Lab 4. Mastering Text Analysis Methodologies

So far we have seen the installation of Solr server, schema design, documents, fields, field types, the Schema API and schemaless mode.

In this lab we will explore:

- Text analysis
- Analyzers
- Tokenizers
- Filters
- Multilingual analysis
- Phonetic matching Understanding text analysis

Nowadays, the search engine plays an important role in any search application. End users always expect accurate, efficient, and fast results from searches. The job of a search engine is to fulfill the search requirement in an easy and faster way. To achieve the expected level of search accuracy, Solr executes multiple processes sequentially behind the scenes: it examines the input string, normalizes the text, generates the token stream, builds indexes, and so on. The set of all of these processes is called text analysis. Let's explore text analysis in detail.

What is text analysis?

Text analysis is a Solr mechanism that takes place in two phases:

- During index time, optimize the input terms, feeding the information, generates the token stream and builds the indexes
- During query time, optimize the query terms, generates the token stream, matches with the term generated at index time, and provides results

Let's dive deeper and understand:

- How exactly Solr works to build indexes
- How to optimize the query terms to match with indexes
- How we get accurate, efficient, and fast results

If someone is searching for the string `The Host Country of Soccer World Cup 2018` and someone else is searching for the string `The Host Nation of Football world cup 2018`, the result should be `Russia` in both the cases. We will learn later in this lab how Solr matches a query containing `Nation` and `Football` to documents containing `Country` and `Soccer`.

We can't assume which type of search input comes from the end users during the search, for example:

- Searching for `Soccer` and `Football`
- Searching for `Unites States Of America` and `USA`
- Searching for `South Africa` and `RSA`
- Searching for `air-crew`, `aircrew`, and `air crew`

All of these are different input patters that contain ideally the same meaning in natural languages. But the user may provide input in a non-natural language also, like this:

- Searching for `Hundred GB` and `100 gigabyte`
- Searching for `Caffé` and `cafe`

There are also other complex search patterns that may be used by end users at query time. So, looking at the overall scope of possible search patterns used by end users during search time, Solr has to be ready to determine all possible search patterns, analyze them, and output them accurately with efficient results.

Here, however, we don't have to worry because [*The Solr*] is an intelligent search engine that handles all search input patterns being used across the world. So now, without thinking too much about Solr's searching capability, Let's see how Solr actually works to meet all our search requirements.

How text analysis works

We have seen an overview of Solr's text analysis; now Let's learn how Solr actually implements the analysis process. Here are the common steps normally used by Solr in an analysis process:

- **Removing stop words**:

  - Stop words (common letters/words) such as `a`, `an`, `the`, `at`, `to`, `for`, and so on are removed from the text string; so Solr will not give results for these common words. These words are configured in a text file (for example, `stopwords_en.txt`) and this file needs to be imported in the analysis configuration.

- **Adding synonyms**:

  - Solr reads synonyms from the text file (synonyms) and adds them to the token stream. All synonyms need to be preconfigured in the text file, such as `Football` and `Soccer`, `Country` and `Nation`, and so on.

- **Stemming the words**:

  - Solr transforms words into a base form using language-specific rules
  - It transforms **removing → remove** (removes **ing**)
  - It transforms **searches → search** (converts to singular)

- **Set all to lowercase**:

  - Solr converts all tokens to lowercase

These are the common steps that Solr performs in all normal scenarios. But in real life, things may not be that easy and straightforward. We can't assume which type of search input pattern in which language end users may use. To meet all possible requirements, Solr's intelligence is hidden in its three powerful tools:

- **Analyzer**: The task of the analyzer is to determine the text and generate the token stream accordingly

- **Tokenizer**: The tokenizer splits the input string at some delimiter and generates a token stream, say `Mastering Apache Solr` to `Mastering`, `Apache`, and `Solr` (splitting at white spaces)

- **Filter**: The filter performs one of these tasks:

  - **Adding**: Adds new tokens to the stream, such as adding synonyms

```
-  **Removing**: Removes tokens from the stream, such
   as stop words
-  **Conversation**: Converts tokens from one form to
   another, such as uppercase to lowercase
```

We will explore each one of these in detail later in this lab. By using these three tools, Solr becomes a powerful search engine to meet any complex search requirement.

Solr provides a UI Admin Console for us to understand the analysis process. Here, we can easily understand what is actually happening and which steps are getting executed during index and query time analysis. To access the admin console, navigate to http://localhost:8983/solr.

In the Solr admin console, the user can easily understand text analysis, querying, and so on:
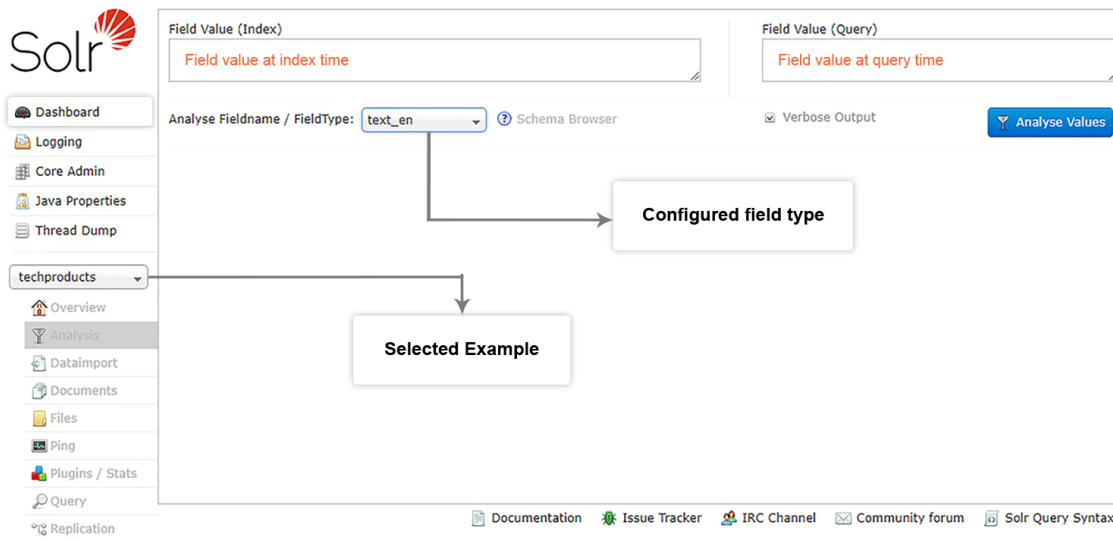
Go to **Dashboard | Core Selector** , select your configured example, and click on **Analysis** . Here we are using Solr's built-in example `techproducts` .

We have configured the `text_en` field as follows in the `managed-schema.xml` file:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer type="index">
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 <filter class="solr.LowerCaseFilterFactory"/>
 </analyzer>
 <analyzer type="query">
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
 <filter class="solr.LowerCaseFilterFactory"/>
 </analyzer>
 </fieldType>
```

Applying **lower case filter** (**LCF**) during index time and query .indexterm} time:

Text analysis applied during index time on the input string.

Input: `The Nation of soccer`

Output: `nation` , `soccer`

Analysis applied:

- It is split at white spaces
- Stop words (`The` , `of`) removed
- All set to lowercase

Text analysis applied during query time on the input string.

Input: `Famous country for football`

Output: `famous` , `nation` , `country` , `soccer` , and `football`

Analysis applied:

- Split at white spaces
- Stop words (`for`) removed
- Synonyms added (`nation` for `country` and `soccer` for `football`)
- All set to lowercase

As we know that analysis takes place in both phases (index time and query time), we have configured two `<analyzer>` elements distinguished by type attribute value (`index` for index time and `query` for query time). And we've configured a set of tokenizers and filters in each phase. The configurations for each phase may vary based on requirements. It is also possible to define a single `<analyzer>` element without the `type` attribute and configure tokenizers and filters inside that `<analyzer>` element. This type of config set will apply the same configurations to both phases. This is useful for cases where we want perfect string matching. We will discuss this later when we explore the analyzer in detail. Now consider the preceding analysis example.

During the index and query phases, the configured set of configurations is executed by the respective tokenizers and filters and the final token stream is generated. This stream (`nation` and `soccer`) is stored as an index. Now, during query time, the final token stream of the query phase (`famous` , `nation` , `country` , `soccer` , and `football`) will match the final token stream of the index phase (`nation` , `and` `soccer`), and we can see that there are multiple similar tokens (`nation` , `and` `soccer`) available in both streams. So here, both `The Nation of soccer` and `Famous country for football` search for the same results. This is the way how text

analysis process executing. Here we have just provided an overview of an analyzer, tokenizer, and filters. We will understand all of these analysis tools in detail later in this lab.

Understanding analyzer

We have seen an overview of text analysis. Now Let's dive deeper and understand the core processes running behind the scenes of analysis. As we have seen previously, the analyzer, tokenizer and filter are the three main components Solr uses for text analysis. Let's explore an analyzer.

What is an analyzer?

An analyzer examines the text of fields and generates a token stream. Normally, only fields of type `solr.TextField` will specify an analyzer. An analyzer is defined as a child element of the `<fieldType>` element in the `managed-schema.xml` file. Here is a simple analyzer configuration:
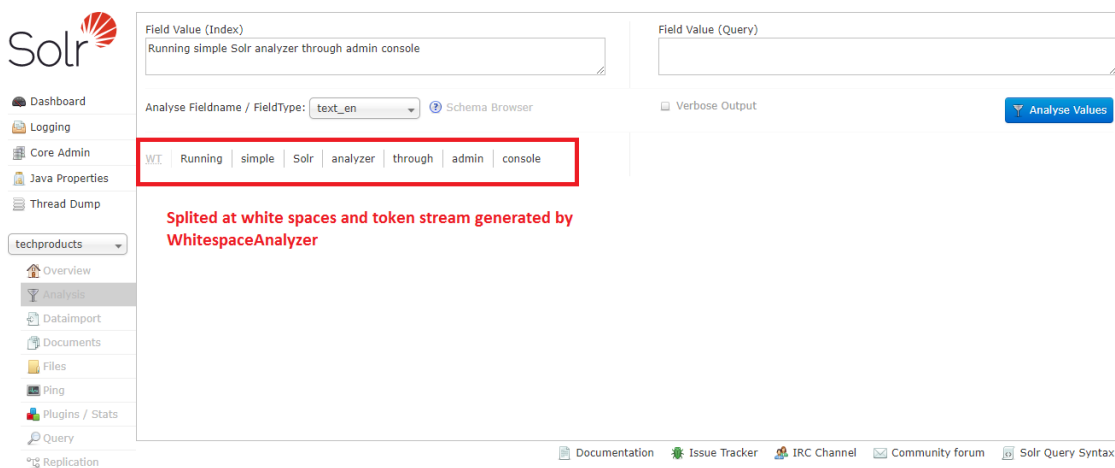
```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer class="org.apache.lucene.analysis.core.WhitespaceAnalyzer"/>
</fieldType>
```

Here, we have defined a single `<analyzer>` element. This is the simplest way to define an analyzer. We've already understood the `positionIncrementGap` attribute, which adds a space between multi-value fields, in the previous lab.

The class attribute value is a fully qualified Java class name. The input text will be analyzed by the analyzer class (`WhitespaceAnalyzer`). Let's configure the following analyzer configuration in `managed-schema.xml` and verify through the admin console:

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

Applying WhitespaceAnalyzer on the input string:

Input: `Running simple Solr analyzer through admin console`

Output: `Running` , `simple` , `Solr` , `analyzer` , `through` , `admin` , and `console`

This is a very simple example. The analyzer class `WhitespaceAnalyzer` splits the string at white spaces and generates the token stream. The named class must derive from `org.apache.lucene.analysis.Analyzer`.

The analyzer may be a single class or may be composed of a series of tokenizers and filter classes. Configuring an analyzer along with tokenizers and filters is very easy and straightforward. Define the `<analyzer>` element with child elements that name factory classes for the tokenizer and filters. Always configure tokenizers and filters in the order you want to run them in. Here is an example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 <filter class="solr.LowerCaseFilterFactory"/>
 </analyzer>
</fieldType>
```

Note

Solr will execute tokenizers and filters in the order in which they are configured. The execution order should be defined logically. For example, applying an LCF before a stop filter will impact the performance, as stop words are always going to be removed from the stream and still we would be unnecessarily applying an LCF.

The input text will be passed to the first element in the list (here it is `StandardTokenizerFactory`) and generate tokens accordingly. The output from this will be the input to the next (immediate successor; here it is `StopFilterFactory`). In this way, all the steps will be executed. The tokens generated from the last filter (`LowerCaseFilterFactory` here) will be the final token stream. Solr builds indexes using this stream.

How an analyzer works

Text analysis takes place in two phases, during index time and during query time. So we need to configure `<analyzer>` for both phases, distinguished by the type attribute. In the preceding example, we have configured a single `<analyzer>` element along with tokenizers and filters and not specified the `type` attribute. So Solr will apply the same configurations for the both phases (index and query). This type of configuration is required for some scenarios, say if we want to match strings exactly.

It is always advisable to define two separate `<analyzer>` for each phase distinguished by a `type` attribute. Doing so is required in some scenarios where we want to apply some steps at query time but not index time. For example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer type="index">
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 <filter class="solr.LowerCaseFilterFactory"/>
 </analyzer>
 <analyzer type="query">
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
```

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

Here, the synonyms filter (`SynonymGraphFilterFactory`) is injected at query time but not index time. If we inject this filter at index time, after adding a new synonym to the `synonyms.txt` file, we need to build document indexes again. Also the indexes for synonyms will be created and the index size will be increased.

At the time of configuring two separate `<analyzer>` for the index and query, we also need to bear in mind that configurations for both `<analyzer>` must be compatible with each other. For example, we have used the `LowerCaseFilterFactory` filter in both `<analyzer>` definitions in the preceding configurations. If we define an LCF only in the indexing phase and not in the querying phase, a query for `Soccer` will never match with the indexed term `soccer` .

So far we have learned the following:

- Solr performs text analysis in both phases: index and query time.
- Solr uses analyzers, tokenizers, and filters for text analysis.
- Using the analyzers, tokenizers, and filters, Solr examines the input string during index time. It normalizes accordingly and generates the token stream. Solr builds indexes based on this token stream.
- Using the same analyzer, tokenizers, and filters during query time, Solr examines the query string, normalizes accordingly, and generates a token stream. This token stream will be compared with the token stream generated at index time and return the matching output.
- The configuration of analyzer depends on the search requirements.
- Defining a single analyzer will be considered as the same analysis configuration for both phases.
- The advisable approach is always to define two separate analyzer for each phase so that we can apply friendlier configurations for each phase.
- When defining the tokenizers and filter, the ordering should be logical.
- Both the phase configurations should be compatible with each other.

So now we can apply text analysis on slightly more complex search strings. Let's apply more tokenizers and filters and understand their behavior. Previously, we mentioned an example of searching `The Host Country of Soccer World Cup 2018` and searching for `The Host Nation of Football world cup 2018` ; in both cases, the result should be `Russia` . Let's see how Solr analyzes this example. Up next are the analyzer configurations configured in the `managed-schema.xml` file. There are two separate `<analyzer>` elements for index time and query time, distinguished by the `type` attribute:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
    <analyzer type="index">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
      <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
    <analyzer type="query">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
      <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
      <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
  </fieldType>
```

Text analysis by configuring various tokenizers and filters during index time and query time:

In the preceding screen, we have provided the string `The Host Country of Soccer World Cup 2018` at index time and `The Host Nation of Football world cup 2018` at query time.

An analyzer during index time.

String: `The Host Country of Soccer World Cup 2018`

Solr executes all tokenizers and filters sequentially configured inside the `<analyzer type="index">` definition and generates a token stream accordingly. It builds indexes using this token stream.

StandardTokenizerFactory (**ST**): Splits the string at white spaces:

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

Input: `The Host Country of Soccer World Cup 2018`

Output: `The`, `Host`, `Country`, `of`, `Soccer`, `World`, `Cup`, and `2018`

The analyzer passes the input text to the first element from the list. Here, it is `StandardTokenizerFactory`. The entire string was split at white spaces using standard tokenizer. Now the output of this tokenizer will be the input to the next (immediate successor) in the sequence chain; here, it is `StopFilterFactory`.

StopFilterFactory: Removes all stop words (common) listed in the `stopwords_en.txt` file:

```
<filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt"
/>
```

Input: `The`, `Host`, `Country`, `of`, `Soccer`, `World`, `Cup`, `2018` (output of immediate predecessor)

*Output: * `Host`, `Country`, `Soccer`, `World`, `Cup`, `2018`

Here all stop words (`The`, `of`, and so on) are removed from the steam. Stop words are nothing but all common words (`a`, `an`, `the`, `of`, `at`, `for`, `in`, and so on) listed in the `stopwords_en.txt` file. After the removal of these stop words, Solr will not give matching results for these words. Now the output of this filter will be the input to its immediate successor in the chain; here, it is `LowerCaseFilterFactory`.

The `attribute ignoreCase="true"` (default `false`) will ignore case when testing for stop words. If it is `true` , the stop list should contain lowercase words.

LowerCaseFilterFactory: Converts all tokens to lowercase:

```
<filter class="solr.LowerCaseFilterFactory"/>
```

Input: `Host` , `Country` , `Soccer` , `World` , `Cup` , `2018`

Output: `host` , `country` , `soccer` , `world` , `cup` , `X`

All the incoming inputs will be converted to lowercase:

Now all three components (`StandardTokenizerFactory` , `StopFilterFactory` , and `LowerCaseFilterFactory`) have executed their job sequentially and generated the final token stream:

- **Final token stream**: `host` , `country` , `soccer` , `world` , `cup` , `2018`

Next, Solr builds indexes based on this final token stream:

- **Analyzer**: Query time
- **String:** `The Host Nation of Football world cup 2018`

At query time, Solr executes all tokenizers and filters sequentially configured inside the `<analyzer type="query">` definition and generates a token stream accordingly. This token stream will be compared with the token stream generated during index time. The matching result will be given as output.

StandardTokenizerFactory: Splits the string at white spaces:

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

Input: `The Host Nation of Football world cup 2018`

*Output: *` The` , `Host` , `Nation` , `of` , `Football` , `world` , `cup` , `2018`

The analyzer passes the input text to the first element from the list. Here, it is `StandardTokenizerFactory` . The entire string was split at white spaces using Standard Tokenizer. The output of this tokenizer will be the input to the immediate successor in the chain; here, it is `StopFilterFactory` .

StopFilterFactory: Removes all stop words (common) listed in the `stopwords_en.txt` file.

```
<filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt"
/>
```

Input: `The` , `Host` , `Nation` , `of` , `Football` , `world` , `cup` , `2018` (output of immediate predecessor)

Output: `Host` , `Nation` , `Football` , `world` , `cup` , `2018`

Here all stop words are removed from the stream. The output of this filter will be the input to its immediate successor in the chain, `SynonymGraphFilterFactory` .

The `attribute ignoreCase="true"` (default: `false`) will ignore casing when testing for stop words. If it is `true` , the stop list should contain lowercase words.

SynonymGraphFilterFactory: Adds synonyms to the token stream:

```
<filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
```

Input: `Host`, `Nation`, `Football`, `world`, `cup`, `2018` **Output**:
`Host`, `country`, `Nation`, `soccer`, `Football`, `world`, `cup`, `2018`

The synonyms (`country` for `nation` and `soccer` for `Football`) are added to the stream. All synonyms are configured in the `synonyms.txt` file, as follows. We will see this filter in detail later in this lab.

`Synonyms.txt file` : Synonym mapping examples. Blank lines and lines that start with `#` will be ignored:

```
football,soccer
dumb,stupid,dull
country => nation
smart,clever,bright => intelligent,genius
```

Here, `SynonymGraphFilterFactory` is configured at query time but not at index time. Previously, we have seen the reason for this type of configuration variation for index time and query time.

The output will be passed to the next filter, `LowerCaseFilterFactory` .

LowerCaseFilterFactory: Converts all tokens to lowercase:

```
<filter class="solr.LowerCaseFilterFactory"/>
```

Input: `Host`, `country`, `Nation`, `soccer`, `Football`, `world`, `cup`, `2018`

Output: `host`, `country`, `nation`, `soccer`, `football`, `world`, `cup`, `2018`

All the incoming inputs will be converted to lowercase.

Solr has examined the query string and produced the final token stream using four components (`StandardTokenizerFactory`, `StopFilterFactory`, `SynonymGraphFilterFactory`, and `LowerCaseFilterFactory`).

Now, these are the final streams of both phases:

- **Indexed stream**: `host`, `country`, `soccer`, `world`, `cup`, `2018`
- **Query **stream****: `host`, `country`, `nation`, `soccer`, `football`, `world`, `cup`, `2018`

We can see that many tokens from the query stream are matching tokens from the indexed stream, such as `country` and `soccer` . So in this way, Solr will bring the same output for the search string `The Host Country of Soccer World Cup 2018` and `The Host Nation of Football world cup 2018` .

Thus, we have covered:

- The Solr text analysis mechanism
- The tasks of analyzer, tokenizer, and filters
- Defining a single `<analyzer>` or multiple `<analyzer>` distinguished by the type attribute based on search requirements
- Defining configurations steps logically such as using a LCF after a stop filter and adding a synonym filter at query time only (we explained both the examples previously)

Now we are familiar with the Solr text analysis mechanism. Here we have tried to explain it by taking a little complex string example, but we can't assume which type of input may be entered during searches by end users. Providing accurate results for any pattern of input string is the main aim of any search engine. Solr comes with a number of

tokenizers and filters to challenge any input pattern. Solr is also efficient at multiple language search. Here we have covered very few and common tokenizers and filters, but Solr's list for tokenizers and filters does not end there. There are many tokenizers and filters available. Let's understand the behavior of tokenizers in the next section.

Understanding tokenizers

We have previously seen that an analyzer may be a single class or a set of defined tokenizer and filter classes.

The analyzer executes the analysis process in two steps:

- **Tokenization (parsing)**: Using configured tokenizer classes
- **Filtering (transformation)**: Using configured filter classes

We can also do preprocessing on a character stream before tokenization; we can do this with the help of `CharFilters` (we will see this later in the lab). An analyzer knows its configured field, but a tokenizer doesn't have any idea about the field. The job of the tokenizer is only to read from a character stream, apply a tokenization mechanism based on its behavior, and produce a new sequence of a token stream.

What is a tokenizer?

A tokenizer is a tool provided by Solr that runs a tokenization process, breaks a stream of text into tokens at some delimiter, and generates a token stream. Tokenizers are configured by their Java implementation factory class in the `managed-schema.xml` file. For example, we can define a standard tokenizer as:

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

Most tokenizer implementation classes do not provide a default no-arg constructor. So, always use a tokenizer factory class instead of a tokenizer class. Solr provides a standard way to define tokenizers in XML format using factory implementation classes. These factory classes translate XML configurations to create an instance of the respective tokenizer implementation class. An analyzer may contain only tokenizers or both tokenizers and filters. If only tokenizers are configured, the output produced from tokenization is ready to use; otherwise, the output will be passed to the first filter in the list.

After tokenization, a new token stream is generated. The newly generated token, along with its normalized text values, also holds some metadata, such as the location at which they are generated. Token metadata is important for things like highlighting search results in the field text. A newly generated token contains a value that may be different from its input text value. We can't assume that the text of a token is the same as the input text, or that the length is the same. It's also possible for more than one token to have the same position or refer to the same set in the original text.

Available tokenizers in Solr

Solr provides a large number of tokenizers. Let's explore the behavior of some of them.

Standard tokenizer

This splits the text field into tokens, treating white space and punctuation as delimiters. It considers white spaces and punctuation (comma, dots, hyphens, semicolons, colons, hashtags, and @) as delimiters and discards all of them, with these exceptions:

- Dots that are not followed by white spaces are kept as part of the token. An example is internet domains such as `www.google.com` .
- Factory class--- `solr.StandardTokenizerFactory` .
- Arguments--- `maxTokenLength` (integer, default 255) The max length of token characters. Tokens that exceed the number of characters specified by `maxTokenLength` will be ignored.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 </analyzer>
</fieldType>
```

Input: `Please send a mail at dharmesh.vasoya@example.com by 12-11.`

Output: `Please` , `send` , `a` , `mail` , `at` , `dharmesh.vasoya` , `example.com` , `by` , `12` , `11`

A total of 10 tokens have been generated by the standard tokenizer. These will be passed to its immediate successor in the chain. Standard tokenizer supports Unicode standard annex UAX#29, http://unicode.org/reports/tr29/#Word_Boundaries word boundaries with the following token types: `<ALPHANUM>` , `<NUM>` , `<SOUTHEAST_ASIAN>` , `<IDEOGRAPHIC>` , and `<HIRAGANA>` .

White space tokenizer

This splits the text stream at white spaces only. However, it will not split the text at any punctuation (like the standard tokenizer). Therefore, all of the punctuation will remain as is inside the generated tokens.

Factory class: `solr.WhitespaceTokenizerFactory`

Arguments:

- `rule` ** **(default: `java`): A rule that considers white space as a delimiter.
- `java` : Uses `Character.isWhitespace(int)` (https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#isWhitespace-int-)
- `unicode` : Uses unicode's `WHITESPACE` property

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.WhitespaceTokenizerFactory" rule="java" />
 </analyzer>
</fieldType>
```

Input: `Please send a mail at dharmesh.vasoya@example.com by 12-11.`

Output: `Please` , `send` , `a` , `mail` , `at` , `dharmesh.vasoya@example.com` , `by` , `12-11.`

The input string was split at white spaces but the punctuation (`@` , `.` ,and `-`) was preserved in the tokens.

Classic tokenizer

This splits the text field into tokens at white spaces and punctuation. The classic tokenizer behaves in the same way as the standard tokenizer of Solr versions 3.1 and older. Like The standard tokenizer, it does not use the Unicode standard annex UAX#29 word boundary rules. Delimiter characters are discarded, with the following exceptions:

- Dots that are not followed by white spaces 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 hyphens are preserved
- It preserves internet domain names and email addresses as a single token

Factory class: `solr.ClassicTokenizerFactory`

Arguments: `maxTokenLength` (integer, default 255): Max length of the token characters. Tokens that exceed the number of characters specified by `maxTokenLength` will be ignored.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
    <analyzer>
      <tokenizer class="solr.ClassicTokenizerFactory"/>
    </analyzer>
  </fieldType>
```

Input: `Please send a mail at dharmesh.vasoya@example.com by 12-Nov.`

Output: `Please`, `send`, `a`, `mail`, `at`, `dharmesh.vasoya@example.com`, `by`, `12-Nov`

The input string is split at white spaces and punctuation, but the email address `dharmesh.vasoya@example.com` and `12-Nov` are preserved as part of the token.

Keyword tokenizer

This treats the entire text field as a single token. The kyword tokenizer is required in scenarios where we want to match the entire string as it is:

Factory class: `solr.KeywordTokenizerFactory`

Arguments: None

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.KeywordTokenizerFactory"/>
 </analyzer>
</fieldType>
```

*Input: * `Please send a mail at dharmesh.vasoya@example.com by 12-Nov.`

Output: `Please send a mail at dharmesh.vasoya@example.com by 12-Nov.`

The entire input string is preserved as a single token.

Lower case tokenizer

The lower case tokenizer considers white spaces and non-letters as delimiters, splits the input string at these delimiters, and then discards all delimiters. Finally, it converts all letters to lowercase.

Factory class: `solr.LowerCaseTokenizerFactory`

Arguments: None

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.LowerCaseTokenizerFactory"/>
 </analyzer>
</fieldType>
```

Input: `Please send a mail at dharmesh.vasoya@example.com by 12-Nov.`

Output: `please` , `send` , `a` , `mail` , `at` , `dharmesh` , `vasoya` , `example` , `com` , `by` , `nov`

The input string was first split at white spaces and punctuation and then converted to lowercase.

Letter tokenizer

The letter tokenizer discards all non-letter characters from the input string and then generates a token at strings of contiguous letters.

Factory class: `solr.LetterTokenizerFactory`

Arguments: None

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.LetterTokenizerFactory"/>
 </analyzer>
</fieldType>
```

Input: `I haven't received mail by Nov12Sunday`

Output: `I` , `haven` , `t` , `received` , `mail` , `by` , `Nov` , `Sunday`

All non-letter characters (`'` and `12`) are discarded first, and then tokens are generated by considering strings of contiguous letters.

N-gram tokenizer

This generates n-gram tokens of sizes in the provided range from the input string.

Factory class: `solr.NGramTokenizerFactory`

Arguments: `minGramSize (integer, default 1)` : The minimum n-gram size. `maxGramSize (integer, default 2)` : The maximum n-gram size

Note

The following condition must be fulfilled when providing gram-size arguments: `0 < minGramSize <= maxGramSize`

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.NGramTokenizerFactory" minGramSize="2" maxGramSize="3"/>
 </analyzer>
 </fieldType>
```

Input: `send me`

Output: `se` , `sen` , `en` , `end` , `nd` , `nd` , `d` , `dm` , `m` , `me` , `me`

N-gram tokenizer executes tokenization over the entire input string. Also, it does not consider white spaces as delimiters, so white space characters are also included in the tokenization. In the preceding example, white spaces are preserved as parts of the token after tokenization. The n-gram tokenizer is required in cases where we want to match search words from the start, end, or somewhere in between the string along with white spaces.

For example, the input string is `Please send me a mail at dharmesh.vasoya@example.com` and we want to match `mail at dharmesh.vasoya@example.com`.

Edge n-gram tokenizer

This generates n-gram tokens from the start over the entire input string. Like the n-gram tokenizer, the edge n-gram tokenizer also does not consider white space as a delimiter, so white space is also considered during tokenization.

Factory class: `solr.EdgeNGramTokenizerFactory`

Arguments:

- `minGramSize` (integer, default is `1`): The minimum n-gram size
- `maxGramSize` (integer, default is `1`): The maximum n-gram size (`0 < minGramSize <= maxGramSize`)

In earlier versions, Solr supported an argument side (`front` or `back` ; the default was `front`), which generated a token from the provided value. This argument has now been removed and Solr generates the token from the front end of the input string.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.EdgeNGramTokenizerFactory" minGramSize="2" maxGramSize="10"/>
 </analyzer>
 </fieldType>
```

Input: `send me`

Output: `se` , `sen` , `send` , `send ` , `send m` , `send me`

The entire input string is split into n-gram pattern tokens considering size parameters (`minGramSize` (2) and `maxGramSize` (10)) along with white spaces. The edge n-gram tokenizer is required for matching n-characters from the start of the string.

For example, the input string is `Please send me a mail at dharmesh.vasoya@example.com` and we want to match `Please send me a mail` but it will not match. Understanding filters

---

We have seen that the analyzer uses a series of tokenizer and filter classes together to transform the input string into a token string, which will be used by Solr in indexing. The job of the filter is different from the tokenizer. The tokenizer mostly splits the input string at some delimiters and generates a token stream. The filter transforms this stream into some other form and generates a new token stream. The input for a filter will be a token stream, not an input string, unlike what we were passing at the time of tokenization. The entire token stream generated through tokenization will be passed to the first filter class in the list. Let's cover filters in detail.

What is a filter?

A filter is a tool provided by Solr that runs a filtering process as follows:

- **Adding**: Adds a new token to the stream, such as adding synonyms

- **Removing**: Removes a token from the stream, such as stop words
- **Conversation**: Converts a token from one form to another form, say uppercase to lower case

When a token stream generated during tokenization is passed to the filter, normally the filter looks at each token sequentially and, as per their behavior, performs one of the preceding activities. It then produces a new token stream. Filters are also derived from `org.apache.lucene.analysis.TokenStream`, so the output of a filter is also a token stream.

We can define a filter by its Java implementation factory class in the `managed-schema.xml` file:

```
<filter class="solr.StandardFilterFactory"/>
```

A filter definition should follow a tokenizer or another filter definition because they take a token stream as input. We can configure filters as a child element of `<analyzer>` following the `<tokenizer>` elements. Here is a simple example configured in the `managed-schema.xml` file inside the field `text_en`:

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

The preceding example is very simple. Solr comes with a large number of filters to meet most of our search requirements. Let's understand them in detail.

Available filters in Solr

Solr provides the following filters. Let's explore their behavior.

Stop filter

This removes all the words listed inside the `stopwords.txt` file. Removing stop words will reduce the size of the index and improve performance. These are the standard English stop words provided by Solr:

[a], [an], [and], [are], [ask], [at], [be], [but], [by], [for], [if], [in], [into], [is], [it], [no], [not], [of], [on], [or], [such], [that], [the], [their], [then], [there], [these], [they], [this], [to], [was], [will], [with].

We can manage (add or remove) words from the file as per our requirement. We can also create a file for other languages and include it by mentioning the file path in the word argument:

Factory class: `solr.StopFilterFactory`

Arguments:

- `words` (optional): The path of the file that contains a list of stop words, one per line. Blank lines and lines that begin with `#` will be ignored from the file. The path may be an absolute or relative path.
- `format` (optional): Indicates the format of the stopword list, for example, `format="snowball"` for a stopwords list that has been formatted for snowball.
- `ignoreCase` (true/false, default false): This ignores casing when testing for stop words. If it is `true`, the stop list should contain lowercase words.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" words="lang/stopwords_en.txt" />
 </analyzer>
 </fieldType>
```

Input: `This is an example`

Tokenizer to filter: `This`, `is`, `an`, `example`

Output: `This`, `example`

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 </analyzer>
 </fieldType>
```

Input: `This is an example`

Tokenizer to filter: `This`, `is`, `an`, `example`

Output: `example`

In the first example, we have not specified the argument `ignoreCase`, but in the second example, we have set it to `true`; so the outputs from both the examples are different. The location of the file `stopwords_en.txt` is `%SOLR_HOME%/example/techproducts/solr/techproducts/conf/lang/stopwords_en.txt`, as we currently understand from Solr's built-in example `techproducts`.

LCF: Converts all uppercase letters to lowercase in the token

Factory class: `solr.LowerCaseFilterFactory`

Arguments: None

Example:

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

Input: `This is An example`

Tokenizer to Filter: `This`, `is`, `An`, `example`

Output: `this`, `is`, `an`, `example`

All uppercase letters from the tokens are converted to lowercase. The sequence order of `LowerCaseFilterFactory` in the filter chain should be significant. If we define LCF before stop filter, Solr will unnecessarily apply lower case filtering on those stop words that are going to be removed in the next step.

Note

If we need to use LCF in text analysis, then must apply LCF to both the phases of an analyzer (index and query). If we define LCF only in the indexing phase and not in the querying phase, a query for `Soccer` will never match with the indexed term `soccer`.

Classic filter

The classic filter is used with the classic tokenizer. It removes dots from acronyms and `'s` from possessives.

Factory class: `solr.ClassicFilterFactory`

Arguments: None

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.ClassicTokenizerFactory"/>
 <filter class="solr.ClassicFilterFactory"/>
 </analyzer>
 </fieldType>
```

Input: `Computer's C.P.U. isn't`

Tokenizer to filter: `Computer's` , `C.P.U.` , `isn't`

Output: `Computer` , `CPU` , `isn't`

The classic tokenizer and classic filter together converted `Computer's` to `Computer` by removing `'s` and reduced `C.P.U.` to `CPU` by removing the dots.

Synonym filter

** **During filtering, the synonym filter looks for synonymous words in the `synonyms.txt` file. The found synonyms will be added at the place of the original token. All the synonym mappings are configured inside the `synonyms.txt` file.

Factory class: `solr.SynonymFilterFactory`

Note

The synonym filter is now deprecated in Solr as it does not support multi-term synonym mapping. Solr provides a synonym graph filter as an alternative to the synonym filter with multi-term support.

Synonym graph filter

The synonym graph filter supports single- or multi-token synonyms. The filter maps single- or multi-token synonyms and generates a correct token, which was not supported by the synonym filter.

The synonym graph filter is normally configured at query time, not index time. This will reduce the size of the index. If this filter is configured at index time, after adding any new synonyms to the `synonyms.txt` file, re-indexing of entire documents is required. The synonym graph filter configuration at query time does not require re-indexing for

adding new synonyms to `synonyms.txt` . To configure this filter at index time, we must mention the flatten graph filter for treating tokens like the synonym filter.

Also, the configuration order for this filter is important. If we are configuring the synonym graph filter before the ASCII folding filter, then we need to maintain all diacritical words (like `caffé`) in `synonyms.txt` as well:

Factory class: `solr.SynonymGraphFilterFactory`

Arguments:

- `synonyms` (required): The path of a file (`synonyms.txt`) that contains a list of synonyms, one per line. Blank lines and lines that begin with `#` are ignored. This may be a comma-separated list of absolute paths, or paths relative to the Solr config directory.

Sample format of `synonyms.txt` :

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.

For example:

```
football,soccer
dumb,stupid,dull
```

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.

For example:

```
country => nation
smart,clever,bright => intelligent,genius
```

- `ignoreCase` (optional; default: `false`): This determines the behavior of the filter in case-sensitive or case insensitive matching from the file. If it is `true` , synonyms will be matched case insensitively.

- `expand` (optional; default: `true`): If this is set to `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. Supported formats are:

  - `solr` (`SolrSynonymParser`)
  - `wordnet` (`WordnetSynonymParser`)
  - We can pass the name of our own `SynonymMap.Builder` subclass.

- `tokenizerFactory` : The name of the tokenizer factory to use when parsing the synonyms file. 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 the analyzer is specified, then `tokenizerFactory` may not be, and vice versa.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
```

```
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true"/>
 </analyzer>
 </fieldType>
```

Input: `He is stupid, not clever`

Tokenizer to filter: `He` , `is` , `stupid` , `not` , `clever`

Output: `He` , `is` , `dumb` , `dull` , `stupid` , `not` , `intelligent` , `genius`

All the matching synonyms (from `synonyms.txt`) are added to the token stream.

If we want to apply the synonym graph filter at index time, we must define `FlattenGraphFilterFactory` in an analyzer definition index.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer type="index">
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true"/>
 <!-- required on index analyzers after synonym graph filters -->
 <filter class="solr.FlattenGraphFilterFactory"/>
 </analyzer>
 <analyzer type="query">
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true"/>
 </analyzer>
 </fieldType>
```

Input: `He is stupid, not clever`

Tokenizer to Filter: `He` , `is` , `stupid` , `not` , `clever`

Output: `He` , `is` , `dumb` , `dull` , `stupid` , `not` , `intelligent` , `genius`

All the matching synonyms (from `synonyms.txt`) are added to the token stream.

ASCII folding filter

You can transform alphabetic, numeric, and symbolic Unicode characters that are not in the Basic Latin Unicode block (the first 127 ASCII characters) into their ASCII equivalents, if one exists.

Factory class: `solr.ASCIIFoldingFilterFactory`

Arguments:

- `preserveOriginal` : A Boolean. The default is `false` . If `true` , the original token is preserved
  (`caffé` `-->` `caffé` , `cafe`).

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.WhitespaceTokenizerFactory"/>
 <filter class="solr.ASCIIFoldingFilterFactory" preserveOriginal="false" />
 </analyzer>
 </fieldType>
```

Input: `thé caffé`

Tokenizer to filter: `thé` , `caffé`

Output: `the` , `caffe`

Keep word filter

We keep only those tokens that are listed in the `keepwords.txt` files. This is the inverse of the stop words filter.

Factory class: `solr.KeepWordFilterFactory`

Arguments:

- `words` : Required. This is the 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.
- `ignoreCase` : `True` or `false` . The default is `false` . If it is `true` , then comparisons are done case insensitively and the word file is assumed to contain only lowercase words.

The following is the sample `keepwords.txt` file:

```
good
great
excellent
```

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.KeepWordFilterFactory" words="keepwords.txt" ignoreCase="true"/>
 </analyzer>
 </fieldType>
```

Input: `Good and excellent job`

Tokenizer to filter: `Good` , `and` , `excellent` , `job`

Output: `Good` , `excellent`

Here, we have set `ignoreCase="true"` to match the words from the `keepwords.txt` file case insensitively. The patch of `keepwords.txt` is `%SOLR_HOME%/example/techproducts/solr/techproducts/conf/keepwords.txt` . We can configure a file patch as relative or absolute as per our needs.

KStem filter

Solr provides a stemming mechanism though which words are converted to their base form by applying language-specific rules. For that, Solr provides a number of stemming filters. The KStem filter is an English-specific and less aggressive stemmer.

Factory class: `solr.KStemFilterFactory`

Arguments: None

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.KStemFilterFactory"/>
 </analyzer>
 </fieldType>
```

Input: `remove removing removed`

Tokenizer to filter: `remove`, `removing`, `removed`

Output: `remove`

From the input string, the KStem filter has transformed three forms of a word to their base form `remove`. The first word `remove` was kept as it is because it is already in its base form, the second word `removing` was transformed to `remove` by cropping `ing` and the third word `removed` was transformed to `remove` by cropping `d`. Solr provides a number of stemmer filters. Selecting these filters completely depends on their behavior and which language we are going to use them for. `PorterStemmer` is one of the popular stemmers. What if we do not want to modify some words by stemming? We configure `KeywordMarkerFilterFactory` before `KStemFilterFactory`. We will cover this soon.

KeywordMarkerFilterFactory

This discards the modification/stemming of words .indexterm} listed in the file `protwords.txt`. Any words in the protected word list will not be modified by any stemmer in Solr.

Arguments:

- `protected` : The path of the file that contains the protected word list, one per line

The following is the sample `protwords.txt` file:

```
removing
transforming
```

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.WhitespaceTokenizerFactory"/>
 <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt" />
 <filter class="solr.KStemFilterFactory"/>
 </analyzer>
 </fieldType>
```

Input: `remove removing removed transforming`

Tokenizer to filter: `remove`, `removing`, `removed`, `transforming`

Output: `remove`, `removing`, `remove`, `transforming`

Here we can see that the words `removing` and `transforming` are not stemmed by `KStemFilterFactory` because they are mentioned in the file `protwords.txt` and protected by `KeywordMarkerFilterFactory`.

Word delimiter graph filter

** **This filter splits tokens at word delimiters. This is an alternative to the word delimiter filter. Always use a word delimiter graph filter at query time and not at index time because the indexer can't directly consume a graph at index time; if you still need to use this filter at index time, use it with a flatten graph filter.

The rules for determining delimiters are as follows:

- A change in case within a word: `KnowMore` -> `Know`, `More`. This can be disabled by setting `splitOnCaseChange="0"`.
- A transition from alpha to numeric characters or vice versa: `Alpha1000` -> `Alpha`, `1000` `100MS` -> `100`, `MS`. This can be disabled by setting `splitOnNumerics="0"`.
- Non-alphanumeric characters are discarded: `air-crew` -> `air`, `crew`.
- A trailing `'s` is removed: `Solr's` -> `Solr`.
- Any leading or trailing delimiters are discarded: `-air-crew!!` -> `air`, `crew`.

Factory class: `solr.WordDelimiterGraphFilterFactory`

Arguments: It's not possible to list all the arguments here. Please refer to the Solr document for these.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer type="index">
 <tokenizer class="solr.WhitespaceTokenizerFactory"/>
 <filter class="solr.WordDelimiterGraphFilterFactory"/>
 <!-- required on index analyzers after graph filters -->
 <filter class="solr.FlattenGraphFilterFactory"/>
 </analyzer>
 <analyzer type="query">
 <tokenizer class="solr.WhitespaceTokenizerFactory"/>
 <filter class="solr.WordDelimiterGraphFilterFactory"/>
 </analyzer>
</fieldType>
```

Input: `KnowMore air-crew Alpha1000`

Tokenizer to filter: `KnowMore`, `air-crew`, `Alpha1000`

Output: `Know`, `More`, `air`, `crew`, `Alpha`, `1000`

This is a simple example of a word delimiter graph filter. However, we can play with this filter by applying much more complex filtering terms.

Understanding CharFilter

During an analysis, the analyzer passes the input string to the first tokenizer in the list. If we want to apply any preprocessing to the input string before passing to the tokenizer, we can do it through `CharFilter`. `CharFilters` can be chained like token filters and placed in front of a tokenizer to add, change, or remove characters from an input string. Here is a list of char filters provided by Solr:

- `solr.MappingCharFilterFactory`
- `solr.HTMLStripCharFilterFactory`
- `solr.ICUNormalizer2CharFilterFactory`
- `solr.PatternReplaceCharFilterFactor`

Understanding PatternReplaceCharFilterFactor

This filter uses regular expressions to replace .indexterm} 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

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <charFilter class="solr.PatternReplaceCharFilterFactory" pattern="(\w+)(ing)"
replacement="$1"/>
 <tokenizer class="solr.WhitespaceTokenizerFactory"/>
 </analyzer>
 </fieldType>
```

Input: `showing see-ing viewing`

Output: `show` , `see-ing` , `view`

As per the behavior of the pattern, `ing` is removed from the end of the words except `see-ing` .

Explaining every char filter is not possible here. Please refer to the Solr documents for these. Understanding multilingual analysis

---

So far, we have concentrated on Solr text analysis (analyzers, tokenizers, and filters) irrespective of any language. Solr support multiple language search and this feature puts Solr at the top of the list of search engines. Let's understand how Solr works for multiple language search.

So far all the examples we have covered are in English. The tokenization and filtering rules for English are very simple and straightforward, such as splitting at white spaces or any other delimiters, stemming, and so on. But once we start focusing on other languages, these rules may differ. Solr is already prepared to meet multiple analysis search requirements such as stemmers, synonyms filters, stop word filters, character query correction capabilities normalization, language identifiers, and so on. Some languages require their own tokenizers for complexity of parsing the language, some require their own stemming filters, and some require multiple filters as per the language characteristics. For example, here is an analyzer configured for Greek in `managed-schema.xml` :

```
<fieldType name="text_el" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <!-- greek specific lowercase for sigma -->
 <filter class="solr.GreekLowerCaseFilterFactory"/>
```

```
    <filter class="solr.StopFilterFactory" ignoreCase="false"
words="lang/stopwords_el.txt" />
    <filter class="solr.GreekStemFilterFactory"/>
    </analyzer>
    </fieldType>
```

Language identification

At the time of indexing, language identification is required to map text to language-specific fields. Solr uses the langid `UpdateRequestProcessor` for language identification. Two types of `UpdateRequestProcessor` are provided by Solr.

- `TikaLanguageIdentifierUpdateProcessor` : Uses the language identification libraries in Apache Tika
- `LangDetectLanguageIdentifierUpdateProcessor` : Uses the open source Language Detection Library for Java

Configuration of `TikaLanguageIdentifierUpdateProcessor` in `solrconfig.xml` :

```
<processor
class="org.apache.solr.update.processor.TikaLanguageIdentifierUpdateProcessorFactory">
    <lst name="defaults">
    <str name="langid.fl">title,subject,text,keywords</str>
    <str name="langid.langField">language_s</str>
    </lst>
</processor>
```

Configuration of `LangDetectLanguageIdentifierUpdateProcessor` in `solrconfig.xml` :

```
<processor
class="org.apache.solr.update.processor.LangDetectLanguageIdentifierUpdateProcessorFacto

    <lst name="defaults">
    <str name="langid.fl">title,subject,text,keywords</str>
    <str name="langid.langField">language_s</str>
    </lst>
</processor>
```

`UpdateRequestProcessor` provides many `langid` parameters. We are not going to explain them here.

Determining the language at index time is always preferable over query time. During query time, the input provided by the user may be short and sometimes not meaningful enough to extract language information. At index time, full documents are present, so language identification becomes easier.

Configuring Solr for multiple language search

There are mainly three approaches to configure Solr for multiple language search.

Creating separate fields per language

This is a very simple and straightforward approach, done by creating a separate field per language. Create a per language field in `managed-schema.xml` .

Example:

```
<!-- English -->
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
```

```
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_en.txt" />
 <filter class="solr.LowerCaseFilterFactory"/>
 </analyzer>
</fieldType>
<!-- Greek -->
<fieldType name="text_el" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <!-- greek specific lowercase for sigma -->
 <filter class="solr.GreekLowerCaseFilterFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="false"
words="lang/stopwords_el.txt" />
 <filter class="solr.GreekStemFilterFactory"/>
 </analyzer>
</fieldType>
<!-- Spanish -->
<fieldType name="text_es" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.LowerCaseFilterFactory"/>
 <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_es.txt" format="snowball" />
 <filter class="solr.SpanishLightStemFilterFactory"/>
 <!-- more aggressive: <filter class="solr.SnowballPorterFilterFactory"
language="Spanish"/> -->
 </analyzer>
</fieldType>
 ...
<field name="content_en" type="text_en" indexed="true" stored="true" />
<field name="content_el" type="text_el" indexed="true" stored="true" />
<field name="content_es" type="text_es" indexed="true" stored="true" />
```

For creating separate fields per language, the DisMax-style query parser is required, which makes it easy to search across multiple fields. Solr provides built-in support for this query parser. Creating separate fields per language is simple and straightforward and fulfills most of the multi-language search requirements. The approach has a disadvantage too. The index size may increase as we define separate fields per language, and so queries may slow down.

Creating separate indexes per language

In this approach, Solr creates a separate Solr index (Solr Core) per language. Solr supports the creation of multiple cores. Every core contains a unique Solr index. Every core uses separate configuration files, including the `managed-schema.xml` file. During searching, every Solr core searches its own data from its own configuration file. After the search, results from all cores are combined together and then returned as an output of the query. Here is the simple configuration of defining a core per language.

File `en_managed-schema.xml` :

```
<field name="content" type="text_en" indexed="true" stored="true" />
```

File `el_managed-schema.xml` :

```
<field name="content" type="text_el" indexed="true" stored="true" />
```

File `es_managed-schema.xml` :

```
<field name="content" type="text_es" indexed="true" stored="true" />
```

File `solr.xml` :

```
<cores>
 <core name="english" instanceDir="shared" dataDir="../cores/core-
perlanguage/data/english/" schema="en_managed-schema.xml" />
 <core name="greek" instanceDir="shared" dataDir="../cores/core-
perlanguage/data/greek/" schema="el_managed-schema.xml" />
 <core name="spanish" instanceDir="shared" dataDir="../cores/core-
perlanguage/data/spanish" schema="es_managed-schema.xml" />
 <core name="aggregator" instanceDir="shared" dataDir="data/aggregator" />
</cores>
```

During the query, a request is sent to each of the language-specific cores using the shards parameter. Now the search will be independent and parallel with other languages. This will improve the performance of Solr. Here the field "content" is defined differently in each language-specific core, the language analysis also executed as per the configuration of that core. The search performance is better by defining a core per language as the search executes in parallel across multiple smaller indexes. As opposed to this, searching across a growing number of fields in a much larger index (the language-per-field approach) will hurt the performance. However, managing each core per language is somehow difficult.

During multiple language search configuration, selecting the implementation approach completely depends on the search requirements. The separate fields per language approach suits cases where the index size is in control. The separate indexes per language approach suits cases where the former does not satisfy the requirements and we have sufficient environment maintenance capabilities.

Understanding phonetic matching

Phonetic matching algorithms are used to match different spellings that are pronounced similarly by encoding them. Some examples are `Sandeep` and `Sandip` ; `Taylor` , `Tailer` , and `Tailor` ; and so on. Solr provides several filters for phonetic matching.

Understanding Beider-Morse phonetic matching

Beider-Morse Phonetic Matching (**BMPM**) helps you search for personal names or surnames. It is a very intelligent algorithm compared to soundex, metaphone, caverphone, and so on. Its purpose is to match names that are phonetically equivalent to the expected name. BMPM does not split spellings and does not generate false hits. It extracts names that are phonetically equivalent.

It executes these steps to extract names that are phonetically equivalent:

- Determines the language from the spelling of the name
- Applies phonetic rules to identify the language and translates the name into phonetic alphabets
- In the case of a language not identified from the name, it applies generic phonetics
- Finally, it applies language-independent rules regarding things such as voiced and unvoiced consonants and vowels to further ensure the reliability of the matches

BMPM supports the following languages: English, French, German, Greek, Hebrew written in Hebrew script, Hungarian, Italian, Polish, Romanian, Russian written in Cyrillic script, Russian transliterated into Latin script, Spanish, and Turkish.

Factory class: `solr.BeiderMorseFilterFactory`

Arguments:

- `nameType` : Types of names. Valid values are `GENERIC` , `ASHKENAZI` , or `SEPHARDIC` . If you are not processing Ashkenazi or Sephardic names, use `GENERIC` .
- `ruleType` : The types of rules to apply. Valid values are `APPROX` or `EXACT` .
- `concat` : Defines whether 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 provided.

Example:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
 <analyzer>
 <tokenizer class="solr.StandardTokenizerFactory"/>
 <filter class="solr.BeiderMorseFilterFactory" nameType="GENERIC" ruleType="APPROX"
concat="true" languageSet="auto" />
 </analyzer>
 </fieldType>
```

Input: `sandeep`

Tokenizer to filter: `sandeep`

Output: `sYndip` , `sandDp` , `sandi` , `sandip` , `sondDp` , `sondi` , `sondip` , `zYndip` , `zandip` , `zondip`

From the generated tokens, token `sandip` is similar to our expectations.

Similar to BMPB, Solr provides many more algorithms with unique behavior for implementing phonetic matching. Following is the list of those algorithms:

- Daitch-Mokotoff soundex
- Double metaphone
- Metaphone
- Soundex
- Refined soundex
- Caverphone
- Kölner Phonetik also known as Cologne Phonetic
- NYSIIS

Explaining each algorithm is not possible, but we can understand their behavior through the Solr Admin console by configuring them in the `managed-schema.xml` file. Summary

---

In this lab, we saw an overview of text analysis, analyzers, tokenizers, filters, and how to configure an analyzer along with tokenizers and filters. We also saw the implementation approach for putting tokenizers and filters together. Then we moved on to multiple search. Here we explored how Solr determines a language, two approaches to creating separate fields and separate indexes per language for multiple-language search, and the pros and cons of each approach. Finally, we understood Solr phonetic matching mechanics using the BMPM algorithm.

In the next lab, we will see how to do indexing using client API, upload data using index handlers, upload data using Apache Tika with Solr Cell, and detect languages while indexing.