJK Width Filter

Folds fullwidth ASCII variants into the equivalent Basic Latin forms, and folds halfwidth Katakana variants into their equivalent fullwidth forms.

Factory class

`CJKWidthFilterFactory`

Arguments

None

Example

```
<fieldType name="text_ja" positionIncrementGap="100" autoGeneratePhraseQueries="
false">
  <analyzer>
    <!-- Uncomment if you need to handle iteration marks: -->
    <!-- <charFilter class="solr.JapaneseIterationMarkCharFilterFactory" /> -->
    <tokenizer class="solr.JapaneseTokenizerFactory" mode="search" userDictionary=
"lang/userdict_ja.txt"/>
    <filter class="solr.JapaneseBaseFormFilterFactory"/>
    <filter class="solr.JapanesePartOfSpeechStopFilterFactory" tags=
"lang/stoptags_ja.txt"/>
    <filter class="solr.CJKWidthFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words=
"lang/stopwords_ja.txt"/>
    <filter class="solr.JapaneseKatakanaStemFilterFactory" minimumLength="4"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Hebrew, Lao, Myanmar, Khmer

Lucene provides support, in addition to UAX#29 word break rules, for Hebrew's use of the double and single quote characters, and for segmenting Lao, Myanmar, and Khmer into syllables with the `solr.ICUTokenizerFactory` in the `analysis-extras` contrib module. To use this tokenizer, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

See the ICUTokenizer for more information.

Latvian

Solr includes support for stemming Latvian, and Lucene includes an example stopword list.

Factory class
: `solr.LatvianStemFilterFactory`

Arguments
: None

Example

```
<fieldType name="text_lvstem" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.LatvianStemFilterFactory"/>
  </analyzer>
</fieldType>
```

In

"tirgiem tirgus"

Tokenizer to Filter

"tirgiem", "tirgus"

Out

"tirg", "tirg"

Norwegian

Solr includes two classes for stemming Norwegian, `NorwegianLightStemFilterFactory` and `NorwegianMinimalStemFilterFactory`. Lucene includes an example stopword list.

Another option is to use the Snowball Porter Stemmer with an argument of language="Norwegian".

Also relevant are the Scandinavian normalization filters.

Norwegian Light Stemmer

The `NorwegianLightStemFilterFactory` requires a "two-pass" sort for the -dom and -het endings. This means that in the first pass the word "kristendom" is stemmed to "kristen", and then all the general rules apply so it will be further stemmed to "krist". The effect of this is that "kristen," "kristendom," "kristendommen," and "kristendommens" will all be stemmed to "krist."

The second pass is to pick up -dom and -het endings. Consider this example:

One pass		**Two passes**	
Before	**After**	**Before**	**After**
forlegen	forleg	forlegen	forleg
forlegenhet	forlegen	forlegenhet	forleg
forlegenheten	forlegen	forlegenheten	forleg
forlegenhetens	forlegen	forlegenhetens	forleg
firkantet	firkant	firkantet	firkant

One pass		Two passes	
firkantethet	firkantet	firkantethet	firkant
firkantetheten	firkantet	firkantetheten	firkant

Factory class
`solr.NorwegianLightStemFilterFactory`

Arguments
`variant`
Choose the Norwegian language variant to use. Valid values are:

- `nb`: Bokmål (default)
- `nn`: Nynorsk
- `no`: both

Example

```
<fieldType name="text_no" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words=
"lang/stopwords_no.txt" format="snowball"/>
    <filter class="solr.NorwegianLightStemFilterFactory"/>
  </analyzer>
</fieldType>
```

In
"Forelskelsen"

Tokenizer to Filter
"forelskelsen"

Out
"forelske"

Norwegian Minimal Stemmer

The `NorwegianMinimalStemFilterFactory` stems plural forms of Norwegian nouns only.

Factory class
`solr.NorwegianMinimalStemFilterFactory`

Arguments

`variant`
: Choose the Norwegian language variant to use. Valid values are:

  - `nb`: Bokmål (default)
  - `nn`: Nynorsk
  - `no`: both

Example

```
<fieldType name="text_no" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words=
"lang/stopwords_no.txt" format="snowball"/>
    <filter class="solr.NorwegianMinimalStemFilterFactory"/>
  </analyzer>
</fieldType>
```

In
: "Bilens"

Tokenizer to Filter
: "bilens"

Out
: "bil"

Persian

Persian Filter Factories

Solr includes support for normalizing Persian, and Lucene includes an example stopword list.

Factory class
: `solr.PersianNormalizationFilterFactory`

Arguments
: None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ArabicNormalizationFilterFactory"/>
  <filter class="solr.PersianNormalizationFilterFactory">
</analyzer>
```

Polish

Solr provides support for Polish stemming with the `solr.StempelPolishStemFilterFactory`, and `solr.MorphologikFilterFactory` for lemmatization, in the `contrib/analysis-extras` module. The `solr.StempelPolishStemFilterFactory` component includes an algorithmic stemmer with tables for Polish. To use either of these filters, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

Factory class

`solr.StempelPolishStemFilterFactory` and `solr.MorfologikFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.StempelPolishStemFilterFactory"/>
</analyzer>
```

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.MorfologikFilterFactory" dictionary-resource="pl"/>
</analyzer>
```

In

""studenta studenci"

Tokenizer to Filter

"studenta", "studenci"

Out

"student", "student"

More information about the Stempel stemmer is available in the Lucene javadocs.

The Morfologik `dictionary-resource` param value is a constant specifying which dictionary to choose. The dictionary resource must be named `morfologik/dictionaries/{dictionaryResource}.dict` and have an associated `.info` metadata file. See the Morfologik project for details.

Portuguese

Solr includes four stemmers for Portuguese: one in the `solr.SnowballPorterFilterFactory`, an alternative stemmer called `solr.PortugueseStemFilterFactory`, a lighter stemmer called `solr.PortugueseLightStemFilterFactory`, and an even less aggressive stemmer called `solr.PortugueseMinimalStemFilterFactory`. Lucene includes an example stopword list.

Factory classes

`solr.PortugueseStemFilterFactory`, `solr.PortugueseLightStemFilterFactory`, `solr.PortugueseMinimalStemFilterFactory`

Arguments

None

Examples

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.PortugueseStemFilterFactory"/>
</analyzer>
```

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.PortugueseLightStemFilterFactory"/>
</analyzer>
```

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.PortugueseMinimalStemFilterFactory"/>
</analyzer>
```

In

"praia praias"

Tokenizer to Filter

"praia", "praias"

Out

"pra", "pra"

Romanian

Solr can stem Romanian using the Snowball Porter Stemmer with an argument of `language="Romanian"`.

Factory class
`solr.SnowballPorterFilterFactory`

Arguments
`language`
(required) stemmer language, "Romanian" in this case

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Romanian" />
</analyzer>
```

Russian

Russian Stem Filter

Solr includes two stemmers for Russian: one in the `solr.SnowballPorterFilterFactory` `language="Russian"`, and a lighter stemmer called `solr.RussianLightStemFilterFactory`. Lucene includes an example stopword list.

Factory class
`solr.RussianLightStemFilterFactory`

Arguments
None

Use of custom charsets is no longer supported as of Solr 3.4. If you need to index text in these encodings, please use Java’s character set conversion facilities (`InputStreamReader`, and so on.) during I/O, so that Lucene can analyze this text as Unicode instead.

Example

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.RussianLightStemFilterFactory"/>
</analyzer>
```

Scandinavian

Scandinavian is a language group spanning three languages Norwegian, Swedish and Danish which are very similar.

Swedish å,ä,ö are in fact the same letters as Norwegian and Danish å,æ,ø and thus interchangeable when used between these languages. They are however folded differently when people type them on a keyboard lacking these characters.

In that situation almost all Swedish people use a, a, o instead of å, ä, ö. Norwegians and Danes on the other hand usually type aa, ae and oe instead of å, æ and ø. Some do however use a, a, o, oo, ao and sometimes permutations of everything above.

There are two filters for helping with normalization between Scandinavian languages: one is `solr.ScandinavianNormalizationFilterFactory` trying to preserve the special characters (æäöå) and another `solr.ScandinavianFoldingFilterFactory` which folds these to the more broad ø/ö→o etc.

See also each language section for other relevant filters.

Scandinavian Normalization Filter

This filter normalize use of the interchangeable Scandinavian characters æÆäÄöÖøØ and folded variants (aa, ao, ae, oe and oo) by transforming them to åÅæÆøØ.

It's a semantically less destructive solution than `ScandinavianFoldingFilter`, most useful when a person with a Norwegian or Danish keyboard queries a Swedish index and vice versa. This filter does **not** perform the common Swedish folds of å and ä to a nor ö to o.

Factory class
`solr.ScandinavianNormalizationFilterFactory`

Arguments
None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ScandinavianNormalizationFilterFactory"/>
</analyzer>
```

In

"blåbærsyltetøj blåbärsyltetöj blaabaarsyltetoej blabarsyltetoj"

Tokenizer to Filter

"blåbærsyltetøj", "blåbärsyltetöj", "blaabaersyltetoej", "blabarsyltetoj"

Out

"blåbærsyltetøj", "blåbærsyltetøj", "blåbærsyltetøj", "blabarsyltetoj"

Scandinavian Folding Filter

This filter folds Scandinavian characters åÅäæÄÆ→a and öÖøØ→o. It also discriminate against use of double vowels aa, ae, ao, oe and oo, leaving just the first one.

It's is a semantically more destructive solution than `ScandinavianNormalizationFilter`, but can in addition help with matching raksmorgas as räksmörgås.

Factory class
`solr.ScandinavianFoldingFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ScandinavianFoldingFilterFactory"/>
</analyzer>
```

In

"blåbærsyltetøj blåbärsyltetöj blaabaarsyltetoej blabarsyltetoj"

Tokenizer to Filter

"blåbærsyltetøj", "blåbärsyltetöj", "blaabaersyltetoej", "blabarsyltetoj"

Out

"blabarsyltetoj", "blabarsyltetoj", "blabarsyltetoj", "blabarsyltetoj"

Serbian

Serbian Normalization Filter

Solr includes a filter that normalizes Serbian Cyrillic and Latin characters to "bald" Latin. Cyrillic characters are first converted to Latin; then, Latin characters have their diacritics removed, with the exception of “đ” which is converted to “dj”. Note that this filter expects lowercased input.

Factory class

`solr.SerbianNormalizationFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SerbianNormalizationFilterFactory"/>
</analyzer>
```

In

"Đura (Ђура) српски"

Tokenizer to Filter

"đura" "ђура" "српски"

Out

"djura", "djura", "srpski"

Spanish

Solr includes two stemmers for Spanish: one in the `solr.SnowballPorterFilterFactory language="Spanish"`, and a lighter stemmer called `solr.SpanishLightStemFilterFactory`. Lucene includes an example stopword list.

Factory class

`solr.SpanishStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SpanishLightStemFilterFactory"/>
</analyzer>
```

In

"torear toreara torearlo"

Tokenizer to Filter

"torear", "toreara", "torearlo"

Out

"tor", "tor", "tor"

Swedish

Swedish Stem Filter

Solr includes two stemmers for Swedish: one in the `solr.SnowballPorterFilterFactory language="Swedish"`, and a lighter stemmer called `solr.SwedishLightStemFilterFactory`. Lucene includes an example stopword list.

Also relevant are the Scandinavian normalization filters.

Factory class
`solr.SwedishStemFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SwedishLightStemFilterFactory"/>
</analyzer>
```

In

"kloke klokhet klokheten"

Tokenizer to Filter

"kloke", "klokhet", "klokheten"

Out

"klok", "klok", "klok"

Thai

This filter converts sequences of Thai characters into individual Thai words. Unlike European languages, Thai does not use whitespace to delimit words.

Factory class
`solr.ThaiTokenizerFactory`

Arguments

None

Example

```
<analyzer type="index">
  <tokenizer class="solr.ThaiTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
```

Turkish

Solr includes support for stemming Turkish through the `solr.SnowballPorterFilterFactory`; support for case-insensitive search through the `solr.TurkishLowerCaseFilterFactory`; support for stripping apostrophes and following suffixes through `solr.ApostropheFilterFactory` (see Role of Apostrophes in Turkish Information Retrieval); support for a form of stemming that truncating tokens at a configurable maximum length through the solr.TruncateTokenFilterFactory (see Information Retrieval on Turkish Texts); and Lucene includes an example stopword list.

Factory class
`solr.TurkishLowerCaseFilterFactory`

Arguments

None

Example

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ApostropheFilterFactory"/>
  <filter class="solr.TurkishLowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Turkish"/>
</analyzer>
```

Another example, illustrating diacritics-insensitive search:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ApostropheFilterFactory"/>
  <filter class="solr.TurkishLowerCaseFilterFactory"/>
  <filter class="solr.ASCIIFoldingFilterFactory" preserveOriginal="true"/>
  <filter class="solr.KeywordRepeatFilterFactory"/>
  <filter class="solr.TruncateTokenFilterFactory" prefixLength="5"/>
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer>
```

Phonetic Matching

Phonetic matching algorithms may be used to encode tokens so that two different spellings that are pronounced similarly will match.

For overviews of and comparisons between algorithms, see http://en.wikipedia.org/wiki/Phonetic_algorithm and http://ntz-develop.blogspot.com/2011/03/phonetic-algorithms.html

Algorithms discussed in this section:

Beider-Morse Phonetic Matching (BMPM)

To use this encoding in your analyzer, see Beider Morse Filter in the Filter Descriptions section.

Beider-Morse Phonetic Matching (BMPM) is a "soundalike" tool that lets you search using a new phonetic matching system. BMPM helps you search for personal names (or just surnames) in a Solr/Lucene index, and is far superior to the existing phonetic codecs, such as regular soundex, metaphone, caverphone, etc.

In general, phonetic matching lets you search a name list for names that are phonetically equivalent to the desired name. BMPM is similar to a soundex search in that an exact spelling is not required. Unlike soundex, it does not generate a large quantity of false hits.

From the spelling of the name, BMPM attempts to determine the language. It then applies phonetic rules for that particular language to transliterate the name into a phonetic alphabet. If it is not possible to determine the language with a fair degree of certainty, it uses generic phonetic instead. Finally, it applies language-independent rules regarding such things as voiced and unvoiced consonants and vowels to further insure the reliability of the matches.

For example, assume that the matches found when searching for Stephen in a database are "Stefan", "Steph", "Stephen", "Steve", "Steven", "Stove", and "Stuffin". "Stefan", "Stephen", and "Steven" are probably relevant, and are names that you want to see. "Stuffin", however, is probably not relevant. Also rejected were "Steph", "Steve", and "Stove". Of those, "Stove" is probably not one that we would have wanted. But "Steph" and "Steve" are possibly ones that you might be interested in.

For Solr, BMPM searching is available for the following languages:

- English
- French
- German
- Greek
- Hebrew written in Hebrew letters
- Hungarian

- Italian
- Polish
- Romanian
- Russian written in Cyrillic letters
- Russian transliterated into English letters
- Spanish
- Turkish

The name matching is also applicable to non-Jewish surnames from the countries in which those languages are spoken.

For more information, see here: http://stevemorse.org/phoneticinfo.htm and http://stevemorse.org/phonetics/bmpm.htm.

Daitch-Mokotoff Soundex

To use this encoding in your analyzer, see Daitch-Mokotoff Soundex Filter in the Filter Descriptions section.

The Daitch-Mokotoff Soundex algorithm is a refinement of the Russel and American Soundex algorithms, yielding greater accuracy in matching especially Slavic and Yiddish surnames with similar pronunciation but differences in spelling.

The main differences compared to the other soundex variants are:

- coded names are 6 digits long
- initial character of the name is coded
- rules to encoded multi-character n-grams
- multiple possible encodings for the same name (branching)

Note: the implementation used by Solr (commons-codec's ` DaitchMokotoffSoundex `) has additional branching rules compared to the original description of the algorithm.

For more information, see http://en.wikipedia.org/wiki/Daitch%E2%80%93Mokotoff_Soundex and http://www.avotaynu.com/soundex.htm

Double Metaphone

To use this encoding in your analyzer, see Double Metaphone Filter in the Filter Descriptions section. Alternatively, you may specify `encoding="DoubleMetaphone"` with the Phonetic Filter, but note that the Phonetic Filter version will **not** provide the second ("alternate") encoding that is generated by the Double Metaphone Filter for some tokens.

Encodes tokens using the double metaphone algorithm by Lawrence Philips. See the original article at http://www.drdobbs.com/the-double-metaphone-search-algorithm/184401251?pgno=2

Metaphone

To use this encoding in your analyzer, specify `encoding="Metaphone"` with the Phonetic Filter.

Encodes tokens using the Metaphone algorithm by Lawrence Philips, described in "Hanging on the Metaphone" in Computer Language, Dec. 1990.

See http://en.wikipedia.org/wiki/Metaphone

Soundex

To use this encoding in your analyzer, specify `encoding="Soundex"` with the Phonetic Filter.

Encodes tokens using the Soundex algorithm, which is used to relate similar names, but can also be used as a general purpose scheme to find words with similar phonemes.

See http://en.wikipedia.org/wiki/Soundex

Refined Soundex

To use this encoding in your analyzer, specify `encoding="RefinedSoundex"` with the Phonetic Filter.

Encodes tokens using an improved version of the Soundex algorithm.

See http://en.wikipedia.org/wiki/Soundex

Caverphone

To use this encoding in your analyzer, specify `encoding="Caverphone"` with the Phonetic Filter.

Caverphone is an algorithm created by the Caversham Project at the University of Otago. The algorithm is optimised for accents present in the southern part of the city of Dunedin, New Zealand.

See http://en.wikipedia.org/wiki/Caverphone and the Caverphone 2.0 specification at http://caversham.otago.ac.nz/files/working/ctp150804.pdf

Kölner Phonetik a.k.a. Cologne Phonetic

To use this encoding in your analyzer, specify `encoding="ColognePhonetic"` with the Phonetic Filter.

The Kölner Phonetik, an algorithm published by Hans Joachim Postel in 1969, is optimized for the German language.

See http://de.wikipedia.org/wiki/K%C3%B6lner_Phonetik

NYSIIS

To use this encoding in your analyzer, specify `encoding="Nysiis"` with the Phonetic Filter.

NYSIIS is an encoding used to relate similar names, but can also be used as a general purpose scheme to find words with similar phonemes.

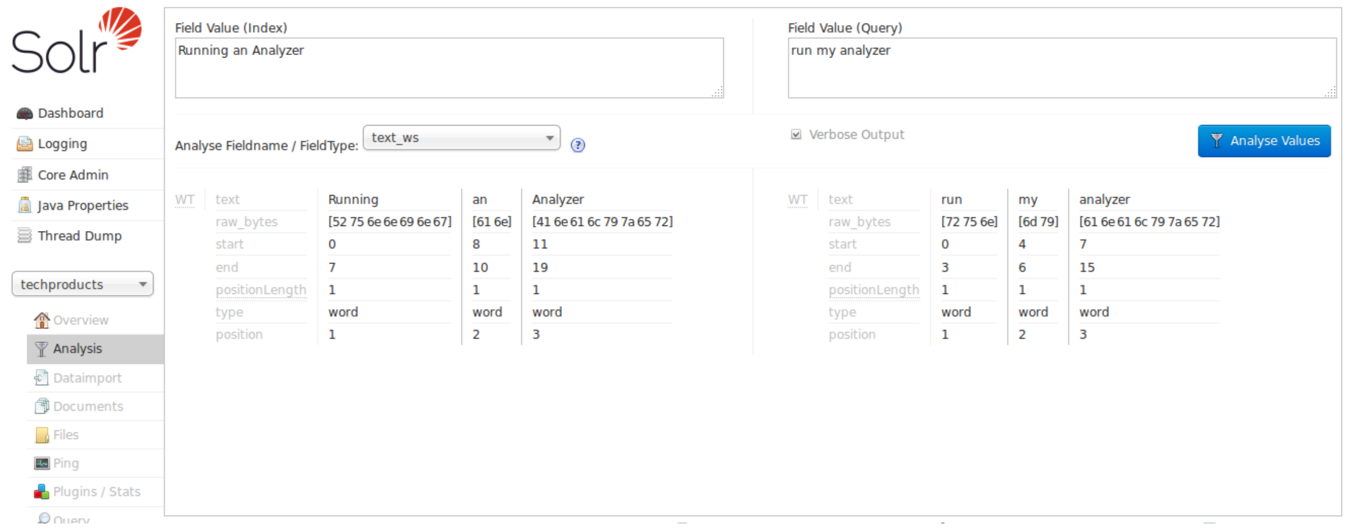
See http://en.wikipedia.org/wiki/NYSIIS and http://www.dropby.com/NYSIIS.html

Running Your Analyzer

Once you’ve defined a field type in `schema.xml` and specified the analysis steps that you want applied to it, you should test it out to make sure that it behaves the way you expect it to. Luckily, there is a very handy page in the Solr admin interface that lets you do just that. You can invoke the analyzer for any text field, provide sample input, and display the resulting token stream.

For example, let’s look at some of the "Text" field types available in the “bin/solr -e techproducts” example configuration, and use the Analysis Screen (http://localhost:8983/solr/#/techproducts/analysis) to compare how the tokens produced at index time for the sentence “Running an Analyzer” match up with a slightly different query text of “run my analyzers”.

We can begin with “text_ws” - one of the most simplified Text field types available:



Field Value (Index): Running an Analyzer

Field Value (Query): run my analyzer

Analyse Fieldname / FieldType: text_ws

Verbose Output ✓

WT	text	Running	an	Analyzer
	raw_bytes	[52 75 6e 6e 69 6e 67]	[61 6e]	[41 6e 61 6c 79 7a 65 72]
	start	0	8	11
	end	7	10	19
	positionLength	1	1	1
	type	word	word	word
	position	1	2	3

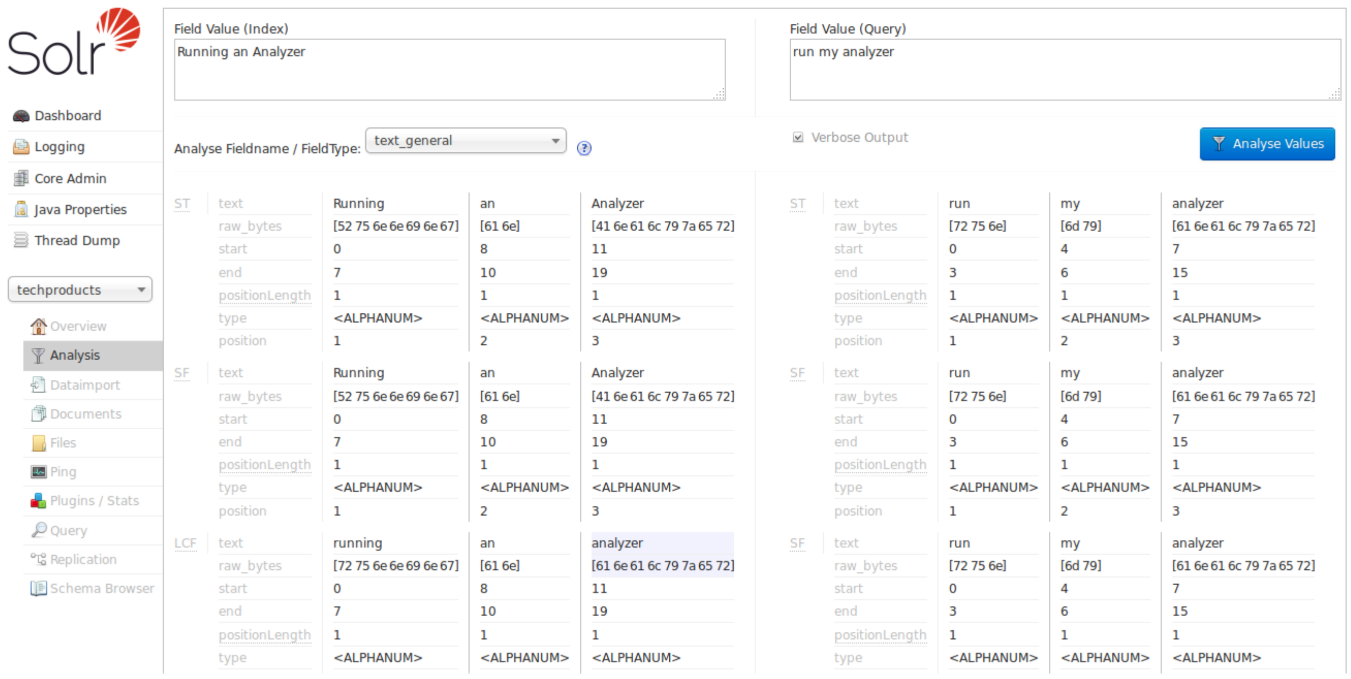
WT	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	word	word	word
	position	1	2	3

We can see very clearly that the only thing this field type does is tokenize text on whitespace. If our objective is to allow queries like “run my analyzer” to match indexed text like “Running an Analyzer” then we will evidently need to pick a different field type with index & query time text analysis that does more processing of the inputs.

In particular we will want:

- Case insensitivity, so "Analyzer" and "analyzer" match.
- Stemming, so words like "Run" and "Running" are considered equivalent terms.
- Stop Word Pruning, so small words like "an" and "my" don’t affect the query.

For our next attempt, let’s try the “text_general” field type:



Field Value (Index): Running an Analyzer

Field Value (Query): run my analyzer

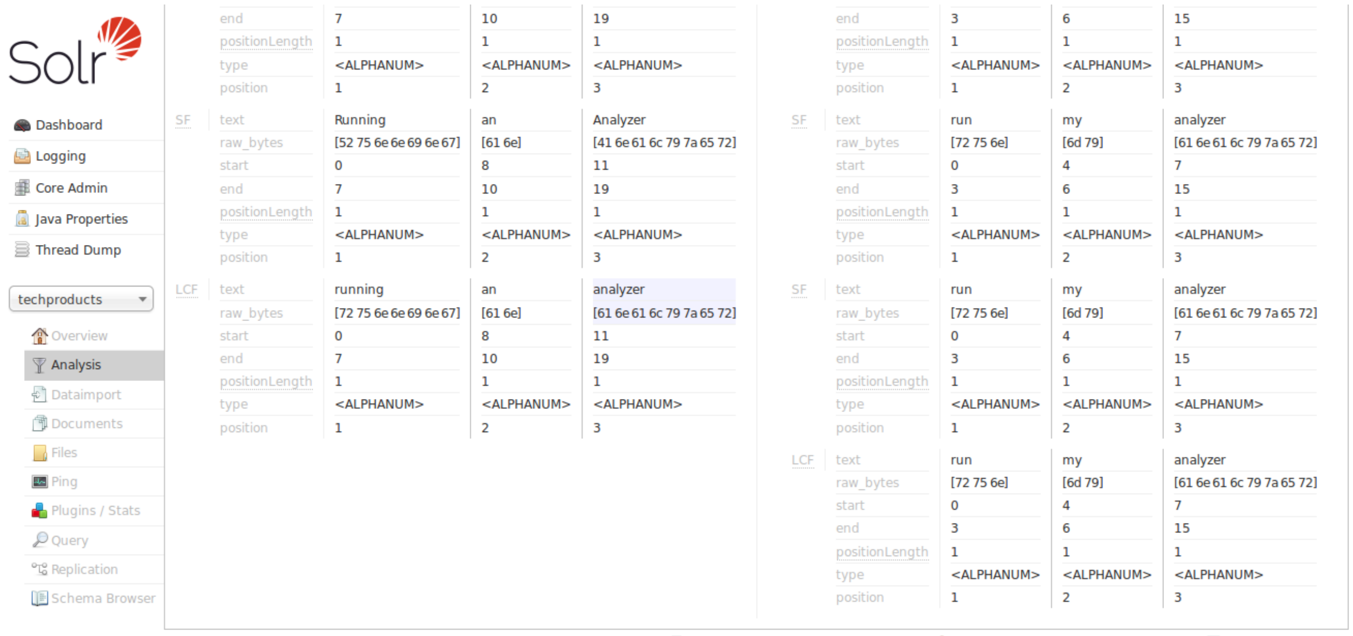
Analyse Fieldname / FieldType: text_general

☑ Verbose Output

ST	text	Running	an	Analyzer
	raw_bytes	[52 75 6e 6e 69 6e 67]	[61 6e]	[41 6e 61 6c 79 7a 65 72]
	start	0	8	11
	end	7	10	19
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
SF	text	Running	an	Analyzer
	raw_bytes	[52 75 6e 6e 69 6e 67]	[61 6e]	[41 6e 61 6c 79 7a 65 72]
	start	0	8	11
	end	7	10	19
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
LCF	text	running	an	analyzer
	raw_bytes	[72 75 6e 6e 69 6e 67]	[61 6e]	[61 6e 61 6c 79 7a 65 72]
	start	0	8	11
	end	7	10	19
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>

ST	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
SF	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
SF	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>

With the verbose output enabled, we can see how each stage of our new analyzers modify the tokens they receive before passing them on to the next stage. As we scroll down to the final output, we can see that we do start to get a match on "analyzer" from each input string, thanks to the "LCF" stage — which if you hover over with your mouse, you'll see is the “LowerCaseFilter”:



	end	7	10	19
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
SF	text	Running	an	Analyzer
	raw_bytes	[52 75 6e 6e 69 6e 67]	[61 6e]	[41 6e 61 6c 79 7a 65 72]
	start	0	8	11
	end	7	10	19
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
LCF	text	running	an	analyzer
	raw_bytes	[72 75 6e 6e 69 6e 67]	[61 6e]	[61 6e 61 6c 79 7a 65 72]
	start	0	8	11
	end	7	10	19
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3

	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
SF	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
SF	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3
LCF	text	run	my	analyzer
	raw_bytes	[72 75 6e]	[6d 79]	[61 6e 61 6c 79 7a 65 72]
	start	0	4	7
	end	3	6	15
	positionLength	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3

“text_general” is designed to be generally useful for any language, and it has definitely gotten us closer to our objective then “text_ws” by solving the problem of case sensitivity, by but it's still not quite what we are looking for. So now let us try the “text_en” field type:

Now we can see the "SF" (`StopFilter`) stage of the analyzers solving the problem of removing Stop Words, and as we scroll down, we also see the "PSF" (`PorterStemFilter`) stage apply stemming rules suitable for our English language input, such that the terms produced by our "index analyzer" and the terms produced by our "query analyzer" match the way we expect.

At this point, we can continue to experiment with additional inputs, verifying that our analyzers produce matching tokens when we expect them to match, and disparate tokens when we do not expect them to match, as we iterate and tweak our field type configuration.

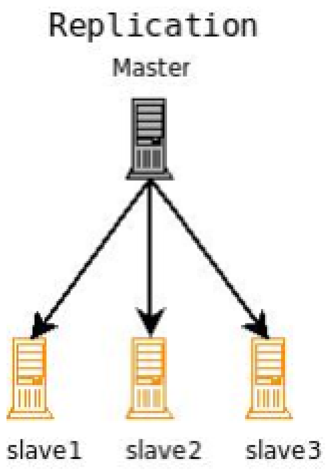
Scaling Solr

This section describes index replication in the master/slave architecture.

Index Replication

Index Replication distributes complete copies of a master index to one or more slave servers. The master server continues to manage updates to the index. All querying is handled by the slaves. This division of labor enables Solr to scale to provide adequate responsiveness to queries against large search volumes.

The figure below shows a Solr configuration using index replication. The master server's index is replicated on the slaves.



A Solr index can be replicated across multiple slave servers, which then processes requests.

Topics covered in this section:

- Index Replication in Solr
- Replication Terminology
- Configuring the ReplicationHandler
- Setting Up a Repeater with the ReplicationHandler
- Commit and Optimize Operations
- Slave Replication
- HTTP API Commands for the ReplicationHandler
- Distribution and Optimization

Index Replication in Solr

Solr includes a Java implementation of index replication that works over HTTP:

- The configuration affecting replication is controlled by a single file, `solrconfig.xml`
- Supports the replication of configuration files as well as index files
- Works across platforms with same configuration

- No reliance on OS-dependent file system features (eg: hard links)
- Tightly integrated with Solr; an admin page offers fine-grained control of each aspect of replication
- The Java-based replication feature is implemented as a request handler. Configuring replication is therefore similar to any normal request handler.

Replication In SolrCloud

Although there is no explicit concept of "master/slave" nodes in a SolrCloud cluster, the `ReplicationHandler` discussed on this page is still used by SolrCloud as needed to support "shard recovery" – but this is done in a peer to peer manner. When using SolrCloud, the `ReplicationHandler` must be available via the `/replication` path. Solr does this implicitly unless overridden explicitly in your `solrconfig.xml`, but If you wish to override the default behavior, make certain that you do not explicitly set any of the "master" or "slave" configuration options mentioned below, or they will interfere with normal SolrCloud operation.

Replication Terminology

The table below defines the key terms associated with Solr replication.

Term	Definition
Index	A Lucene index is a directory of files. These files make up the searchable and returnable data of a Solr Core.
Distribution	The copying of an index from the master server to all slaves. The distribution process takes advantage of Lucene’s index file structure.
Inserts and Deletes	As inserts and deletes occur in the index, the directory remains unchanged. Documents are always inserted into newly created files. Documents that are deleted are not removed from the files. They are flagged in the file, deletable, and are not removed from the files until the index is optimized.

Term	Definition
Master and Slave	A Solr replication master is a single node which receives all updates initially and keeps everything organized. Solr replication slave nodes receive no updates directly, instead all changes (such as inserts, updates, deletes, etc.) are made against the single master node. Changes made on the master are distributed to all the slave nodes which service all query requests from the clients.
Update	An update is a single change request against a single Solr instance. It may be a request to delete a document, add a new document, change a document, delete all documents matching a query, etc. Updates are handled synchronously within an individual Solr instance.
Optimization	A process that compacts the index and merges segments in order to improve query performance. Optimization should only be run on the master nodes. An optimized index may give query performance gains compared to an index that has become fragmented over a period of time with many updates. Distributing an optimized index requires a much longer time than the distribution of new segments to an un-optimized index.
Segments	A self contained subset of an index consisting of some documents and data structures related to the inverted index of terms in those documents.
mergeFactor	A parameter that controls the number of segments in an index. For example, when mergeFactor is set to 3, Solr will fill one segment with documents until the limit maxBufferedDocs is met, then it will start a new segment. When the number of segments specified by mergeFactor is reached (in this example, 3) then Solr will merge all the segments into a single index file, then begin writing new documents to a new segment.
Snapshot	A directory containing hard links to the data files of an index. Snapshots are distributed from the master nodes when the slaves pull them, "smart copying" any segments the slave node does not have in snapshot directory that contains the hard links to the most recent index data files.

Configuring the ReplicationHandler

In addition to `ReplicationHandler` configuration options specific to the master/slave roles, there are a few special configuration options that are generally supported (even when using SolrCloud).

- `maxNumberOfBackups` an integer value dictating the maximum number of backups this node will keep on disk as it receives `backup` commands.
- Similar to most other request handlers in Solr you may configure a set of "defaults, invariants, and/or appends" parameters corresponding with any request parameters supported by the `ReplicationHandler` when processing commands.

Configuring the Replication RequestHandler on a Master Server

Before running a replication, you should set the following parameters on initialization of the handler:

Name	Description
replicateAfter	String specifying action after which replication should occur. Valid values are commit, optimize, or startup. There can be multiple values for this parameter. If you use "startup", you need to have a "commit" and/or "optimize" entry also if you want to trigger replication on future commits or optimizes.
backupAfter	String specifying action after which a backup should occur. Valid values are commit, optimize, or startup. There can be multiple values for this parameter. It is not required for replication, it just makes a backup.
maxNumberOfBackups	Integer specifying how many backups to keep. This can be used to delete all but the most recent N backups.
confFiles	The configuration files to replicate, separated by a comma.
commitReserveDuration	If your commits are very frequent and your network is slow, you can tweak this parameter to increase the amount of time taken to download 5Mb from the master to a slave. The default is 10 seconds.

The example below shows a possible 'master' configuration for the `ReplicationHandler`, including a fixed number of backups and an invariant setting for the `maxWriteMBPerSec` request parameter to prevent slaves from saturating it's network interface

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">optimize</str>
    <str name="backupAfter">optimize</str>
    <str name="confFiles">schema.xml,stopwords.txt,elevate.xml</str>
    <str name="commitReserveDuration">00:00:10</str>
  </lst>
  <int name="maxNumberOfBackups">2</int>
  <lst name="invariants">
    <str name="maxWriteMBPerSec">16</str>
  </lst>
</requestHandler>
```

Replicating `solrconfig.xml`

In the configuration file on the master server, include a line like the following:

```
<str name="confFiles">solrconfig_slave.xml:solrconfig.xml,x.xml,y.xml</str>
```

This ensures that the local configuration `solrconfig_slave.xml` will be saved as `solrconfig.xml` on the slave. All other files will be saved with their original names.

On the master server, the file name of the slave configuration file can be anything, as long as the name is correctly identified in the `confFiles` string; then it will be saved as whatever file name appears after the colon, ":".

Configuring the Replication RequestHandler on a Slave Server

The code below shows how to configure a ReplicationHandler on a slave.

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="slave">

    <!-- fully qualified url for the replication handler of master. It is
         possible to pass on this as a request param for the fetchindex command -->
    <str name="masterUrl">http://remote_host:port/solr/core_name/replication</str>

    <!-- Interval in which the slave should poll master.  Format is HH:mm:ss .
         If this is absent slave does not poll automatically.

         But a fetchindex can be triggered from the admin or the http API -->

    <str name="pollInterval">00:00:20</str>

    <!-- THE FOLLOWING PARAMETERS ARE USUALLY NOT REQUIRED-->

    <!-- To use compression while transferring the index files. The possible
         values are internal|external.  If the value is 'external' make sure
         that your master Solr has the settings to honor the accept-encoding header.
         See here for details: http://wiki.apache.org/solr/SolrHttpCompression
         If it is 'internal' everything will be taken care of automatically.
         USE THIS ONLY IF YOUR BANDWIDTH IS LOW.
         THIS CAN ACTUALLY SLOWDOWN REPLICATION IN A LAN -->
    <str name="compression">internal</str>

    <!-- The following values are used when the slave connects to the master to
         download the index files.  Default values implicitly set as 5000ms and
         10000ms respectively. The user DOES NOT need to specify these unless the
         bandwidth is extremely low or if there is an extremely high latency -->

    <str name="httpConnTimeout">5000</str>
    <str name="httpReadTimeout">10000</str>

    <!-- If HTTP Basic authentication is enabled on the master, then the slave
         can be configured with the following -->

    <str name="httpBasicAuthUser">username</str>
    <str name="httpBasicAuthPassword">password</str>
  </lst>
</requestHandler>
```

Setting Up a Repeater with the ReplicationHandler

A master may be able to serve only so many slaves without affecting performance. Some organizations have deployed slave servers across multiple data centers. If each slave downloads the index from a remote data center, the resulting download may consume too much network bandwidth. To avoid performance degradation in cases like this, you can configure one or more slaves as repeaters. A repeater is simply a node that acts as both a master and a slave.

- To configure a server as a repeater, the definition of the Replication `requestHandler` in the `solrconfig.xml` file must include file lists of use for both masters and slaves.
- Be sure to set the `replicateAfter` parameter to commit, even if `replicateAfter` is set to optimize on the main master. This is because on a repeater (or any slave), a commit is called only after the index is downloaded. The optimize command is never called on slaves.
- Optionally, one can configure the repeater to fetch compressed files from the master through the compression parameter to reduce the index download time.

Here is an example of a ReplicationHandler configuration for a repeater:

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">commit</str>
    <str name="confFiles">schema.xml,stopwords.txt,synonyms.txt</str>
  </lst>
  <lst name="slave">
    <str name="masterUrl">
http://master.solr.company.com:8983/solr/core_name/replication</str>
    <str name="pollInterval">00:00:60</str>
  </lst>
</requestHandler>
```

Commit and Optimize Operations

When a commit or optimize operation is performed on the master, the RequestHandler reads the list of file names which are associated with each commit point. This relies on the `replicateAfter` parameter in the configuration to decide which types of events should trigger replication.

Setting on the Master	Description
commit	Triggers replication whenever a commit is performed on the master index.
optimize	Triggers replication whenever the master index is optimized.
startup	Triggers replication whenever the master index starts up.

The replicateAfter parameter can accept multiple arguments. For example:

```
<str name="replicateAfter">startup</str>
<str name="replicateAfter">commit</str>
<str name="replicateAfter">optimize</str>
```

Slave Replication

The master is totally unaware of the slaves. The slave continuously keeps polling the master (depending on the `pollInterval` parameter) to check the current index version of the master. If the slave finds out that the master has a newer version of the index it initiates a replication process. The steps are as follows:

- The slave issues a `filelist` command to get the list of the files. This command returns the names of the files as well as some metadata (for example, size, a lastmodified timestamp, an alias if any).
- The slave checks with its own index if it has any of those files in the local index. It then runs the filecontent command to download the missing files. This uses a custom format (akin to the HTTP chunked encoding) to download the full content or a part of each file. If the connection breaks in between, the download resumes from the point it failed. At any point, the slave tries 5 times before giving up a replication altogether.
- The files are downloaded into a temp directory, so that if either the slave or the master crashes during the download process, no files will be corrupted. Instead, the current replication will simply abort.
- After the download completes, all the new files are moved to the live index directory and the file's timestamp is same as its counterpart on the master.
- A commit command is issued on the slave by the Slave's ReplicationHandler and the new index is loaded.

Replicating Configuration Files

To replicate configuration files, list them using using the `confFiles` parameter. Only files found in the `conf` directory of the master's Solr instance will be replicated.

Solr replicates configuration files only when the index itself is replicated. That means even if a configuration file is changed on the master, that file will be replicated only after there is a new commit/optimize on master's index.

Unlike the index files, where the timestamp is good enough to figure out if they are identical, configuration files are compared against their checksum. The `schema.xml` files (on master and slave) are judged to be identical if their checksums are identical.

As a precaution when replicating configuration files, Solr copies configuration files to a temporary directory before moving them into their ultimate location in the conf directory. The old configuration files are then renamed and kept in the same `conf/` directory. The ReplicationHandler does not automatically clean up these old files.

If a replication involved downloading of at least one configuration file, the ReplicationHandler issues a core-reload command instead of a commit command.

Resolving Corruption Issues on Slave Servers

If documents are added to the slave, then the slave is no longer in sync with its master. However, the slave will not undertake any action to put itself in sync, until the master has new index data. When a commit operation takes place on the master, the index version of the master becomes different from that of the slave. The slave then fetches the list of files and finds that some of the files present on the master are also present in the local index but with different sizes and timestamps. This means that the master and slave have incompatible indexes. To correct this problem, the slave then copies all the index files from master to a new index directory and asks the core to load the fresh index from the new directory.

HTTP API Commands for the ReplicationHandler

You can use the HTTP commands below to control the ReplicationHandler's operations.

Command	Description
http://*master_host:port*/solr/*core_name*/replication?command=enablereplication	Enables replication on the master for all its slaves.
http://*master_host:port*/solr/*core_name*/replication?command=disablereplication	Disables replication on the master for all its slaves.
http://*host:port*/solr/*core_name*/replication?command=indexversion	Returns the version of the latest replicatable index on the specified master or slave.
http://*slave_host:port*/solr/*core_name*/replication?command=fetchindex	Forces the specified slave to fetch a copy of the index from its master. If you like, you can pass an extra attribute such as masterUrl or compression (or any other parameter which is specified in the `<lst name="slave">` tag) to do a one time replication from a master. This obviates the need for hard-coding the master in the slave.
http://*slave_host:port*/solr/*core_name*/replication?command=abortfetch	Aborts copying an index from a master to the specified slave.
http://*slave_host:port*/solr/*core_name*/replication?command=enablepoll	Enables the specified slave to poll for changes on the master.
http://*slave_host:port*/solr/*core_name*/replication?command=disablepoll	Disables the specified slave from polling for changes on the master.
http://*slave_host:port*/solr/*core_name*/replication?command=details	Retrieves configuration details and current status.

Command	Description
http://*host:port*/solr/*core_name*/replication?command=filelist&indexversion=<*index-version-number*>	Retrieves a list of Lucene files present in the specified host's index. You can discover the version number of the index by running the `indexversion` command.
http://*master_host:port*/solr/*core_name*/replication?command=backup	Creates a backup on master if there are committed index data in the server; otherwise, does nothing. This command is useful for making periodic backups. supported request parameters: * `numberToKeep:` request parameter can be used with the backup command unless the `maxNumberOfBackups` initialization parameter has been specified on the handler – in which case `maxNumberOfBackups` is always used and attempts to use the `numberToKeep` request parameter will cause an error. * `name` : (optional) Backup name . The snapshot will be created in a directory called snapshot.<name> within the data directory of the core . By default the name is generated using date in `yyyyMMddHHmmssSSS` format. If `location` parameter is passed , that would be used instead of the data directory * ` location ` : Backup location
http://*master_host:port* /solr/*core_name*/replication?command=deletebackup	Delete any backup created using the `backup` command . request parameters: * name: The name of the snapshot . A snapshot with the name snapshot.<name> must exist .If not, an error is thrown * location: Location where the snapshot is created

Distribution and Optimization

Optimizing an index is not something most users should generally worry about - but in particular users should be aware of the impacts of optimizing an index when using the `ReplicationHandler`.

The time required to optimize a master index can vary dramatically. A small index may be optimized in minutes. A very large index may take hours. The variables include the size of the index and the speed of the hardware.

Distributing a newly optimized index may take only a few minutes or up to an hour or more, again depending on the size of the index and the performance capabilities of network connections and disks. During optimization the machine is under load and does not process queries very well. Given a schedule of updates being driven a few times an hour to the slaves, we cannot run an optimize with every committed snapshot.

Copying an optimized index means that the **entire** index will need to be transferred during the next snappull. This is a large expense, but not nearly as huge as running the optimize everywhere. Consider this example: on a three-slave one-master configuration, distributing a newly-optimized index takes approximately 80 seconds *total*. Rolling the change across a tier would require approximately ten minutes per machine (or machine group). If this optimize were rolled across the query tier, and if each slave node being optimized were disabled and not receiving queries, a rollout would take at least twenty minutes and potentially as long as an hour and a half. Additionally, the files would need to be synchronized so that the *following* the optimize, snappull would not think that the independently optimized files were different in any way. This would also leave the door open to independent corruption of indexes instead of each being a perfect copy of the master.

Optimizing on the master allows for a straight-forward optimization operation. No query slaves need to be taken out of service. The optimized index can be distributed in the background as queries are being normally serviced. The optimization can occur at any time convenient to the application providing index updates.

While optimizing may have some benefits in some situations, a rapidly changing index will not retain those benefits for long, and since optimization is an intensive process, it may be better to consider other options, such as lowering the merge factor (discussed in the section on Index Configuration).