Scaling Big Data with Hadoop and Solr

Hrishikesh Karambelkar

Chapter No. 2
"Understanding Solr"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.2 "Understanding Solr"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Hrishikesh Karambelkar is a software architect with a blend of entrepreneurial and professional experience. His core expertise involves working with multiple technologies such as Apache Hadoop and Solr, and architecting new solutions for the next generation of a product line for his organization. He has published various research papers in the domains of graph searches in databases at various international conferences in the past. On a technical note, Hrishikesh has worked on many challenging problems in the industry involving Apache Hadoop and Solr.

While writing the book, I spend my late nights and weekends bringing in the value for the readers. There were few who stood by me during good and bad times, my lovely wife Dhanashree, my younger brother Rupesh, and my parents. I dedicate this book to them. I would like to thank the Apache community users who added a lot of interesting content for this topic, without them, I would not have got an opportunity to add new interesting information to this book.

For More Information:
Scaling Big Data with Hadoop and Solr

This book will provide users with a step-by-step guide to work with Big Data using Hadoop and Solr. It starts with a basic understanding of Hadoop and Solr, and gradually gets into building efficient, high performance enterprise search repository for Big Data.

You will learn various architectures and data workflows for distributed search system. In the later chapters, this book provides information about optimizing the Big Data search instance ensuring high availability and reliability.

This book later demonstrates two real world use cases about how Hadoop and Solr can be used together for distributer enterprise search.

What This Book Covers

Chapter 1, Processing Big Data Using Hadoop and MapReduce, introduces you with Apache Hadoop and its ecosystem, HDFS, and MapReduce. You will also learn how to write MapReduce programs, configure Hadoop cluster, the configuration files, and the administration of your cluster.

Chapter 2, Understanding Solr, introduces you to Apache Solr. It explains how you can configure the Solr instance, how to create indexes and load your data in the Solr repository, and how you can use Solr effectively for searching. It also discusses interesting features of Apache Solr.

Chapter 3, Making Big Data Work for Hadoop and Solr, brings the two worlds together; it drives you through different approaches for achieving Big Data work with architectures and their benefits and applicability.

Chapter 4, Using Big Data to Build Your Large Indexing, explains the NoSQL and concepts of distributed search. It then gets you into using different algorithms for Big Data search covering shards and indexing. It also talks about SolrCloud configuration and Lily.

Chapter 5, Improving Performance of Search while Scaling with Big Data, covers different levels of optimizations that you can perform on your Big Data search instance as the data keeps growing. It discusses different performance improvement techniques which can be implemented by the users for their deployment.

For More Information:
Appendix A, *Use Cases for Big Data Search*, describes some industry use cases and case studies for Big Data using Solr and Hadoop.

Appendix B, *Creating Enterprise Search Using Apache Solr*, shares a sample Solr schema which can be used by the users for experimenting with Apache Solr.

Appendix C, *Sample MapReduce Programs to Build the Solr Indexes*, provides a sample MapReduce program to build distributed Solr indexes for different approaches.

For More Information:  
Understanding Solr

The exponential growth of data coming from various applications over the past decade has created many challenges. Handling such massive data demanded focus on the development of scalable search engines. It also triggered development of data analytics. Apache Lucene along with Mahout and Solr were developed to address these needs. Out of these, Mahout was moved as a separate Apache top-level project, and Apache Solr was merged into the Lucene project itself.

Apache Solr is an open source enterprise search application which provides user abilities to search structured as well as unstructured data across the organization. It is based on the Apache Lucene libraries for information retrieval. Apache Lucene is an open source information retrieval library used widely by various organizations. Apache Solr is completely developed on Java stack of technologies. Apache Solr is a web application, and Apache Lucene is a library consumed by Apache Solr for performing search. We will try to understand Apache Solr in this chapter, while covering the following topics:

- Installation of an Apache Solr
- Understanding the Apache Solr architecture
- Configuring a Solr instance
- Understanding various components of Solr in detail
- Understanding data loading

For More Information:
Installing Solr

Apache Solr comes by default with a demo server based on Jetty, which can be downloaded and run. However, you can choose to customize it, and deploy it in your own environment. Before installation, you need to make sure that you have JDK 1.5 or above on your machines. You can download the stable installer from http://lucene.apache.org/solr/ or from its nightly builds running on the same site. You may also need a utility called curl to run your samples. There are commercial versions of Apache Solr available from a company called LucidWorks (http://www.lucidworks.com). Solr being a web-based application can run on many operating systems such as *nix and Windows.

Some of the older versions of Solr have failed to run properly due to locale differences on host systems. If your system's default locale, or character set is non-english (that is, en/en-US), for safety, you can override your system defaults for Solr by passing -Duser.language and -Duser.country in your Jetty to ensure smooth running of Solr.

If you are planning to run Solr in your own container, you need to deploy solr.war from the distribution to your container. You can simply check whether your instance is running or not by accessing its admin page available at http://localhost:8983/solr/admin. The following screenshot shows the Solr window:

If you are building Solr from source, then you need Java SE 6 JDK (Java Development Kit), Apache Ant distribution (1.8.2 or higher), and Apache Ivy (2.2.0 or higher). You can compile the source by simply navigating to Solr directory and running Ant from the directory.
Apache Solr architecture

Apache Solr is composed of multiple modules, some of them being separate projects in themselves. Let's understand the different components of Apache Solr architecture. The following diagram depicts the Apache Solr conceptual architecture:

Apache Solr can run as a single core or multicore. A Solr core is nothing but the running instance of a Solr index along with its configuration. Earlier, Apache Solr had a single core which in turn limited the consumers to run Solr on one application through a single schema and configuration file. Later support for creating multiple cores was added. With this support, now, one can run one Solr instance for multiple schemas and configurations with unified administrations. You can run Solr in multicore with the following command:

```java
java -Dsolr.solr.home=multicore -jar start.jar
```

Storage

The storage of Apache Solr is mainly used for storing metadata and the actual index information. It is typically a file store locally, configured in the configuration of Apache Solr. The default Solr installation package comes with a Jetty server, the respective configuration can be found in the solr.home/conf folder of Solr install.

For More Information:

There are two major configuration files in Solr described as follows:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solrconfig.xml</td>
<td>This is the main configuration file of your Solr install. Using this you can control each and every thing possible; write from caching, specifying customer handlers, codes, and commit options.</td>
</tr>
<tr>
<td>schema.xml</td>
<td>This file is responsible for defining a Solr schema for your application. For example, Solr implementation for log management would have schema with log-related attributes such as log levels, severity, message type, container name, application name, and so on.</td>
</tr>
<tr>
<td>solr.xml</td>
<td>Using solr.xml, you can configure Solr cores (single or multiple) for your setup. It also provides additional parameters such as zookeeper timeout, transient cache size, and so on.</td>
</tr>
</tbody>
</table>

Apache Solr (underlying Lucene) indexing is a specially designed data structure, stored in the file system as a set of index files. The index is designed with specific format in such a way to maximize the query performance.

**Solr engine**

A Solr engine is nothing but the engine responsible for making Solr what it is today. A Solr engine with metadata configuration together forms the Solr core. When Solr runs in a replication mode, the index replicator is responsible for distributing indexes across multiple slaves. The master server maintains index updates, and slaves are responsible for talking with master to get them replicated. Apache Lucene core gets packages as library with Apache Solr application. It provides core functionality for Solr such as index, query processing, searching data, ranking matched results, and returning them back.

**The query parser**

Apache Lucene comes with variety of query implementations. Query parser is responsible for parsing the queries passed by the end search as a search string. Lucene provides TermQuery, BooleanQuery, PhraseQuery, PrefixQuery, RangeQuery, MultiTermQuery, FilteredQuery, SpanQuery, and so on as query implementations. IndexSearcher is a basic component of Solr searched with a default base searcher class. This class is responsible for returning ordered matched results of searched keywords ranked as per the computed score. IndexReader provides access to indexes stored in the file system. It can be used for searching for an index. Similar to IndexReader, IndexWriter allows you to create and maintain indexes in Apache Lucene.

For More Information:

Tokenizer breaks field data into lexical units or tokens. Filter examines field of tokens from Tokenizer and either it keeps them, transforms them, discards them, or creates new ones. Tokenizer and Filter together form chain or pipeline of analyzers. There can only be one Tokenizer per Analyzer. The output of one chain is fed to another. Analyzing process is used for indexing as well as querying by Solr. They play an important role in speeding up the query as well as index time; they also reduce the amount of data that gets generated out of these operations. You can define your own customer analyzers depending upon your use case. The following diagram shows the example of a filter:

Application layer represents Apache Solr web application. It consists of different UI templates, request/response handlers, and different faceting provided by Solr.

Faceted browsing is one of the main features of Apache Solr; it helps users reach the right set of information they wanted to get. The facets and components deal with providing the faceted search capabilities on top of Lucene. When a user fires a search query on Solr, it actually gets passed on to a request handler. By default, Apache Solr provides DisMaxRequestHandler. This handler is designed to work for simple user queries. It can only search one field by default. You can visit here to find more details about this handler. Based on the request, request handler calls query parser.

For More Information:
**Query parser** is responsible for parsing the queries, and converting it into Lucene query objects. There are different types of parsers available (Lucene, DisMax, eDisMax, and so on). Each parser offers different functionalities and it can be used based on the requirements. Once a query is parsed, it hands it over to index searcher or reader. The job of **index reader** is to run the queries on index store, and gather the results to response writer.

**Response Writer** is responsible for responding back to the client; it formats the query response based on search outcomes from the Lucene engine.

The following diagram displays complete process flow when a search is fired from a client:

Apache Solr ships with an example schema that runs using Apache velocity. Apache velocity is a fast, open source template engine which quickly generates an HTML-based frontend. Users can customize these templates as per their requirements.

**Index handler** is one type of update handler that handles the task of addition, updation, and deletion of document for indexing. Apache Solr supports updates through index handler through JSON, XML, and text format.

**Data Import Handler** (DIH) provides a mechanism for integrating different data sources with Apache Solr for indexing. The data sources could be relational databases or web-based sources (for example, RSS, ATOM feeds, and e-mails).

For More Information:
Although DIH is part of Solr development, the default installation does not include it in the Solr application.

Apache Tika, a project in itself extends capabilities of Apache Solr to run on top of different types of files. When assigned a document to Tika, it automatically determines the type of file (that is, Word, Excel, or PDF) and extracts the content. Tika also extracts document metadata such as author, title, creation date, and so on, which if provided in schema go as text field in Apache Solr.

Interaction
Apache Solr, although a web-based application, can be integrated with different technologies. So, if a company has Drupal-based e-commerce site, they can integrate Apache Solr application and provide its rich faceted search to the user.

Client APIs and SolrJ client
Apache Solr client provides different ways of talking with Apache Solr web application. This enables Solr to easily get integrated with any application. Using client APIs, consumers can run search, and perform different operations on indexes. SolrJ or Solr Java client is an interface of Apache Solr with Java. SolrJ client enables any Java application to talk directly with Solr through its extensive library of APIs. Apache SolrJ is part of Apache Solr package.

Other interfaces
Apache Solr can be integrated with other various technologies using its API library and standards-based interfacing. JavaScript-based clients can straightaway talk with Solr using JSON-based messaging. Similarly, other technologies can simply connect to Apache Solr running instance through HTTP, and consume its services either through JSON, XML, and text formats.

Configuring Apache Solr search
Apache Solr allows extensive configuration to meet the needs of the consumer. Configuring the instance revolves around the following:

- Defining a schema
- Configuring Solr parameters

Let's look at all these steps to understand the configuration of Apache Solr.

For More Information:
Defining a Schema for your instance

Apache Solr lets you define the structure of your data to extend support for searching across the traditional keyword search. You can allow Solr to understand the structure of your data (coming from various sources) by defining fields in the schema definition file. These fields once defined, will be made available at the time of data import or data upload. The schema is stored in the `schema.xml` file in the `etc` folder of Apache Solr.

Apache Solr ships with a default `schema.xml` file, which you have to change to fit your needs. In schema configuration, you can define field types, (for example, `String`, `Integer`, `Date`), and map them to respective Java classes. Apache Solr ships with default data types for text, integer, date, and so on.

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
```

This enables users to define the custom type in case if they wish to. Then you can define the fields with name and type pointing to one of the defined types. A field in Solr will have the following major attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Sets default value, if not read while importing a document</td>
</tr>
<tr>
<td>indexed</td>
<td>True, when it has to be indexed (that is, can be searched, sorted, and facet creation)</td>
</tr>
<tr>
<td>stored</td>
<td>When true, a field is stored in the index store, and it will be accessible while displaying results</td>
</tr>
<tr>
<td>compressed</td>
<td>When true, the field will be zipped (using gzip). It is applicable for text-based fields</td>
</tr>
<tr>
<td>multiValued</td>
<td>If a field contains multiple values in the same import cycle of the document/row</td>
</tr>
<tr>
<td>omitNorms</td>
<td>When true, it omits the norms associated with field (such as length normalization and index boosting)</td>
</tr>
<tr>
<td>termVectors</td>
<td>When true, it also persists metadata related to document, and returns that when queried.</td>
</tr>
</tbody>
</table>

For More Information:
Each Solr instance should have a unique identifier field (ID) although it's not mandatory condition. In addition to static fields, you can also use Solr dynamic fields for getting the flexibility in case if you do not know the schema afront. Use the `<dynamicField>` declaration for creating a field rule to allow Solr understands which data type to be used. In the following sample code, any field imported, and identified as * _no (for example, id_no, book_no) will in turn be read as integer by Solr.

```xml
<dynamicField name="*_no" type="integer" indexed="true"
stored="true"/>
```

You can also index same data into multiple fields by using the `<copyField>` directive. This is typically needed when you want to have multi-indexing for same data type, for example, if you have data for refrigerator with company followed by model number (WHIRLPOOL-1000LTR, SAMSUNG-980LTR), you can have these indexed separately by applying your own tokenizers to different field. You might generate indexes for two different fields, company name and model number. You can define Tokenizers specific to your field types. Similarly, a Lucene class responsible for scoring the matched results. Solr allows you to override default similarity behavior through the `<similarity>` declaration. Similarity can be configured at the global level; however with Solr 4.0, it extends similarity to be configured at the field level.

## Configuring a Solr instance

Once a schema is configured, next step would be to configure the instance itself. To configure the instance, you need to touch upon many files, some of them are configuration files, and some of them are metadata files. The entire configuration is part of `/conf` directory where Solr instance is setup. You can simply run examples by going to the `examples/example-docs` directory and running the following code:

```
java -jar post.jar solr.xml monitor.xml
```

For More Information:

Now, try accessing your instance by typing http://localhost:8983/solr/collection1/browse, and you will be able to see the following screenshot when you search on Advanced:

![Screenshot of Solr interface showing search results for Advanced configuration files]

**Configuration files**

There are two major configurations that go in the Solr configuration: namely, solrconfig.xml and solr.xml. Among these, solr.xml is responsible for maintaining configuration for logging, cloud setup, and Solr core primarily, whereas solrconfig.xml focuses more on the Solr application front. Let's look at the solrconfig.xml file, and understand all the important declarations you'd be using frequently.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>luceneMatchVersion</td>
<td>It tells which version of Lucene/Solr the solrconfig.xml configuration file is set to. When upgrading your Solr instances, you need to modify this attribute.</td>
</tr>
</tbody>
</table>
| lib             | In case if you create any plugins for Solr, you need to put a library reference here, so that it gets picked up. The libraries are loaded in the same sequence that of the configuration order. The paths are relative; you can also specify regular expressions. For example, <lib dir=".../../../contrib/velocity/lib" regex=".*\.jar" />

For More Information:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataDir</td>
<td>By default Solr uses ./data directory for storing indexes, however this can be overridden by changing the directory for data using this directive.</td>
</tr>
<tr>
<td>indexConfig</td>
<td>This directive is of xsd complexType, and it allows you to change the settings of some of the internal indexing configuration of Solr.</td>
</tr>
<tr>
<td>filter</td>
<td>You can specify different filters to be run at the time of index creation.</td>
</tr>
<tr>
<td>writeLockTimeout</td>
<td>This directive denotes maximum time to wait for the write lock for IndexWriter.</td>
</tr>
<tr>
<td>maxIndexingThreads</td>
<td>It denotes maximum number of index threads that can run in the IndexWriter class; if more threads arrive, they have to wait. The default value is 8.</td>
</tr>
<tr>
<td>ramBufferSizeMB</td>
<td>It specifies the maximum RAM you need in the buffer while index creation, before the files are flushed to filesystem.</td>
</tr>
<tr>
<td>maxBufferedDocs</td>
<td>It specifies the limited number of documents buffered.</td>
</tr>
<tr>
<td>lockType</td>
<td>When indexes are generated and stored in the file, this mechanism decides which file locking mechanism to be used to manage concurrent read/writes. There are three types: single (one process at a time), native (native operating system driven), and simple (based on locking using plain files).</td>
</tr>
<tr>
<td>unlockOnStartup</td>
<td>When true, it will release all the write locks held in the past.</td>
</tr>
<tr>
<td>Jmx</td>
<td>Solr can expose statistics of runtime through MBeans. It can be enabled or disabled through this directive.</td>
</tr>
<tr>
<td>updateHandler</td>
<td>This directive is responsible for managing the updates to Solr. The entire configuration for updateHandler goes as a part of this directive.</td>
</tr>
<tr>
<td>updateLog</td>
<td>You can specify the directory and other configuration for transaction logs during index updates.</td>
</tr>
<tr>
<td>autoCommit</td>
<td>It enables automatic commit, when updates are done. This could be based on the documents or time.</td>
</tr>
<tr>
<td>Listener</td>
<td>Using this directive, you can subscribe to update events when IndexWriter is updating the index. The listeners can be run either at the time of postCommit or postOptimize.</td>
</tr>
<tr>
<td>Query</td>
<td>This directive is mainly responsible for controlling different parameters at the query time.</td>
</tr>
</tbody>
</table>

For More Information:  
### Understanding Solr

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestDispatcher</td>
<td>By setting parameters in this directive, you can control how a request will be processed by SolrDispatchFilter.</td>
</tr>
<tr>
<td>requestHandler</td>
<td>It is described separately in the next section.</td>
</tr>
<tr>
<td>searchComponent</td>
<td>It is described separately in the next section.</td>
</tr>
<tr>
<td>updateRequestProcessor</td>
<td>The updateRequestProcessor chain defines how update requests are processed; you can define your own updateRequestProcessor to perform things such as cleaning up data, optimizing text fields, and so on.</td>
</tr>
<tr>
<td>queryResponseWriter</td>
<td>Each request for query is formatted and written back to user through queryResponseWriter. You can extend your Solr instance to have responses for XML, JSON, PHP, Ruby, Python, and CSVs by enabling respective predefined writers. If you have a custom requirement for a certain type of response, it can easily be extended.</td>
</tr>
<tr>
<td>queryParser</td>
<td>The queryParser directive tells Apache Solr which query parser to be used for parsing the query and creating the Lucene query objects. Apache Solr contains predefined query parsers such as Lucene (default), DisMax (based on weights of fields), and eDisMax (similar to DisMax, with some additional features).</td>
</tr>
</tbody>
</table>

### Request handlers and search components

Apache Solr gets requests for searching on data or index generation. In such cases, RequestHandler is the directive through which you can define different ways of tackling these requests. One request handler is assigned with one relative URL where it would serve the request. A request handler may or may not provide search facility. In case if it provides, it is also called searchHandler. RealTimeGetHandler provides latest stored fields of any document. UpdateRequestHandler is responsible for updating the process of the index. Similarly, CSVRequestHandler and JsonUpdateRequestHandler takes the responsibility of updating the indexes with CSV and JSON formats, respectively. ExtractingRequestHandler uses Apache Tika to extract the text out of different file formats. By default, there are some important URLs configured with Apache Solr which are listed as follows:

For More Information:
A searchComponent is a one of the main feature of Apache Solr. It brings the capability of enhancing new features to Apache Solr. You can use searchComponent in your searchHandler. It has to be defined separately from requestHandler. These components can be defined, and then they can be used in any of the requestHandler directives. Some components also allow access through either searchComponent, or directly as a separate request handler. You can alternatively specify your query parser in the context of your requestHandler. Different parsers can be used for this. The default parser is the Lucene-based standard parser.
Understanding Solr

## Facet

Facets are one of the primary features of Apache Solr. Your search results can be organized in different formats through facets. This is an effective way of helping users to drill down to right set of information. The following screenshot shows one of a customized instance of Apache Solr with facets on the left-hand side:

Using facets, you can filter down your query. Facets can be created on your schema-based fields. So, considering the log-based search, you can create facets based on the log severity. There are different types of facets:

<table>
<thead>
<tr>
<th>Facet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field-value</td>
<td>You can have your schema fields as facet component here. It shows the count of top fields.</td>
</tr>
<tr>
<td>Range</td>
<td>Range faceting is mostly used on date/numeric fields, and it supports range queries. You can specify start and end dates, gap in the range, and so on.</td>
</tr>
<tr>
<td>Date</td>
<td>This is a deprecated faceting, and it is now being handled in the range faceting itself.</td>
</tr>
<tr>
<td>Pivot</td>
<td>Pivot gives you the ability to perform simple math on your data. With this facet, you can summarize your results, and then you can get them sorted, and take average. This gives you hierarchical results (also sometimes called hierarchical faceting).</td>
</tr>
</tbody>
</table>

For More Information:  
MoreLikeThis

The Solr-based search results are enhanced with the MoreLikeThis component, because they provide a better user-browsing experience by allowing the user to choose similar results. This component can be accessed either through requestHandler, or through searchcomponent.

Highlight

The matched search string can be highlighted in the search results when a user fires a query to Apache Solr through the highlight search component.

SpellCheck

Searching in Solr can be extended further with the support for spell checks using the spellcheck component. You can get support for multiple dictionaries together per field. This is very useful in case of multilingual data. It also has a Suggestor that responds to users with Did you mean type of suggestions. Additionally, Suggestor with autocomplete feature starts providing users options right at the time when user is typing search query enhancing the overall experience.

Metadata management

We have already seen the solr.xml, solrconfig.xml, and schema.xml configuration files. Besides these, there are other files where a metadata can be specified. These files again appear in the conf directory of Apache Solr.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protwords.txt</td>
<td>In this file, you can specify protected words that you do not wish to get stemmed. For example, a stemmer might stem the word catfish to cat or fish.</td>
</tr>
<tr>
<td>currency.txt</td>
<td>Current stores mapping between exchange rates across different countries; this file is helpful when you have your application accessed by people from different country.</td>
</tr>
<tr>
<td>elevate.txt</td>
<td>With this file, you can influence the search results and make your own results among the top ranked results. This overrides Lucene's standard ranking scheme taking into account the elevations from this file.</td>
</tr>
<tr>
<td>spellings.txt</td>
<td>In this file, you can provide spelling suggestions to the end user.</td>
</tr>
</tbody>
</table>

For More Information:

Using this file, you can specify your own synonyms. For example, Cost => money, Money => dollars.

Stopwords are those words which will not be indexed and used by Solr in the applications; this is particularly helpful when you wish to get rid of certain words, for example, in the string, Jamie and Joseph, the word and can be marked as a stopword.

**Loading your data for search**

Once a Solr instance is configured, next step is to index your data, and then simply use the instance for querying and analyzing. Apache Solr/Lucene is designed in such a way that it allows you to plugin any type of data from any data source in the world. If you have structured data, it makes sense to extract the structured information, create exhaustive Solr schema, and feed in the data to Solr, effectively adding different data dimensions to your search. Data Import Handler (DIH) is used mainly for indexing structured data. It is mainly associated with data sources such as relational databases, XML databases, RSS feeds, and ATOM feeds. DIH uses multiple entity processors to extract the data from various data sources, transform them, and finally generate indexes out of it. For example, in a relational database, a table or a view can be viewed as an entity. DIH allows you to write your own custom entity processors. There are different ways to load the data in Apache Solr as shown in the following diagram:

[Diagram showing data loading process]

For More Information:
**ExtractingRequestHandler/Solr Cell**

Solr Cell is one of the most powerful handlers for uploading any type of data. If you wish you can run Solr on a set of files/unstructured data containing different formats such as MS Office, PDF, e-book, e-mail, text, and so on. In Apache Tika, text extraction is based purely on how exhaustive any file is. Therefore, if you have a PDF of scanned images containing text, Apache Tika won't be able to extract any of the text out of it. In such cases, you need to use **Optical Character Recognition (OCR)**-based software to bring in such functionality for Solr. You can simply try this on your downloaded curl utility, and then running it on your document:

```
curl 'http://localhost:8983/solr/update/extract?literal.id=doc1&commit=true' -F "myfile=@<your document name with extension>"
```

Index handlers such as **SimplePostTool**, **UpdateRequestHandler**, and **SolrJ** provide addition, updation, and deletion of documents to index them for XML, JSON, and CSV format. **UpdateRequestHandler** provide web-based URL for uploading the document. This can be done through curl utility.

[Curl/wget utilities can be used for uploading data to Solr in your environment. They are command line based; you can also use the FireCURL plugin to upload data through your Firefox browser.]

Simple post tool is a command-line tool for uploading the raw data to Apache Solr. You can simply run it on any file or type in your input through **STDIN** to load it in Apache Solr.

**SolrJ**

SolrJ or (SolrJava) is a tool that can be used by your Java-based application to connect to Apache Solr for indexing. It provides a user-friendly interface hiding connection details from consumer application. Using SolrJ, you can index your documents and perform your queries. There are two major ways to do so; one is using the **EmbeddedSolrServer** interface. If you are using Solr in an embedded application, this is the recommended interface suited for you. It does not use HTTP-based connection. The other way is to use the **HTTPSolrServer** interface, which talks with Solr server through HTTP protocol. This is suited if you have a remote client-server based application. You can use **ConcurrentUpdateSolrServer** for bulk uploads whereas **CloudSolrServer** for communicating with Solr running in a cloud setup.
In analyzing and querying your data, we have already seen how Apache Solr effectively uses different request handlers to provide consumers with extensive ways of getting search results. Each request handler uses its own query parser, which extracts the parameters and their values from the query string, and forms the Lucene query objects. Standard query parser allows greater precision over search data; DisMaxQueryParser and ExtendedDisMaxQueryParser provide a Google-like searching syntax while searching. Depending upon which request handler is called, the query syntax is changed. Let's look at some of the important terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>q?&lt;string&gt;</td>
<td>Can support wildcard (<em>:</em>), for example, title:Scaling*</td>
</tr>
<tr>
<td>fl=id,book-name</td>
<td>Field list that a search response will return</td>
</tr>
<tr>
<td>sort=author asc</td>
<td>Results/facets to be sorted on authors in ascending order</td>
</tr>
<tr>
<td>price[* TO 100]&amp;rows=10&amp;start=5</td>
<td>Limits the result to 10 rows at a time, starting at fifth matched result</td>
</tr>
<tr>
<td>hl=true&amp;hl.fl=name,features</td>
<td>Enables highlighting on field list name and features</td>
</tr>
<tr>
<td>&amp;q=<em>:</em>&amp;facet=true&amp;facet.field=year</td>
<td>Enables faceted search on field year</td>
</tr>
<tr>
<td>Publish-date:[NOW-1YEAR/DAY TO NOW/DAY]</td>
<td>Published date between last year (same day) until today</td>
</tr>
<tr>
<td>description:&quot;Java sql&quot;-10</td>
<td>Called proximity search. Searches for the descriptions containing Java and sql in a single document with a proximity of 10 words maximum</td>
</tr>
<tr>
<td>&quot;open jdk&quot; NOT &quot;Sun JDK&quot;</td>
<td>Searches for the open jdk term in the document</td>
</tr>
<tr>
<td>&amp;q=id:938099893&amp;mlt=true</td>
<td>Searches for a specific ID, and also searches for similar results (more like this)</td>
</tr>
</tbody>
</table>

**Summary**

We have gone through various details of Apache Solr in this chapter. We reviewed the architecture, the configuration, the data loading, and its features. In the next chapter, we will look into how you can bring the two worlds of Apache Solr and Apache Hadoop together to work with Big Data.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

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