

# Search smarter with Apache Solr, Part 1: Essential features and the Solr schema

## Indexing, searching, and faceted browsing with Solr

Skill Level: Intermediate

[Grant Ingersoll \(solr@grantingersoll.com\)](mailto:solr@grantingersoll.com)

Senior software engineer

Center for Natural Language Processing at Syracuse University

29 May 2007

Solr is an enterprise-ready, Lucene-based search server that supports faceted searching, hit highlighting, and multiple output formats. In this two-part article, Lucene Java™ committer Grant Ingersoll introduces Solr and shows you how to easily incorporate its impressive full-text search functionality into your Web applications.

Connecting your users with the content they need when they need it isn't just optional anymore. With the rise of Google and similarly sophisticated search engines, users expect high-quality search results that help them find what they're looking for quickly and easily. Your manager has equally high expectations for your online shopping site -- a scalable, highly available, easy-to-maintain search solution that doesn't cost a fortune to install. As for you, you just want to advance your career, keep your employer and customers happy, and hold on to your sanity.

You can meet all these needs with Apache Solr, an open source, Lucene Java-based search server that is easy to incorporate into your Web applications. Solr offers faceted searching, hit highlighting, and support for multiple output formats, including XML/XSLT and JSON. It is easy to install and configure and comes with an HTTP-based administration interface. You can stick with Solr's basic search functionality, which is impressive, or extend it to meet the needs of your enterprise. Solr also has a vibrant developer community that you can call on for help if you need it.

This two-part article introduces Solr, demonstrates its features, and shows you how

to fully incorporate it into an example Web application. I'll start with a basic introduction to Solr, including installation and configuration instructions. I'll then introduce an example application (a blogging interface) that you can use to familiarize yourself with Solr's features. You'll learn how to use Solr to index and search content and explore Solr's support for faceted browsing. I'll conclude Part 1 with a look at Solr's schema and an explanation of how it is configured for the example application's index structure.

### Solr's history

Solr was first developed at CNET Networks and donated to the Apache Software Foundation in early 2006 under the Lucene top-level project umbrella. During its incubation period, which ended in January 2007, Solr steadily accumulated features and attracted a robust community of users, contributors, and committers. Done with incubation, Solr is now a subproject of [Lucene](#), Apache's Java-based full-text search engine library.

## Installation and configuration

You must have the following software installed to get started with Solr:

- [Java 1.5](#) or higher.
- [Ant 1.6.x](#) or higher.
- A Web browser, which you will use to view the administration pages. [Firefox](#) is recommended; your mileage may vary with Internet Explorer.
- A servlet container such as [Tomcat 5.5](#). The examples in this article assume you have Tomcat running on port 8080, which is the default for Tomcat. If you are running a different servlet container or if you're running on a different port, you may have to alter the URLs provided to access the sample application and Solr. I have assumed all parts of the sample application are running on Tomcat's localhost. Also note that Solr comes packaged with Jetty.

See [Resources](#) to download and install any of these applications.

### Setting up Solr

Once you have the prerequisites installed, download Solr version 1.1 from the [Apache Mirrors Web site](#). Next, do the following:

1. Stop your servlet container.
2. From the command line, in the directory you wish to work in, `mkdir`

- ```
dw-solr
```
3. `cd dw-solr`
  4. Copy the Solr release download into the current directory and unzip it. This will create the `apache-solr-1.1.0-incubating` directory. Pay no attention to the incubating tag; Solr has emerged from incubation.
  5. Copy the Solr WAR (located in `apache-solr-1.1.0-incubating/example/webapps/solr.war`) file to your servlet container's webapps directory.
  6. [Download the sample application](#), copy it into the current directory, and unzip it, which will create a `solr` directory in the current working directory. This will be the Solr home directory used throughout the article.
  7. You can set the Solr home location one of three ways:
    - Set the java system property `solr.solr.home` (yes, that is `solr.solr.home`).
    - Configure a JNDI lookup of `java:comp/env/solr/home` to point to the `solr` directory.
    - Start the servlet container in the directory containing the `solr` directory. (The default Solr home is `solr` under the current working directory.)
  8. `cd solr`
  9. Create the sample WAR file: `ant war`.
  10. Copy the sample WAR file (located in `dw-solr/solr/dist/dw.war`) to your servlet container's webapps directory, just as you did for the Solr WAR file. The Java code for the WAR file is located in `dw-solr/solr/src/java` and the JSPs and other Web files are located in `dw-solr/solr/src/webapp`.
  11. To verify everything is working correctly, start up your servlet container and point your browser to <http://localhost:8080/solr/admin/>. If all goes well, you should see a page similar to the one shown in Figure 1. If the admin page doesn't come up, check your container's logs for error messages. Also, make sure you started your servlet container from the `dw-solr` directory so that the Solr home location is properly set.

**Figure 1. An example of a Solr admin screen**

**About Lucene**  
 Getting to know Solr means becoming familiar with Lucene. Lucene is a Java-based, high performance text search engine library originally written by Doug Cutting and donated to the Apache Software Foundation. Many applications utilize Lucene to power their search capabilities because of its speed, ease of use, and active community. Solr builds on top of these capabilities to make Lucene ready for the enterprise with minimal programming required. See [Resources](#) to learn more about Lucene.

## Solr basics

Because Solr wraps and extends Lucene, it uses much of the same terminology. More importantly, indexes created by Solr are completely compatible with the Lucene search engine library. With proper configuration, and in some cases a bit of coding, Solr can read and use indexes built into other Lucene applications. Furthermore, many Lucene tools like [Luke](#) work just as well on indexes created by Solr.

In Solr and Lucene, an index is built of one or more `Documents`. A `Document` consists of one or more `Fields`. A `Field` consists of a name, content, and metadata telling Solr how to handle the content. For instance, `Fields` can contain strings, numbers, booleans, or dates, as well as any types you wish to add. A `Field` can be described using a number of options that tell Solr how to treat the content during indexing and searching. I'll discuss these options in greater detail later in the article. For now, take a look at the subset of important attributes listed in Table 1:

**Table 1. Field attributes**

| Attribute name | Description                                                                                                                                                                                                                                                                                                  |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>indexed</b> | Indexed <code>Fields</code> are searchable and sortable. You also can run Solr's analysis process on indexed <code>Fields</code> , which can alter the content to improve or change results. The following section provides more information about Solr's analysis process.                                  |
| <b>stored</b>  | The contents of a stored <code>Field</code> are saved in the index. This is useful for retrieving and highlighting the contents for display but is not necessary for the actual search. For example, many applications store pointers to the location of contents rather than the actual contents of a file. |

## About Solr's analysis process

You can run Solr's analysis process to modify application content before indexing it. In Solr and Lucene, an `Analyzer` consists of a `Tokenizer` and one or more `TokenFilters`. A `Tokenizer` is responsible for producing `Tokens`, which in most cases correspond to words to be indexed. A `TokenFilter` takes in `Tokens` from the `Tokenizer` and can modify or remove a `Token` before indexing. For instance, Solr's `WhitespaceTokenizer` breaks words on whitespace, and its `StopFilter` removes common words from search results. Other types of analysis include stemming, synonym expansion, and case folding. Chances are, if you need analysis done in a particular way for your application, Solr has one or more tokenizers and filters to meet your needs.

You can also apply analysis to a query during a search operation. As a general rule, you should run the same analysis on a query as on the document to be indexed. Users new to these concepts commonly make the mistake of stemming document tokens but not query tokens, which often results in zero search matches. Solr's XML configuration makes it easy to create `Analyzers` using simple declarations, as I show later in the article.

See [Resources](#) to learn more about Solr and Lucene's analysis tools, as well as index structures and other capabilities.

## The example application

The following sections use a realistic example application to introduce you to Solr's functionality. The example application is a Web-based blogging interface that allows you to log entries, assign metadata to the entries, and then index and search the entries. At each step of the indexing and searching process, you have the option to display the commands being sent to Solr.

To see the sample application, point your browser to

<http://localhost:8080/dw/index.jsp>. If you have everything set up correctly (as described in "[Setting up Solr](#)"), you see a simple user interface entitled "Sample Solr Blog Search" with some menu items located directly below the title. You will explore all the topics in the menu as you proceed through both parts of this article.

## Indexing operations

In Solr, you initiate indexing and searching by sending HTTP requests to the Solr Web application deployed in a servlet container. Solr takes in the request, determines the appropriate `SolrRequestHandler` to use, and then processes the request. Responses are returned over HTTP in the same way. The default configuration returns Solr's standard XML response. You can also configure Solr for alternate response formats. I'll show you how to customize request and response handling in the second half of this article.

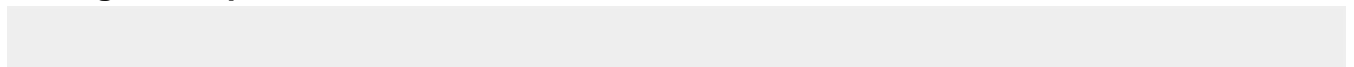
Indexing is the process of taking input (in this case blog entries, keywords, and other metadata) and passing them to Solr to index through `HTTP Post XML` messages. You can pass four different indexing requests to the Solr indexing servlet:

- **add/update** allows you to add or update a document to Solr. Additions and updates are not available for searching until a commit takes place.
- **commit** tells Solr that all changes made since the last commit should be made available for searching.
- **optimize** restructures Lucene's files to improve performance for searching. Optimization is generally good to do when indexing has completed. If there are frequent updates, you should schedule optimization for low-usage times. An index does not need to be optimized to work properly. Optimization can be a time-consuming process.
- **delete** can be specified by id or by query. Delete by id deletes the document with the specified id; delete by query deletes all documents returned by a query.

### An indexing example

Browse to <http://localhost:8080/dw/index.jsp> to look at the indexing process in greater detail. Start by filling in the form with the appropriate entries for each field and press the **Submit** button. The sample application takes in the entries, creates a Solr request, and displays it for viewing on the next screen. Listing 1 contains a sample of the `add` command that is sent to Solr when you press the **Submit** button.

#### Listing 1. Sample Solr add command



```
<add>
  <doc>
    <field name="url">http://localhost/myBlog/solr-rocks.html</field>
    <field name="title">Solr Search is Simply Great</field>
    <field name="keywords">solr,lucene,enterprise,search</field>
    <field name="creationDate">2007-01-06T05:04:00.000Z</field>
    <field name="rating">10</field>
    <field name="content">Solr is a really great open source search server. It scales,
    it's easy to configure and the Solr community is really supportive.</field>
    <field name="published">on</field>
  </doc>
</add>
```

Each field entry in the `<doc>` in Listing 1 tells Solr what `Fields` should be added to the Lucene index for the document you created. It is possible to add more than one `<doc>` to the `add` command. I'll explain later how Solr handles these fields. For now, it is enough to know that one document containing the seven fields specified in Listing 1 will be indexed.

When you submit the command from the "Solr XML Command" page, the results are sent to Solr to be processed. An `HTTP POST` sends the command to the Solr Update Servlet running at <http://localhost:8080/solr/update>. If all goes well, an XML document is returned with `<result status="0"/>`. Solr automatically updates documents with the same URLs (the URL in the sample application is the unique id Solr uses to identify whether a document has been added before or not).

## Practice indexing

Now add a few more documents and commit them so that you will have some documents to search in the next section. Once you are comfortable with the syntax of the `add` command, you can uncheck the "Display XML..." check box near the **Index** button to skip the "Solr XML Command" page. The [sample files](#) included with this article contain a version of the index used in many of these examples.

You can delete a document by entering its URL on the <http://localhost:8080/dw/delete.jsp> page, submitting and viewing the command, and then submitting the command to Solr. See the "Solr Wiki" reference in [Resources](#) to learn more about indexing and deletion commands.

## Search commands

Now that you have added a few documents, you can search them. Solr accepts both `HTTP GET` and `HTTP POST` messages for queries. Incoming queries are processed by the appropriate `SolrRequestHandler`. For the purposes of this discussion, you will use the `StandardRequestHandler`, which is the default. In the second part of this article, I'll show you how to configure Solr for other `SolrRequestHandlers`.

To see searching in action, return to the example application and browse to <http://localhost:8080/dw/searching.jsp>. This screen should look very similar to the

Indexing screen, with the addition of several options related to searching. Similar to indexing, you can enter values into the various input fields, select search parameters, and submit the query to the sample application. The sample application highlights some of the more common query parameters available in Solr. They are as follows:

### A note on query syntax

Solr query syntax for the `StandardRequestHandler` is the same as that supported by the `Lucene QueryParser`, plus some added support for sorting. The sample application does little validation of the values entered and does not demonstrate capabilities like boosting, phrases, range filters, etc., all of which are valid in Solr and Lucene. See [Resources](#) for more information about the `Lucene QueryParser`. In the second part of this article, I will introduce some tools in the administration interface that help debug query syntax and results.

- **Boolean operator:** By default, the boolean operator used to combine search terms is `OR`. Setting it to `AND` requires that all terms appear in a matched document.
- **Number of results:** Specifies the maximum number of results to return.
- **Start:** The offset to start at in the result set. This is useful for pagination.
- **Highlight:** Highlights terms that match in the fields of the document. Refer to the node named `<lst>` toward the bottom of [Listing 2](#). Highlighted terms are marked by `<em>`.

Once you have entered and submitted the values, the blogging application returns a query string that is ready for submission to Solr. Once submitted, if everything is correct and there are matching documents, Solr returns an XML response containing the results, highlighting information, and some metadata about the query. Listing 2 contains a sample search result:

### Listing 2. Example of a search result

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">6</int>
    <lst name="params">
      <str name="rows">10</str>
      <str name="start">0</str>
      <str name="fl">*,score</str>
      <str name="hl">true</str>
      <str name="q">content:"faceted browsing"</str>
    </lst>
  </lst>
  <result name="response" numFound="1" start="0" maxScore="1.058217">
    <doc>
      <float name="score">1.058217</float>
      <arr name="all">
        <str>http://localhost/myBlog/solr-rocks-again.html</str>
```

```

    <str>Solr is Great</str>
    <str>solr,lucene,enterprise,search,greatness</str>
    <str>Solr has some really great features, like faceted browsing
    and replication</str>
  </arr>
  <arr name="content">
    <str>Solr has some really great features, like faceted browsing
    and replication</str>
  </arr>
  <date name="creationDate">2007-01-07T05:04:00.000Z</date>
  <arr name="keywords">
    <str>solr,lucene,enterprise,search,greatness</str>
  </arr>
  <int name="rating">8</int>
  <str name="title">Solr is Great</str>
  <str name="url">http://localhost/myBlog/solr-rocks-again.html</str>
</doc>
</result>
<lst name="highlighting">
  <lst name="http://localhost/myBlog/solr-rocks-again.html">
    <arr name="content">
      <str>Solr has some really great features, like <em>faceted</em>
      <em>browsing</em> and replication</str>
    </arr>
  </lst>
</lst>
</response>

```

A query message can contain a number of parameters, the highlights of which are described in Table 2. See the "Solr Wiki" reference in [Resources](#) for a complete listing.

**Table 2. Highlighted query parameters**

Parameter	Description	Example
<b>q</b>	The query to search with in Solr. See "Lucene QueryParser Syntax" in <a href="#">Resources</a> for a full description of the syntax. Sorting information can be included by appending a semi-colon and the name of an indexed, non-tokenized field (explained below). The default sort is <code>score desc</code> , which means sort by descending score.	<code>q=myField:Java AND otherField:developerWorks; date asc</code> This query searches the two fields specified and sorts the results based on a date field.
<b>start</b>	Specifies the starting offset into the result set. Useful for paging through results. The default value is 0.	<code>start=15</code> Returns results starting with the fifteenth ranked result.
<b>rows</b>	The maximum number of documents to return. The default value is 10.	<code>rows=25</code>
<b>fq</b>	Provide an optional filtering query. Results of the query	Any valid query that could be passed in the <code>q</code> parameter,

	are restricted to searching only those results returned by the filter query. Filtered queries are cached by Solr. They are very useful for improving the speed of complex queries.	not including sort information.
<b>hl</b>	When <code>hl=true</code> , highlight snippets in the query response. Default is false. See the Solr Wiki section on highlighting parameters for more options (in <a href="#">Resources</a> ).	<code>hl=true</code>
<b>fl</b>	Specify as a comma-separated list the set of <code>Fields</code> that should be returned in the document results. "*" is the default and means all fields. "score" indicates the score should be returned as well.	<code>*,score</code>

## Faceted browsing

Lately, it seems all the popular shopping sites have been adding those handy lists of criteria that help users narrow down results by manufacturer, price, or author. These lists are the result of faceted browsing, which is a way of classifying results that have already been returned into meaningful categories that are guaranteed to exist. Facets are used to help users narrow down search results.

You can see facets in action by browsing to <http://localhost:8080/dw/facets.jsp>. On this page, you see two input forms of interest:

- A text area where you can enter a query to be issued against the `all` field in the index. Think of the `all` field as a concatenation of all the other fields that you have indexed. (More on this later.)
- A drop-down list of fields that can be used for faceting. Not all indexed fields are listed here, for reasons that will be explained momentarily.

Enter a query and pick a facet field from the drop-down list and then click **Submit** to pass along the generated query to Solr. The blogging application parses the results and returns results similar to those in Figure 2:

### Figure 2. Sample faceted browsing results

Facets: [search \(2\)](#) [solr \(2\)](#) [caching \(2\)](#) [sorting \(1\)](#) [replication \(1\)](#) [features \(1\)](#) [enterprise \(2\)](#)

- Title: Solr  
 URL: http://localhost/myBlog/solr.html  
 Contents: Solr is used for enterprise search. It has many great features, like caching, replication, index warming and sorting  
 Rating: 8  
 Keywords: solr enterprise search caching replication sorting  
 Published: false  
 Creation Date: 2007-01-07T05:04:00.000Z
- Title: Solr Enterprise Search  
 URL: http://localhost/myBlog/solr-enterprise.html  
 Contents: Solr is useful for enterprise search. It has caching, replication and a number of other useful features  
 Rating: 8  
 Keywords: solr enterprise search caching features  
 Published: false  
 Creation Date: 2007-04-08T04:00:00.000Z

In Figure 2, you see facet counts for all non-zero values displayed across the top and the two search results that matched the submitted query below. Clicking a facet link in the example submits the original query, plus the *Facet* keyword as a new keyword. If the original query was `q=Solr` and the facet field was `keywords` and you clicked the `replication` link in Figure 2, the new query would be `q=Solr AND keywords:replication`.

Faceting does not have to be turned on or configured in Solr to work, but it may require you to index your application content in new ways. Faceting is done on indexed fields and works best on non-tokenized words that are not lower-cased. (This is why I did not include the `content` field or some of the other fields added to documents in the Facet Field drop-down list.) Facet fields usually do not need to be stored because the whole point of faceting is to display values that are human-readable.

Note also that Solr does not create the categories in the facets; they must be added by the application itself during indexing, just as you assigned keywords to the documents when indexing the application. Solr does provide the logic for figuring out what the facets are and their counts, given a facet field.

## The Solr schema

Up to now, I have introduced you to Solr's features without actually explaining how they are configured. The remainder of the article focuses on configuration, first introducing the Solr schema (`schema.xml`) and then showing you how it relates to Solr's features.

In an editor, preferably one that supports XML tag highlighting, open the `schema.xml` file located in `<INSTALL_DIR>/dw-solr/solr/conf`. The first thing to notice is the multitude of comments. If you have used open source software before, you will be very appreciative of the documentation in the schema file and in Solr as a whole. Because `schema.xml` is so well commented, I will focus on some of the key attributes of the file and leave the details to the documentation. First off, notice the name of the schema (`dw-solr`) in the `<schema>` tag. Solr supports one schema per

deployment. In the future, it may support multiple schemas, but for now, only one schema is allowed. (See the "Solr Wiki" reference in [Resources](#) to learn how to easily configure Tomcat and other containers for multiple deployments per container.)

The schema can be organized into three sections:

- Types
- Fields
- Other declarations

In the `<types>` section are common, reusable definitions of how `Fields` should be processed by Solr (and Lucene). In the example schema, there are 13 field types, ranging, by name, from `string` to `text`. The field types declared at the top of the `<types>` section, like `sint` and `boolean`, are used to store primitive types in Solr. For the most part, Lucene only deals with strings, so integers, floats, dates, and doubles require special handling to be searchable. Using these field types alerts Solr to treat the content to be indexed with the appropriate special handling, requiring no intervention on your part.

## Types

[Earlier in this article](#), I touched on the basics of Solr's analysis process. Taking a closer look at the `text` field type declaration, you can see the details of just how Solr manages analysis. Listing 3 shows the `text` field type declaration:

### Listing 3. Declaration for the text field type

```
<fieldtype name="text" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"/>
    <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
      generateNumberParts="1" catenateWords="1"
      catenateNumbers="1" catenateAll="0"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPorterFilterFactory"
      protected="protowords.txt"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
      ignoreCase="true" expand="true"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"/>
    <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
      generateNumberParts="1" catenateWords="0"
      catenateNumbers="0" catenateAll="0"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPorterFilterFactory" protected="protowords.txt"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
```

```
</fieldtype>
```

### The class attribute

In many instances in the Solr schema, the class attribute is abbreviated to something like `solr.TextField`. This is simply shorthand for `org.apache.solr.schema.TextField`. As you will see in the second part of this article, any valid class in the classpath that extends the `org.apache.solr.schema.FieldType` class may be used.

First, note that I have declared two different `Analyzers` in Listing 3. While the `Analyzers` are not exactly the same for both indexing and searching, they really only differ by the addition of synonyms during query analysis. Stemming, stopword removal, and similar operations are all still applied to the tokens before indexing or searching, resulting in the same types of tokens to be used. Next, notice that I declared the tokenizer first and then any filters used. The Solr configuration for the example application is set as follows:

- Tokenize on whitespace, then removed any common words (StopFilterFactory)
- Handle special cases with dashes, case transitions, etc. (WordDelimiterFilterFactory; lowercase all terms (LowerCaseFilterFactory))
- Stem using the Porter Stemming algorithm (EnglishPorterFilterFactory)
- Remove any duplicates (RemoveDuplicatesTokenFilterFactory)

This example analysis incorporates many common approaches to improving search results but should not be seen as the only way to analyze text. Each application may have its own analysis needs that are not covered by this example or even any of the existing `Analyzers` in Solr or Lucene. See "More Info On Solr Analysis" in [Resources](#) for more options related to analysis, as well as information on how to use other `Analyzers`.

## Fields

Continuing on to the `<fields>` section of the schema, you can see how Solr handles the eight (seven, really, plus `all`) fields used during indexing and searching. These fields are repeated in Listing 4:

### Listing 4. Declared fields for the blogging application

```
<field name="url" type="string" indexed="true" stored="true"/>
<field name="title" type="text" indexed="true" stored="true"/>
<field name="keywords" type="text_ws" indexed="true" stored="true"
      multiValued="true" omitNorms="true"/>
```

```
<field name="creationDate" type="date" indexed="true" stored="true"/>
<field name="rating" type="sint" indexed="true" stored="true"/>

<field name="published" type="boolean" indexed="true" stored="true"/>

<field name="content" type="text" indexed="true" stored="true" />
<!-- catchall field,
containing many of the other searchable text fields
      (implemented via copyField further on in this schema) -->
<field name="all" type="text" indexed="true" stored="true" multiValued="true"/>
```

Understanding field types, you can see clearly how each of the fields is processed. For instance, the `url` field is an indexed, stored, unanalyzed `string` field. Meanwhile, the `text` field is analyzed using the `Analyzer` declared in [Listing 3](#). What about the `all` field? The `all` field is a `text` field just like `title` or `content`, but it contains the contents of several fields concatenated together to facilitate alternate search mechanisms (remember that the faceted search used the `all` field).

As for attributes on the fields, you learned the meaning of `indexed` and `stored` in [Table 1](#). The `multiValued` attribute is a special case, meaning a `Document` can have a `Field` with the same name added more than once. In our sample, we could have added `keywords` more than once, for example. The `omitNorms` attribute tells Solr (and Lucene) not to store norms. Omitting norms is useful for saving memory on `Fields` that do not affect scoring, such as those used for calculating facets.

Before finishing up the `<fields>` section, a brief word about the `<dynamicField>` declarations located below the field declarations. Dynamic fields are special kinds of fields that can be added to any document at any time with the attributes defined by the field declaration. The key difference between a dynamic field and a regular field is that dynamic fields do not need to have a name declared ahead of time in the `schema.xml`. Solr applies the glob-like pattern in the name declaration to any incoming field name not already declared and processes the field according to the semantics defined by its `<dynamicField>` declaration. For instance, `<dynamicField name="*_i" type="sint" indexed="true" stored="true"/>` means that a field named `myRating_i` would be treated as an `sint` by Solr, even though it wasn't declared as a field. This is convenient, for example, when letting users define the content to be searched.

## Other declarations

Finally, the last part of the `schema.xml` file contains various declarations related to fields and querying. The most important declaration is the `<uniqueKey>url</uniqueKey>`. This tells Solr that the `url` field declared earlier is the unique identifier that is used to determine when a document is being added or updated. The `defaultSearchField` is the `Field` Solr uses in queries when no field is prefixed to a query term. The example uses queries that look like `q=title:Solr`. If you entered `q=Solr` instead, the default search field would

apply. The end result is equivalent to `q=all:Solr` because `all` is the default search field in the blogging application.

The `<copyField>` mechanism lets you create the `all` field without manually adding all the content of your document to a separate field. Copy fields are convenient ways to index the same content in multiple ways. For instance, if you wanted to provide both exact matching respecting case and matching ignoring case, you could use a copy field to automatically analyze the incoming content. Content would then be indexed exactly as received, with all letters in lowercase.

## Next up: Solr for the enterprise

So far, you have installed Solr and learned how to use it to index and search documents in an example application. You've also seen how faceted browsing works in Solr and learned how to declare an index structure through Solr's `schema.xml` file. The [example application](#) included with this article demonstrates these capabilities and shows how to format commands for Solr.

In the second part of the article, I will introduce the features that extend Solr beyond a simple searching interface into being an enterprise-ready search solution. You will learn about Solr's administration interfaces and advanced configuration options, as well as performance-related features such as caching, replication, and logging. I will also briefly discuss the ways you can extend Solr to meet the needs of your enterprise. In the meantime, enjoy the sample application and use it to familiarize yourself with Solr's basic functionality.

## Downloads

Description	Name	Size	Download method
Sample Solr application	j-solr1.zip	500KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- ["Beef up Web search applications with Lucene"](#) (Deng Peng Zhou, developerWorks, August 2006): Learn more about the Lucene search library, which serves as the basis for Solr.
- ["Parsing, indexing, and searching XML with Digester and Lucene"](#) (Otis Gospodneti, developerWorks, June 2003): An early look at Lucene.
- [Solr homepage](#): Explore tutorials, browse Javadocs, and keep up with the Solr community.
- [The Solr Wiki](#): See the Wiki for many documents on the workings of Solr.
- [Solr analysis](#): Learn more about how Solr's analyzers, tokenizers, and token filters work.
- [Lucene QueryParser Syntax](#): Learn more about Solr's (and Lucene's) query parser syntax.
- [The Porter Stemming Algorithm](#): Learn more about the stemming algorithm used by Solr.
- [Public Websites using Solr](#): A listing of Web sites that use Solr's functionality.
- [Lucene Java home](#): Explore Solr's ancestry.
- [Lucene In Action](#) (Otis Gospodneti and Erik Hatcher; Manning, 2004): A must-read for anyone interested in Lucene.
- [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

## Get products and technologies

- [Apache Mirrors](#): Download Solr 1.1 or the latest release.
- [Download Tomcat](#): See the [Solr Tomcat](#) section of the Solr Wiki for specific details related to running Solr and Tomcat together.
- [Download Ant](#).
- [Download Firefox](#).
- [Download JDK 1.5](#).
- [Get Luke](#): A very handy tool for examining the contents of a Lucene index. Consult Luke when you have questions about what is in an index or why a query isn't working.
- [Download curl](#): For help submitting HTTP requests from the command line.

## Discuss

- [Solr Mailing Lists](#): Become part of the Solr community.

## About the author

Grant Ingersoll

Grant Ingersoll is a senior software engineer at the Center for Natural Language Processing at Syracuse University. Grant's programming interests include information retrieval, question answering, text categorization, and extraction. He is a committer and speaker on the Lucene Java project.

## Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.