Community Experience Distilled

Leverage the power of Apache Solr to power up your business by navigating your users to their data quickly and efficiently

Apache Solr Search Patterns

Apache Solr is an open source search platform built on a Java library called Lucene. It serves as a search platform for many websites, as it has the capability of indexing and searching multiple websites to fetch desired results.

We begin with a brief introduction of analyzers and tokenizers to understand the challenges associated with implementing large-scale indexing and multilingual search functionality. We then move on to working with custom queries and understanding how filters work internally. While doing so, we also create our own query language or Solr plugin that does proximity searches. Furthermore, we discuss how Solr can be used for real-time analytics and tackle problems faced during its implementation in e-commerce search. We then dive deep into the spatial features such as indexing strategies and search/filtering strategies for a spatial search. We also do an in-depth analysis of problems faced in an ad serving platform and how Solr can be used to solve these problems.

Who this book is written for

This book is for developers who already know how to use Solr and are looking at procuring advanced strategies for improving their search using Solr. This book is also for people who work with analytics to generate graphs and reports using Solr. Moreover, if you are a search architect who is looking forward to scale your search using Solr, this is a must-have book for you.

It would be helpful if you are familiar with the Java programming language.

What you will learn from this book

- Customize the Solr scoring algorithm to get better and more relevant search results
- Use Solr with big data for analytical purposes
- Get insights into Solr internals—indexing and search
- Setting up and scaling with Solr cloud
- Implement spatial search with Solr
- Understand Finite State Transducers (FST) and implement text tagging using FST
- Breeze through the strategies used in executing search using Solr in e-commerce, advertising, and real estate websites
- Learn more about how to use Solr with AJAX

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Solr Indexing Internals'
- A synopsis of the book’s content
- More information on Apache Solr Search Patterns

About the Author

Jayant Kumar is an experienced software professional with a bachelor of engineering degree in computer science and more than 14 years of experience in architecting and developing large-scale web applications.

Jayant is an expert on search technologies and PHP and has been working with Lucene and Solr for more than 11 years now. He is the key person responsible for introducing Lucene as a search engine on www.naukri.com, the most successful job portal in India.

Jayant is also the author of the book Apache Solr PHP Integration, Packt Publishing, which has been very successful.

Jayant has played many different important roles throughout his career, including software developer, team leader, project manager, and architect, but his primary focus has been on building scalable solutions on the Web. Currently, he is associated with the digital division of HT Media as the chief architect responsible for the job site www.shine.com.

Jayant is an avid blogger and his blog can be visited at http://jayant7k.blogspot.in. His LinkedIn profile is available at http://www.linkedin.com/in/jayantkumar.
Apache Solr Search Patterns

Apache Solr is the most widely used full text search solution. Almost all the websites today use Solr to provide the search function. Development of the search feature with a basic Solr setup is the starting point. At a later stage, most developers find it imperative to delve into Solr to provide solutions to certain problems or add specific features. This book will provide a developer working on Solr with a deeper insight into Solr. The book will also provide strategies and concepts that are employed in the development of different solutions using Solr. You will not only learn how to tweak Solr, but will also understand how to use it to handle big data and solve scalability problems.

What This Book Covers

Chapter 1, Solr Indexing Internals, delves into how indexing happens in Solr and how analyzers and tokenizers work during index creation.

Chapter 2, Customizing the Solr Scoring Algorithm, discusses different scoring algorithms in Solr and how to tweak these algorithms and implement them in Solr.

Chapter 3, Solr Internals and Custom Queries, discusses in-depth how relevance calculation happens and how scorers and filters work internally in Solr. This chapter will outline how to create custom plugins in Solr.

Chapter 4, Solr for Big Data, focuses on churning out big data for analysis purposes, including various faceting concepts and tools that can be used with Solr in order to plot graphs and charts.

Chapter 5, Solr in E-commerce, discusses the problems faced during the implementation of Solr in an e-commerce website and the related strategies and solutions.

Chapter 6, Solr for Spatial Search, focuses on spatial capabilities that the current and previous Solr versions possess. This chapter will also cover important concepts such as indexing and searching or filtering strategies together with varied query types that are available with a spatial search.

Chapter 7, Using Solr in an Advertising System, discusses the problems faced during the implementation of Solr to search in an advertising system and the related strategies and solutions.
Chapter 8, *AJAX Solr*, focuses on an AJAX Solr feature that helps reduce dependency on the application. This chapter will also cover an in-depth understanding of AJAX Solr as a framework and its implementation.

Chapter 9, *SolrCloud*, provides the complete procedure to implement SolrCloud and examines the benefits of using a distributed search with SolrCloud.

Chapter 10, *Text Tagging with Lucene FST*, focuses on the basic understanding of an FST and its implementation and guides us in designing an algorithm for text tagging, which can be implemented using FSTs and further integrated with Solr.
Solr Indexing Internals

This chapter will walk us through the indexing process in Solr. We will discuss how input text is broken and how an index is created in Solr. Also, we will delve into the concept of analyzers and tokenizers and the part they play in the creation of an index. Second, we will look at multilingual search using Solr and discuss the concepts used for measuring the quality of an index. Third, we will look at the problems faced during indexing while working with large amounts of input data. Finally, we will discuss SolrCloud and the problems it solves. The following topics will be discussed throughout the chapter. We will discuss use cases for Solr in e-commerce and job sites. We will look at the problems faced while providing search in an e-commerce or job site:

- Solr indexing fundamentals
- Working of analyzers, tokenizers, and filters
- Handling a multilingual search
- Measuring the quality of search results
- Challenges faced in large-scale indexing
- Problems SolrCloud intends to solve
- The e-commerce problem statement
The job site problem statement – Solr indexing fundamentals

The index created by Solr is known as an inverted index. An inverted index contains statistics and information on terms in a document. This makes a term-based search very efficient. The index created by Solr can be used to list the documents that contain the searched term. For an example of an inverted index, we can look at the index at the back of any book, as this index is the most accurate example of an inverted index. We can see meaningful terms associated with pages on which they occur within the book. Similarly, in the case of an inverted index, the terms serve to point or refer to documents in which they occur.

Inverted index example

Let us study the Solr index in depth. A Solr index consists of documents, fields, and terms, and a document consists of strings or phrases known as terms. Terms that refer to the context can be grouped together in a field. For example, consider a product on any e-commerce site. Product information can be broadly divided into multiple fields such as product name, product description, product category, and product price. Fields can be either stored or indexed or both. A stored field contains the unanalyzed, original text related to the field. The text in indexed fields can be broken down into terms. The process of breaking text into terms is known as tokenization. The terms created after tokenization are called tokens, which are then used for creating the inverted index. The tokenization process employs a list of token filters that handle various aspects of the tokenization process. For example, the tokenizer breaks a sentence into words, and the filters work on converting all of those words to lowercase. There is a huge list of analyzers and tokenizers that can be used as required.
Let us look at a working example of the indexing process with two documents having only a single field. The following are the documents:

<table>
<thead>
<tr>
<th>Document Id</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harry Potter And The Half Blood Prince</td>
</tr>
<tr>
<td>2</td>
<td>Harry Potter And The Deathly Hallows</td>
</tr>
</tbody>
</table>

Documents with Document Id and content (Text)

Suppose we tell Solr that the tokenization or breaking of terms should happen on whitespace. Whitespace is defined as one or more spaces or tabs. The tokens formed after the tokenization of the preceding documents are as follows:

<table>
<thead>
<tr>
<th>Document Id</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harry Potter And The Half Blood Prince</td>
</tr>
<tr>
<td>2</td>
<td>Harry Potter And The Deathly Hallows</td>
</tr>
</tbody>
</table>

Tokens in both documents

The inverted index thus formed will contain the following terms and associations:

<table>
<thead>
<tr>
<th>Token</th>
<th>Document Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry</td>
<td>1, 2</td>
</tr>
<tr>
<td>Potter</td>
<td>1, 2</td>
</tr>
<tr>
<td>And</td>
<td>1, 2</td>
</tr>
<tr>
<td>The</td>
<td>1, 2</td>
</tr>
<tr>
<td>Half</td>
<td>1</td>
</tr>
<tr>
<td>Blood</td>
<td>1</td>
</tr>
<tr>
<td>Prince</td>
<td>1</td>
</tr>
<tr>
<td>Deathly</td>
<td>2</td>
</tr>
<tr>
<td>Hallows</td>
<td>2</td>
</tr>
</tbody>
</table>

Inverted index

In the index, we can see that the token Harry appears in both documents. If we search for Harry in the index we have created, the result will contain documents 1 and 2. On the other hand, the token Prince has only document 1 associated with it in the index. A search for Prince will return only document 1.
Let us look at how an index is stored in the filesystem. Refer to the following image:

![Index files on disk]

For the default installation of Solr, the index can be located in the `<Solr_directory>/example/solr/collection1/data`. We can see that the index consists of files starting with `_0` and `_1`. There are two segments* files and a `write.lock` file. An index is built up of sub-indexes known as segments. The `segments*` file contains information about the segments. In the present case, we have two segments namely `_0.*` and `_1.*`. Whenever new documents are added to the index, new segments are created or multiple segments are merged in the index. Any search for an index involves all the segments inside the index. Ideally, each segment is a fully independent index and can be searched separately.

Lucene keeps on merging these segments into one to reduce the number of segments it has to go through during a search. The merger is governed by `mergeFactor` and `mergePolicy`. The `mergeFactor` class controls how many segments a Lucene index is allowed to have before it is coalesced into one segment. When an update is made to an index, it is added to the most recently opened segment. When a segment fills up, more segments are created. If creating a new segment would cause the number of lowest-level segments to exceed the `mergeFactor` value, then all those segments are merged to form a single large segment. Choosing a `mergeFactor` value involves a trade-off between indexing and search. A low `mergeFactor` value indicates a small number of segments and a fast search. However, indexing is slow as more and more mergers continue to happen during indexing. On the other hand, maintaining a high value of `mergeFactor` speeds up indexing but slows down the search, since the number of segments to search increases. Nevertheless, documents can be pushed to newer segments on disk with fewer mergers. The default value of `mergeFactor` is 10. The `mergePolicy` class defines how segments are merged together. The default method is `TieredMergePolicy`, which merges segments of approximately equal sizes subject to an allowed number of segments per tier.
Let us look at the file extensions inside the index and understand their importance. We are working with Solr Version 4.8.1, which uses Lucene 4.8.1 at its core. The segment file names have **Lucene41** in them, but this string is not related to the version of Lucene being used.

The index structure is almost similar for Lucene 4.2 and later.

The file types in the index are as follows:

- **segments.gen**, **segments_N**: These files contain information about segments within an index. The **segments_N** file contains the active segments in an index as well as a generation number. The file with the largest generation number is considered to be active. The **segments.gen** file contains the current generation of the index.

- **.si**: The segment information file stores metadata about the segments. It contains information such as segment size (number of documents in the segment), whether the segment is a compound file or not, a checksum to check the integrity of the segment, and a list of files referred to by this segment.

- **write.lock**: This is a write lock file that is used to prevent multiple indexing processes from writing to the same index.

- **.fnm**: In our example, we can see the _0.fnm and _1.fnm files. These files contain information about fields for a particular segment of the index. The information stored here is represented by **FieldsCount**, **FieldName**, **FieldNumber**, and **FieldBits**. **FieldCount** is used to generate and store ordered number of fields in this index. If there are two fields in a document, **FieldsCount** will be 0 for the first field and 1 for the second field. **FieldName** is a string specifying the name as we have specified in our configuration. **FieldBits** are used to store information about the field such as whether the field is indexed or not, or whether term vectors, term positions, and term offsets are stored. We study these concepts in depth later in this chapter.

- **.fdx**: This file contains pointers that point a document to its field data. It is used for stored fields to find field-related data for a particular document from within the field data file (identified by the **.fdt** extension).

- **.fdt**: The field data file is used to store field-related data for each document. If you have a huge index with lots of stored fields, this will be the biggest file in the index. The **.fdt** and **.fdx** files are respectively used to store and retrieve fields for a particular document from the index.
The term dictionary file contains information related to all terms in an index. For each term, it contains per-term statistics, such as document frequency and pointers to the frequencies, skip data (the .doc file), position (the .pos file), and payload (the .pay file) for each term.

The term index file contains indexes to the term dictionary file. The .tip file is designed to be read entirely into memory to provide fast and random access to the term dictionary file.

The frequencies and skip data file consists of the list of documents that contain each term, along with the frequencies of the term in that document. If the length of the document list is greater than the allowed block size, the skip data to the beginning of the next block is also stored here.

The positions file contains the list of positions at which each term occurs within documents. In addition to terms and their positions, the file also contains part payloads and offsets for speedy retrieval.

The payload file contains payloads and offsets associated with certain term document positions. Payloads are byte arrays (strings or integers) stored with every term on a field. Payloads can be used for boosting certain terms over others.

The normalization files contain lengths and boost factors for documents and fields. This stores boost values that are multiplied into the score for hits on that field.

The per-document value files store additional scoring factors or other per-document information. This information is indexed by the document number and is intended to be loaded into main memory for fast access.

The term vector index file contains pointers and offsets to the .tvd (term vector document) file.

The term vector data file contains information about each document that has term vectors. It contains terms, frequencies, positions, offsets, and payloads for every document.

This file will be created only if some documents are deleted from the index. It contains information about what files were deleted from the index.

These files are used to create a compound index where all files belonging to a segment of the index are merged into a single .cfs file with a corresponding .cfe file indexing its subfiles. Compound indexes are used when there is a limitation on the system for the number of file descriptors the system can open during indexing. Since a compound file merges or collapses all segment files into a single file, the number of file descriptors to be used for indexing is small. However, this has a performance impact as additional processing is required to access each file within the compound file.
Ideally, when an index is created using Solr, the document to be indexed is broken down into tokens and then converted into an index by filling relevant information into the files we discussed earlier. We are now clear with the concept of tokens, fields, and documents. We also discussed payload. Term vectors, frequencies, positions, and offsets form the term vector component in Solr. The term vector component in Solr is used to store and return additional information about terms in a document. It is used for fast vector highlighting and some other features like "more like this" in Solr. Norms are used for calculating the score of a document during a search. It is a part of the scoring formula.

Now, let us look at how analyzers, tokenizers, and filters work in the conversion of the input text into a stream of tokens or terms for both indexing and searching purposes in Solr.

**Working of analyzers, tokenizers, and filters**

When a document is indexed, all fields within the document are subject to analysis. An analyzer examines the text within fields and converts them into token streams. It is used to pre-process the input text during indexing or search. Analyzers can be used independently or can consist of one tokenizer and zero or more filters. Tokenizers break the input text into tokens that are used for either indexing or search. Filters examine the token stream and can keep, discard, or convert them on the basis of certain rules. Tokenizers and filters are combined to form a pipeline or chain where the output from one tokenizer or filter acts as an input to another. Ideally, an analyzer is built up of a pipeline of tokenizers and filters and the output from the analyzer is used for indexing or search.

Let us see the example of a simple analyzer without any tokenizers and filters. This analyzer is specified in the schema.xml file in the Solr configuration with the help of the `<analyzer>` tag inside a `<fieldtype>` tag. Analyzers are always applied to fields of type `solr.TextField`. An analyzer must be a fully qualified Java class name derived from the Lucene analyzer `org.apache.lucene.analysis.Analyzer`. The following example shows a simple whitespace analyzer that breaks the input text by whitespace (space, tab, and new line) and creates tokens, which can then be used for both indexing and search:

```xml
<fieldType name="whitespace" class="solr.TextField">
    <analyzer class="org.apache.lucene.analysis.WhitespaceAnalyzer"/>
</fieldType>
```
A custom analyzer is one in which we specify a tokenizer and a pipeline of filters. We also have the option of specifying different analyzers for indexing and search operations on the same field. Ideally, we should use the same analyzer for indexing and search so that we search for the tokens that we created during indexing. However, there might be cases where we want the analysis to be different during indexing and search.

The job of a tokenizer is to break the input text into a stream of characters or strings, or phrases that are usually sub-sequences of the characters in the input text. An analyzer is aware of the field it is configured for, but a tokenizer is not. A tokenizer works on the character stream fed to it by the analyzer and outputs tokens. The tokenizer specified in schema.xml in the Solr configuration is an implementation of the tokenizer factory - org.apache.solr.analysis.TokenizerFactory.

A filter consumes input from a tokenizer or an analyzer and produces output in the form of tokens. The job of a filter is to look at each token passed to it and to pass, replace, or discard the token. The input to a filter is a token stream and the output is also a token stream. Thus, we can chain or pipeline one filter after another. Ideally, generic filtering is done first and then specific filters are applied.

An analyzer can have only one tokenizer. This is because the input to a tokenizer is a character stream and the output is tokens. Therefore, the output of a tokenizer cannot be used by another.

In addition to tokenizers and filters, an analyzer can contain a char filter. A char filter is another component that pre-processes input characters, namely adding, changing, or removing characters from the character stream. It consumes and produces a character stream and can thus be chained or pipelined.

Let us look at an example from the schema.xml file, which is shipped with the default Solr:

```xml
<fieldType name="text_general" class="solr.TextField"
positionIncrementGap="100">
  <analyzer type="index">
```

```
```
The field type specified here is named text_general and it is of type solr.TextField. We have specified a position increment gap of 100. That is, in a multivalued field, there would be a difference of 100 between the last token of one value and first token of the next value. A multivalued field has multiple values for the same field in a document. An example of a multivalued field is tags associated with a document. A document can have multiple tags and each tag is a value associated with the document. A search for any tag should return the documents associated with it. Let us see an example.

<table>
<thead>
<tr>
<th>Doc id</th>
<th>Text</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harry Potter And The Half Blood Prince</td>
<td>Harry potter series, book, half blood prince</td>
</tr>
<tr>
<td>2</td>
<td>Harry Potter And The Deathly Hallows</td>
<td>Book, harry potter series, deathly hallows</td>
</tr>
</tbody>
</table>

Example of multivalued field - documents with tags

Here each document has three tags. Suppose that the tags associated with a document are tokenized on comma. The tags will be multiple values within the index of each document. In this case, if the position increment gap is specified as 0 or not specified, a search for series book will return the first document. This is because the token series and book occur next to each other in the index. On the other hand, if a positionIncrementGap value of 100 is specified, there will be a difference of 100 positions between series and book and none of the documents will be returned in the result.
Solr Indexing Internals

In this example, we have multiple analyzers, one for indexing and another for search. The analyzer used for indexing consists of a `StandardTokenizer` class and two filters, `stop` and `lowercase`. The analyzer used for the search (query) consists of three filters, `stop`, `synonym`, and `lowercase` filters.

The standard tokenizer splits the input text into tokens, treating whitespace and punctuation as delimiters that are discarded. Dots not followed by whitespace are retained as part of the token, which in turn helps in retaining domain names. Words are split at hyphens (-) unless there is a number in the word. If there is a number in the word, it is preserved with hyphen. `@` is also treated as a delimiter, so e-mail addresses are not preserved.

The output of a standard tokenizer is a list of tokens that are passed to the stop filter and lowercase filter during indexing. The `stop` filter class contains a list of stop words that are discarded from the tokens received by it. The lowercase filter converts all tokens to lowercase. On the other hand, during a search, an additional filter known as `synonym filter` is applied. This filter replaces a token with its synonyms. The synonyms are mentioned in the `synonyms.txt` file specified as an attribute in the filter.

Let us make some modifications to the `stopwords.txt` and `synonyms.txt` files in our Solr configuration and see how the input text is analyzed.

Add the following two words, each in a new line in the `stopwords.txt` file:

```
and
the
```

Add the following in the `synonyms.txt` file:

```
King => Prince
```

We have now told Solr to treat `and` and `the` as stop words, so during analysis they would be dropped. During the search phrase, we map `King` to `Prince`, so a search for `king` will be replaced by a search for `prince`.

In order to view the results, perform the following steps:

- Open up your Solr interface, select a core (say collection1), and click on the `Analysis` link on the left-hand side.
- Enter the text of the first document in text box marked field value (index).
- Select the field name and field type value as `text`.
- Click on `Analyze values`. 
We can see the complete analysis phase during indexing. First, a standard tokenizer is applied that breaks the input text into tokens. Note that here *Half-Blood* was broken into *Half* and *Blood*. Next, we saw the stop filter removing the stop words we mentioned previously. The words *And* and *The* are discarded from the token stream. Finally, the lowercase filter converts all tokens to lowercase.

During the search, suppose the query entered is *Half-Blood* and *King*. To check how it is analyzed, enter the value in Field Value (Query), select the text value in the Field Name / Field Type, and click on Analyze values.

---

**Solr analysis for indexing**
We can see that during the search, as before, **Half-Blood** is tokenized as **Half** and **Blood**, **And** and is dropped in the stop filter phase. **King** is replaced with **prince** during the synonym filter phase. Finally, the lowercase filter converts all tokens to lowercase.

An important point to note over here is that the lowercase filter appears as the last filter. This is to prevent any mismatch between the text in the index and that in the search due to either of them having a capital letter in the token.

The Solr analysis feature can be used to analyze and check whether the analyzer we have created gives output in the desired format during indexing and search. It can also be used to debug if we find any cases where the results are not as expected.

What is the use of such complex analysis of text? Let us look at an example to understand a scenario where a result is expected from a search but none is found.

The following two documents are indexed in Solr with the custom analyzer we just discussed:

<table>
<thead>
<tr>
<th>Doc id</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project management</td>
</tr>
<tr>
<td>2</td>
<td>Project manager</td>
</tr>
</tbody>
</table>

After indexing, the index will have the following terms associated with the respective document ids:

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>1, 2</td>
</tr>
<tr>
<td>Management</td>
<td>1</td>
</tr>
<tr>
<td>Manager</td>
<td>2</td>
</tr>
</tbody>
</table>

A search for **project** will return both documents 1 and 2. However, a search for **manager** will return only document 2. Ideally, **manager** is equal to **management**. Therefore, a search for **manager** should also return both documents. This intelligence has to be built into Solr with the help of analyzers, tokenizers, and filters. In this case, a synonym filter mentioning **manager**, **management**, **manages** as synonyms should do the trick. Another way to handle the same scenario is to use **stemmers**. Stemmers reduce words into their stem, base, or root form. In this chase, the stem for all the preceding words will be **manage**. There is a huge list of analyzers, tokenizers, and filters available with Solr by default that should be able to satisfy any scenario we can think of.
For more information on analyzers, tokenizers, and filters, refer to: http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters

AND and OR queries are handled by respectively performing an intersection or union of documents returned from a search on all the terms of the query. Once the documents or hits are returned, a scorer calculates the relevance of each document in the result set on the basis of the inbuilt Term Frequency-Inverse Document Frequency (TF-IDF) scoring formula and returns the ranked results. Thus, a search for Project AND Manager will return only the 2nd document after the intersection of results that are available after searching both terms on the index.

It is important to remember that text processing during indexing and search affects the quality of results. Better results can be obtained by high-quality and well thought of text processing during indexing and search.

TF-IDF is a formula used to calculate the relevancy of search terms in a document against terms in existing documents. In a simple form, it favors a document that contains the term with high frequency and has lower occurrence in all the other documents.

In a simple form, a document with a high TF-IDF score contains the search term with high frequency, and the term itself does not appear as much in other documents.

More details on TF-IDF will be explained in Chapter 2, Customizing a Solr Scoring Algorithm.

Handling a multilingual search

Content is produced and consumed in native languages. Sometimes even normal-looking documents may contain more than one language. This makes language an important aspect for search. A user should be able to search in his or her language. Each language has its own set of characters. Some languages use characters to form words, while some use characters to form sentences. Some languages do not even have spaces between the characters forming sentences. Let us look at some examples to understand the complexities that Solr should handle during text analysis for different languages.

Suppose a document contains the following sentence in English:

Incorporating the world’s largest display screen on the slimmest of bodies the Xperia Z Ultra is Sony’s answer to all your recreational needs.
The question here is whether the words world’s and Sony’s should be indexed. If yes, then how? Should a search for Sony return this document in the result? What would be the stop words here—the words that do not need to be indexed? Ideally, we would like to ignore stop words such as the, on, of, is, all, or your. How should the document be indexed so that Xperia Z Ultra matches this document? First, we need to ensure that z is not a stop word. The search should contain the term xperia z ultra. This would break into +xperia OR z OR ultra. Here xperia is the only mandatory term. The results would be sorted in such a fashion that the document (our document) that contains all three terms will be at the top. Also, ideally we would like the search for world or sony to return this document in the result. In this case, we can use the LetterTokenizerFactory class, which will separate the words as follows:

- World’s => World, s
- Sony’s => Sony, s

Then, we need to pass the tokens through a stop filter to remove stop words. The output from the stop filter passes through a lowercase filter to convert all tokens to lowercase. During the search, we can use a WhiteSpaceTokenizer and a LowerCaseFilter tokenizer to tokenize and process our input text.

In a real-life situation, it is advisable to take multiple examples with different use cases and work around the scenarios to provide the desired solutions for those use cases. Given that the numbers of examples are large, the derived solution should satisfy most of the cases.

If we translate the same sentence into German, here is how it will look:

```
Die Einbeziehung der weltweit größten Display-Bildschirm auf der flächste der Körper das Xperia Z Ultra ist Sony's Antwort auf alle Ihre Freizeit-Bedarfnisse.
```

German

Solr comes with an inbuilt field type for German - text_de, which has a StandardTokenizer class followed by a lowercaseFilter class and a stopFilter class for German words. In addition, the analyzer has two German-specific filters, GermanNormalizationFilter and GermanLightStemFilter. Though this text analyzer does a pretty good job, there may be cases where it will need improvement.

Let’s translate the same sentence into Arabic and see how it looks:

```
دمج أكبر شاشة عرض في العالم على الأقل سمكة من الهياكل ارتكاب Z ألترا هو ألوان سوتي لجميع الاحتمالات الترفيهية الخاصة بك.
```

Arabic
Note that Arabic is written from right to left. The default analyzer in the Solr schema configuration is `text_ar`. Again tokenization is carried out with `StandardTokenizer` followed by `LowerCaseFilter` (used for non-Arabic words embedded inside the Arabic text) and the Arabic `StopFilter` class. This is followed by the Arabic Normalization filter and the Arabic Stemmer. Another aspect used in Arabic is known as a diacritic. A diacritic is a mark (also known as glyph) added to a letter to change the sound value of the letter. Diacritics generally appear either below or above a letter or, in some cases, between two letters or within the letter. Diacritics such as `‘` in English do not modify the meaning of the word. In contrast, in other languages, the addition of a diacritic modifies the meaning of the word. Arabic is such a language. Thus, it is important to decide whether to normalize diacritics or not.

Let us translate the same sentence into Japanese and see what we get:

```
のXperia Zはウルトラポディの最薄で、世界最大のディスプレイ画面を組み込むことは、すべてあなたのレクレーションのニーズにソニーの答えです。
```

Now that the complete sentence does not have any whitespace to separate the words, how do we identify words or tokens and index them? The Japanese analyzer available in our Solr schema configuration is `text_ja`. This analyzer identifies the words in the sentence and creates tokens. A few tokens identified are as follows:

<table>
<thead>
<tr>
<th>Token</th>
<th>Pronunciation (en)</th>
<th>Indexed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ポディ</td>
<td>hodi</td>
<td>Yes</td>
</tr>
<tr>
<td>の</td>
<td>no</td>
<td>No (stop word)</td>
</tr>
<tr>
<td>は</td>
<td>wa</td>
<td>No (stop word)</td>
</tr>
<tr>
<td>Xperia</td>
<td>xperia</td>
<td>Yes</td>
</tr>
<tr>
<td>ソニー</td>
<td>soni</td>
<td>Yes</td>
</tr>
</tbody>
</table>

It also identifies some of the stop words and removes them from the sentence.

As in English, there are other languages where a word is modified by adding a suffix or prefix to change the tense, grammatical mood, voice, aspect, person, number, or gender of the word. This concept is called inflection and is handled by stemmers during indexing. The purpose of a stemmer is to change words such as indexing, indexed, or indexes into their base form, namely index. The stemmer has to be introduced during both indexing and search so that the stems or roots are compared during both indexing and search.
Solr Indexing Internals

The point to note is that each language is unique and presents different challenges to the search engine. In order to create a language-aware search, the steps that need to be taken are as follows:

- **Identification of the language**: Decide whether the search would handle the dominant language in a document or find and handle multiple languages in the document.
- **Tokenization**: Decide the way tokens should be formed from the language.
- **Token processing**: Given a token, what processing should happen on the token to make it a part of the index? Should words be broken up or synonyms added? Should diacritics and grammars be normalized? A stop-word dictionary specific to the language needs to be applied.

Token processing can be done within Solr by using an appropriate analyzer, tokenizer, or filter. However, for this, all possibilities have to be thought through and certain rules need to be formed. The default analyzers can also be used, but it may not help in improving the relevance factor of the result set. Another way of handling a multilingual search is to process the document during indexing and before providing the data to Solr for indexing. This ensures more control on the way a document can be indexed.

The strategies used for handling a multilingual search with the same content across multiple languages at the Solr configuration level are:

- **Use one Solr field for each language**: This is a simple approach that guarantees that the text is processed the same way as it was indexed. As different fields can have separate analyzers, it is easy to handle multiple languages. However, this increases the complexity at query time as the input query language needs to be identified and the related language field needs to be queried. If all fields are queried, the query execution speed goes down. Also, this may require creation of multiple copies of the same text across fields for different languages.
- **Use one Solr core per language**: Each core has the same field with different analyzers, tokenizers, and filters specific to the language on that core. This does not have much query time performance overhead. However, there is significant complexity involved in managing multiple cores. This approach would prove complex in supporting multilingual documents across different cores.
• **All languages in one field:** Indexing and search are much easier as there is only a single field handling multiple languages. However, in this case, the analyzer, tokenizer, and filter have to be custom built to support the languages that are expected in the input text. The queries may not be processed in the same fashion as the index. Also, there might be confusion in the scoring calculation. There are cases where particular characters or words may be stop words in one language and meaningful in another language.

Custom analyzers are built as Solr plugins. The following link gives more details regarding the same: https://wiki.apache.org/solr/SolrPlugins#Analyzer.

The final aim of a multilingual search should be to provide better search results to the end users by proper processing of text both during indexing and at query time.

**Measuring the quality of search results**

Now that we know what analyzers are and how text analysis happens, we need to know whether the analysis that we have implemented provides better results. There are two concepts in the search result set that determine the quality of results, **precision** and **recall**:

• **Precision:** This is the fraction of retrieved documents that are relevant. A precision of 1.0 means that every result returned by the search was relevant, but there may be other relevant documents that were not a part of the search result.

\[
\text{precision} = \frac{|\{\text{relevant documents} \cap \{\text{retrieved documents}\}\}|}{|\{\text{retrieved documents}\}|}
\]

**Precision equation**

• **Recall:** This is the fraction of relevant documents that are retrieved. A recall of 1.0 means that all relevant documents were retrieved by the search irrespective of the irrelevant documents included in the result set.

\[
\text{recall} = \frac{|\{\text{relevant documents} \cap \{\text{retrieved documents}\}\}|}{|\{\text{relevant documents}\}|}
\]

**Recall equation**
Another way to define precision and recall is by classifying the documents into four classes between relevancy and retrieval as follows:

<table>
<thead>
<tr>
<th></th>
<th>Relevant</th>
<th>Irrelevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieved</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Not retrieved</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

We can define the formula for precision and recall as follows:

\[
\text{Precision} = \frac{A}{A \cup B} \\
\text{Recall} = \frac{A}{A \cup C}
\]

We can see that as the number of irrelevant documents or $B$ increases in the result set, the precision goes down. If all documents are retrieved, then the recall is perfect but the precision would not be good. On the other hand, if the document set contains only a single relevant document and that relevant document is retrieved in the search, then the precision is perfect but again the result set is not good. This is a trade-off between precision and recall as they are inversely related. As precision increases, recall decreases and vice versa. We can increase recall by retrieving more documents, but this will decrease the precision of the result set. A good result set has to be a balance between precision and recall.

We should optimize our results for precision if the hits are plentiful and several results can meet the search criteria. Since we have a huge collection of documents, it makes sense to provide a few relevant and good hits as opposed to adding irrelevant results in the result set. An example scenario where optimization for precision makes sense is web search where the available number of documents is huge.

On the other hand, we should optimize for recall if we do not want to miss out any relevant document. This happens when the collection of documents is comparatively small. It makes sense to return all relevant documents and not care about the irrelevant documents added to the result set. An example scenario where recall makes sense is patent search.

Traditional accuracy of the result set is defined by the following formula:

\[
\text{Accuracy} = 2 \times \frac{(\text{precision} \times \text{recall})}{(\text{precision} + \text{recall})}
\]
This combines both precision and recall and is a harmonic mean of precision and recall. Harmonic mean is a type of averaging mean used to find the average of fractions. This is an ideal formula for accuracy and can be used as a reference point while figuring out the combination of precision and recall that your result set will provide.

Let us look at some practical problems faced while searching in different business scenarios.

The e-commerce problem statement

E-commerce provides an easy way to sell products to a large customer base. However, there is a lot of competition among multiple e-commerce sites. When users land on an e-commerce site, they expect to find what they are looking for quickly and easily. Also, users are not sure about the brands or the actual products they want to purchase. They have a very broad idea about what they want to buy. Many customers nowadays search for their products on Google rather than visiting specific e-commerce sites. They believe that Google will take them to the e-commerce sites that have their product.

The purpose of any e-commerce website is to help customers narrow down their broad ideas and enable them to finalize the products they want to purchase. For example, suppose a customer is interested in purchasing a mobile. His or her search for a mobile should list mobile brands, operating systems on mobiles, screen size of mobiles, and all other features as facets. As the customer selects more and more features or options from the facets provided, the search narrows down to a small list of mobiles that suit his or her choice. If the list is small enough and the customer likes one of the mobiles listed, he or she will make the purchase.

The challenge is also that each category will have a different set of facets to be displayed. For example, searching for books should display their format, as in paperpack or hardcover, author name, book series, language, and other facets related to books. These facets were different for mobiles that we discussed earlier. Similarly, each category will have different facets and it needs to be designed properly so that customers can narrow down to their preferred products, irrespective of the category they are looking into.
The takeaway from this is that categorization and feature listing of products should be taken care of. Misrepresentation of features can lead to incorrect search results. Another takeaway is that we need to provide multiple facets in the search results. For example, while displaying the list of all mobiles, we need to provide facets for a brand. Once a brand is selected, another set of facets for operating systems, network, and mobile phone features has to be provided. As more and more facets are selected, we still need to show facets within the remaining products.

Another problem is that we do not know what product the customer is searching for. A site that displays a huge list of products from different categories, such as electronics, mobiles, clothes, or books, needs to be able to identify what the customer is searching for. A customer can be searching for Samsung, which can be in mobiles, tablets, electronics, or computers. The site should be able to identify whether the customer has input the author name or the book name. Identifying the input would help in increasing the relevance of the result set by increasing the precision of the search results. Most e-commerce sites provide search suggestions that include the category to help customers target the right category during their search.

Amazon, for example, provides search suggestions that include both latest searched terms and products along with category-wise suggestions:
Search suggestions on Amazon.com

It is also important that products are added to the index as soon as they are available. It is even more important that they are removed from the index or marked as sold out as soon as their stock is exhausted. For this, modifications to the index should be immediately visible in the search. This is facilitated by a concept in Solr known as Near Real Time Indexing and Search (NRT). More details on using Near Real Time Search will be explained later in this chapter.

The job site problem statement

A job site serves a dual purpose. On the one hand, it provides jobs to candidates, and on the other, it serves as a database of registered candidates' profiles for companies to shortlist.

A job search has to be very intuitive for the candidates so that they can find jobs suiting their skills, position, industry, role, and location, or even by the company name. As it is important to keep the candidates engaged during their job search, it is important to provide facets on the abovementioned criteria so that they can narrow down to the job of their choice. The searches by candidates are not very elaborate. If the search is generic, the results need to have high precision. On the other hand, if the search does not return many results, then recall has to be high to keep the candidate engaged on the site. Providing a personalized job search to candidates on the basis of their profiles and past search history makes sense for the candidates.
On the recruiter side, the search provided over the candidate database is required to have a huge set of fields to search upon every data point that the candidate has entered. The recruiters are very selective when it comes to searching for candidates for specific jobs. Educational qualification, industry, function, key skills, designation, location, and experience are some of the fields provided to the recruiter during a search. In such cases, the precision has to be high. The recruiter would like a certain candidate and may be interested in more candidates similar to the selected candidate. The more like this search in Solr can be used to provide a search for candidates similar to a selected candidate.

NRT is important as the site should be able to provide a job or a candidate for a search as soon as any one of them is added to the database by either the recruiter or the candidate. The promptness of the site is an important factor in keeping users engaged on the site.

Challenges of large-scale indexing
Let us understand how indexing happens and what can be done to speed it up. We will also look at the challenges faced during the indexing of a large number of documents or bulky documents. An e-commerce site is a perfect example of a site containing a large number of products, while a job site is an example of a search where documents are bulky because of the content in candidate resumes.

During indexing, Solr first analyzes the documents and converts them into tokens that are stored in the RAM buffer. When the RAM buffer is full, data is flushed into a segment on the disk. When the numbers of segments are more than that defined in the MergeFactor class of the Solr configuration, the segments are merged. Data is also written to disk when a commit is made in Solr.

Let us discuss a few points to make Solr indexing fast and to handle a large index containing a huge number of documents.

Using multiple threads for indexing on Solr
We can divide our data into smaller chunks and each chunk can be indexed in a separate thread. Ideally, the number of threads should be twice the number of processor cores to avoid a lot of context switching. However, we can increase the number of threads beyond that and check for performance improvement.
Using the Java binary format of data for indexing

Instead of using XML files, we can use the Java bin format for indexing. This reduces a lot of overhead of parsing an XML file and converting it into a binary format that is usable. The way to use the Java bin format is to write our own program for creating fields, adding fields to documents, and finally adding documents to the index.

Here is a sample code:

```java
//Create an instance of the Solr server
String SOLR_URL = "http://localhost:8983/solr"
SolrServer server = new HttpSolrServer(SOLR_URL);

//Create collection of documents to add to Solr server
SolrInputDocument doc1 = new SolrInputDocument();
document.addField("id",1);
document.addField("desc", "description text for doc 1");

SolrInputDocument doc2 = new SolrInputDocument();
document.addField("id",2);
document.addField("desc", "description text for doc 2");

Collection<SolrInputDocument> docs = new ArrayList<SolrInputDocument>();
docs.add(doc1);
docs.add(doc2);

//Add the collection of documents to the Solr server and commit.
server.add(docs);
server.commit();
```


Add all files from the `<solr_directory>/dist` folder to the classpath for compiling and running the HttpSolrServer program.
Using the ConcurrentUpdateSolrServer class for indexing

Using the ConcurrentUpdateSolrServer class instead of the HttpSolrServer class can provide performance benefits as the former uses buffers to store processed documents before sending them to the Solr server. We can also specify the number of background threads to use to empty the buffers. The API docs for ConcurrentUpdateSolrServer are found in the following link: http://lucene.apache.org/solr/4_6_0/solr-solrj/org/apache/solr/client/solrj/impl/ConcurrentUpdateSolrServer.html

The constructor for the ConcurrentUpdateSolrServer class is defined as:

    ConcurrentUpdateSolrServer(String solrServerUrl, int queueSize, int threadCount)

Here, queueSize is the buffer and threadCount is the number of background threads used to flush the buffers to the index on disk.

Note that using too many threads can increase the context switching between threads and reduce performance. In order to optimize the number of threads, we should monitor performance (docs indexed per minute) after each increase and ensure that there is no decrease in performance.

Solr configuration changes that can improve indexing performance

We can change the following directives in solrconfig.xml file to improve indexing performance of Solr:

- **ramBufferSizeMB**: This property specifies the amount of data that can be buffered in RAM before flushing to disk. It can be increased to accommodate more documents in RAM before flushing to disk. Increasing the size beyond a particular point can cause swapping and result in reduced performance.

- **maxBufferedDocs**: This property specifies the number of documents that can be buffered in RAM before flushing to disk. Make this a large number so that commit always happens on the basis of the RAM buffer size instead of the number of documents.

- **useCompoundFile**: This property specifies whether to use a compound file or not. Using a compound file reduces indexing performance as extra overhead is required to create the compound file. Disabling a compound file can create a large number of file descriptors during indexing.
The default number of file descriptors available in Linux is 1024. Check the number of open file descriptors using the following command:

cat /proc/sys/fs/file-max

Check the hard and soft limits of file descriptors using the ulimit command:

ulimit -Hn
ulimit -Sn

To increase the number of file descriptors system wide, edit the file /etc/sysctl.conf and add the following line:

fs.file-max = 100000

The system needs to be rebooted for the changes to take effect.

To temporarily change the number of file descriptors, run the following command as root:

Sysctl –w fs.file-max = 100000

- **mergeFactor**: Increasing the mergeFactor can cause a large number of segments to be merged in one go. This will speed up indexing but slow down searching. If the merge factor is too large, we may run out of file descriptors, and this may even slow down indexing as there would be lots of disk I/O during merging. It is generally recommended to keep the merge factor constant or lower it to improve searching.

**Planning your commit strategy**

Disable the autocommit property during indexing so that commit can be done manually. Autocommit can be a pain as it can cause too frequent commits. Instead, committing manually can reduce the overhead during commits by decreasing the number of commits. Autocommit can be disabled in the solrconfig.xml file by setting the `<autocommit><maxtime>` properties to a very large value.

Another strategy would be to configure the `<autocommit><maxtime>` properties to a large value and use the autoSoftCommit property for short-time commits to disk. Soft commits are faster as the commit is not synced to disk. Soft commits are used to enable near real time search.

We can also use the commitWithin tag instead of the autoSoftCommit tag. The former forces documents to be added to Solr via soft commit at certain intervals of time. The commitWithin tag can also be used with hard commits via the following configuration:

```xml
<commitWithin><softCommit>false</softCommit></commitWithin>
```
Avoid using the autoSoftCommit / autoCommit / commitWithin tags while adding bulk documents as it has a major performance impact.

**Using better hardware**

Indexing involves lots of disk I/O. Therefore, it can be improved by using a local file system instead of a remote file system. Also, using better hardware with higher IO capability, such as **Solid State Drive (SSD)**, can improve writes and speed up the indexing process.

**Distributed indexing**

When dealing with large amounts of data to be indexed, in addition to speeding up the indexing process, we can work on distributed indexing. Distributed indexing can be done by creating multiple indexes on different machines and finally merging them into a single, large index. Even better would be to create the separate indexes on different Solr machines and use Solr sharding to query the indexes across multiple shards.

For example, an index of 10 million products can be broken into smaller chunks based on the product ID and can be indexed over 10 machines, with each indexing a million products. While searching, we can add these 10 Solr servers as shards and distribute our search queries over these machines.

**The SolrCloud solution**

SolrCloud provides the high availability and failover solution for an index spanning over multiple Solr servers. If we go ahead with the traditional master-slave model and try implementing a sharded Solr cluster, we will need to create multiple master Solr servers, one for each shard and then slaves for these master servers. We need to take care of the sharding algorithm so that data is distributed across multiple shards. A search has to happen across these shards. Also, we need to take care of any shard that goes down and create a failover setup for the same. Load balancing of search queries is manual. We need to figure out how to distribute the search queries across multiple shards.

SolrCloud handles the scalability challenge for large indexes. It is a cluster of Solr servers or cores that can be bound together as a single Solr (cloud) server. SolrCloud is used when there is a need for highly scalable, fault-tolerant, distributed indexing and search capabilities. With SolrCloud, a single index can span across multiple Solr cores that can be on different Solr servers. Let us go through some of the concepts of SolrCloud:
• **Collection**: A logical index that spans across multiple Solr cores is called a collection. Thus, if we have a two-core Solr index on a single Solr server, it will create two collections with multiple cores in each collection. The cores can reside on multiple Solr servers.

• **Shard**: In SolrCloud, a collection can be sliced into multiple shards. A shard in SolrCloud will consist of multiple copies of the slice residing on different Solr cores. Therefore, in SolrCloud, a collection can have multiple shards. Each shard will have multiple Solr cores that are copies of each other.

• **Leader**: One of the cores within a shard will act as a leader. The leader is responsible for making sure that all the replicas within a shard are up to date.

SolrCloud has a central configuration that can be replicated automatically across all the nodes that are part of the SolrCloud cluster. The central configuration is maintained using a configuration management and coordination system known as Zookeeper. Zookeeper provides reliable coordination across a huge cluster of distributed systems. Solr does not have a master node. It uses Zookeeper to maintain node, shard, and replica information based on configuration files and schemas. Documents can be sent to any server, and Zookeeper will be able to figure out where to index them. If a leader for a shard goes down, another replica is automatically elected as the new leader using Zookeeper.

If a document is sent to a replica during indexing, it is forwarded to the leader. On receiving the document at a leader node, the SolrCloud determines whether the document should go to another shard and forwards it to the leader of that shard. The leader indexes the document and forwards the index notification to its replicas.
SolrCloud provides automatic failover. If a node goes down, indexing and search can happen over another node. Also, search queries are load balanced across multiple shards in the Solr cluster. Near Real Time Indexing is a feature where, as soon as a document is added to the index, the same is available for search. The latest Solr server contains commands for soft commit, which makes documents added to the index available for search immediately without going through the traditional commit process. We would still need to make a hard commit to make changes onto a stable data store. A soft commit can be carried out within a few seconds, while a hard commit takes a few minutes. SolrCloud exploits this feature to provide near real time search across the complete cluster of Solr servers.

It can be difficult to determine the number of shards in a Solr collection in the first go. Moreover, creating more shards or splitting a shard into two can be a tedious task if done manually. Solr provides inbuilt commands for splitting a shard. The previous shard is maintained and can be deleted at a later date.

SolrCloud also provides the ability to search the complete collection of one or more particular shards if needed.

SolrCloud removes all the hassles of maintaining a cluster of Solr servers manually and provides an easy interface to handle distributed search and indexing over a cluster of Solr servers with automatic failover. We will be discussing SolrCloud in Chapter 9, SolrCloud.

Summary

In this chapter, we went through the basics of indexing in Solr. We saw the structure of the Solr index and how analyzers, tokenizers, and filters work in the conversion of text into searchable tokens. We went through the complexities involved in multilingual search and also discussed the strategies that can be used to handle the complexities. We discussed the formula for measuring the quality of search results and understood the meaning of precision and recall. We saw in brief the problems faced by e-commerce and job websites during indexing and search. We discussed the challenges faced while indexing a large number of documents. We saw some tips on improving the speed of indexing. Finally, we discussed distributed indexing and search and how SolrCloud provides a solution for implementing the same.
Where to buy this book

You can buy Apache Solr Search Patterns from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.