1.4

lucidworks for Solr

CERTIFIED DISTRIBUTION REFERENCE GUIDE

version 1.4

lucid IMAGINATION

Version 1.4_01

Table of Contents

This page intentionally left blank.

1 About This Guide

1.1 LucidWorks for Solr Certified Distribution

This reference guide describes the LucidWorks for Solr Certified Distribution. This is a tested, documented release of Solr 1.4, an open source solution for search. In addition to the core software of the Apache Solr 1.4 release, the Certified Distribution includes a software installer and this reference guide.

You can download the LucidWorks for Solr Certified Distribution here:

```
http://www.lucidimagination.com/Downloads
```

1.1.1 Solr and Lucene

Solr makes it easy for programmers to develop sophisticated, high performance search applications with advanced features such as faceting (arranging search results in columns with numerical counts of key terms). Solr builds on another open source search technology—Lucene, a Java library that provides indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities. Both Solr and Lucene are managed by the Apache Software Foundation (www.apache.org).

The Lucene search library currently ranks among the top 15 open source projects and is one of the top 5 Apache projects, with installations at over 4,000 companies. Lucene/Solr downloads have grown nearly 10x over the past three years, with a current run-rate of over 6,000 downloads a day. The Solr search

server, which provides application builders a ready-to-use search platform on top of the Lucene search library, is the fastest growing Lucene sub-project. Apache Lucene/Solr offers an attractive alternative to the proprietary licensed search and discovery software vendors.

1.1.2 Lucid Imagination

Lucid Imagination is the first commercial company exclusively dedicated to Apache Lucene/Solr open source technology. This Certified Distribution of Solr 1.4 is among the first of many offerings that Lucid Imagination is bringing to the Lucene/Solr community.

1.2 About This Guide

This *Reference Guide* describes all of the important features and functions of the LucidWorks for Solr Certified Distribution. It's available free when you download the LucidWorks for Solr Certified Distribution.

Designed to provide complete, comprehensive documentation, the *Reference Guide* is intended to be more encyclopedic and less of a cookbook. It is structured to address a broad spectrum of needs, ranging from new developers getting started to well experienced developers extending their application or troubleshooting. It will be of use at any point in the application lifecycle, for whenever you need deep, authoritative information about Solr.

The material as presented assumes that you're familiar with some basic search concepts and that you can read XML. It does not assume that you are a Java programmer, although knowledge of Java is helpful when working directly with Lucene or when developing custom extensions to a Lucene/Solr installation.

Here's a summary of the contents of this guide:

- **Chapter 1: About This Guide**
  The chapter you are reading.
- **Chapter 2: Getting Started**
  This chapter guides you through the installation and set-up of the LucidWorks for Solr Certified Distribution.

- **Chapter 3: Using the Admin Web Interface**
  This chapter introduces the Solr Web interface. From your browser, you can view configuration files, submit queries, view logfile settings and Java environment settings, and monitor and control distributed configurations.

- **Chapter 4: Documents, Fields, and Schema Design**
  This chapter describes how Solr organizes its data for indexing. It explains how a Solr schema defines the fields and field types which Solr uses to organize data within the document files it indexes.

- **Chapter 5: Understanding Analyzers, Tokenizers, and Filters**
  This chapter explains how Solr prepares text for indexing and searching. Analyzers parse text and produce a stream of tokens, lexical units used for indexing and searching. Tokenizers break field data down into tokens. Filters perform other transformational or selective work on token streams.

- **Chapter 6: Indexing and Basic Data Operations**
  This chapter describes the indexing process and basic index operations, such as commit, optimize, and rollback.

- **Chapter 7: Searching**
  This chapter presents an overview of the search process in Solr. It describes the main components used in searches, including request handlers, query parsers, and response writers. It lists the query parameters that can be passed to Solr, and it describes features such as boosting and faceting, which can be used to fine-tune search results.

- **Chapter 8: The Well Configured Solr Instance**
  This chapter discusses performance tuning for Solr. It begins with an overview of the `solrconfig.xml` file, then tells you how to configure multiple SolrCores, how to configure the Lucene index writer, and more.

- **Chapter 9: Managing Solr**
  This chapter discusses important topics for running and monitoring Solr. It describes running Solr in the Apache Tomcat servlet runner and Web server. It also describes LucidGaze, Lucid Imagination's tool for statistical reporting about Solr. Other topics include how to back up a Solr instance, and how to run Solr with Java Management Extensions (JMX).

- **Chapter 10: Scaling and Distribution**
  This chapter tells you how to grow a Solr distribution by dividing a large index into sections called shards, which are then distributed across multiple servers, or by replicating a single index across multiple services.

- **Chapter 11: Client APIs**
  This chapter tells you how to access Solr through various client APIs, including JavaScript, JSON, and Ruby.

The manual also includes an index.

NOTE: The default port configured for LucidWorks during the install process is 8983. The samples, URLs and screenshots in this guide may show different ports, because the port number that LucidWorks uses is configurable. If you have not customized your installation of LucidWorks, please make sure that you use port 8983 when following the examples, or configure your own installation to use the port numbers shown in the examples. For information about configuring port numbers used by Tomcat or Jetty, see Chapter 9.

1.3 Further Assistance

In addition to providing this Reference Guide for the Certified Distribution of Solr, Lucid Imagination offers other helpful documentation and tips on its Web site, www.lucidimagination.com. Visit the Web site for:

- Technical Notes on special topics
- White Papers about important search topics and methodologies
- Blog posts about the latest news and events of interest to the Lucene and Solr communities
- Podcasts presenting Lucene and Solr tutorials, as well as interview with Lucene and Solr committers and customers

For more information, you can contact Lucid Imagination here:

Lucid Imagination
1875 South Grant Street
10th Floor
San Mateo, CA 94402

Tel: 650.353.4057
Fax: 650.525.1365

For support and service inquiries, please write to:

`support@lucidimagination.com`

2 Getting Started

The point of this chapter is to help you get Solr up and running quickly, and to introduce you to the basic Solr architecture and features.

2.1 Installing LucidWorks for Solr

This section describes how to install LucidWorks for Solr. You can install LucidWorks anywhere that a suitable Java Runtime Environment (JRE) is available, as detailed below. Currently this includes Linux, OS X, and Microsoft Windows. The instructions in this chapter should work for any platform, with a few exceptions for Windows as noted.

2.1.1 Got Java?

You will need the Java Runtime Environment (JRE) version 1.5 or higher, although 1.6 is highly recommended. At a command line, check your Java version like this:

```
$ java -version
java version "1.6.0_0"
IcedTea6 1.3.1 (6b12-0ubuntu6.1) Runtime Environment (build 1.6.0_0-b12)
OpenJDK Client VM (build 1.6.0_0-b12, mixed mode, sharing)
```

The output will vary, but you need to make sure you have version 1.5 or higher. If you don't have the required version, or if the `java` command is not found, download and install the latest version from Sun:

http://java.sun.com/javase/downloads/

2.1.2 Downloading the LucidWorks for Solr Installer

The installer is available here:

http://www.lucidimagination.com/Downloads

The file will have a name like `SolrInstaller.jar`.

2.1.3 Running the Installer

At a command line, go to the same directory as the installation file. Then run the installer like this:

```
$ java -jar SolrInstaller.jar
```

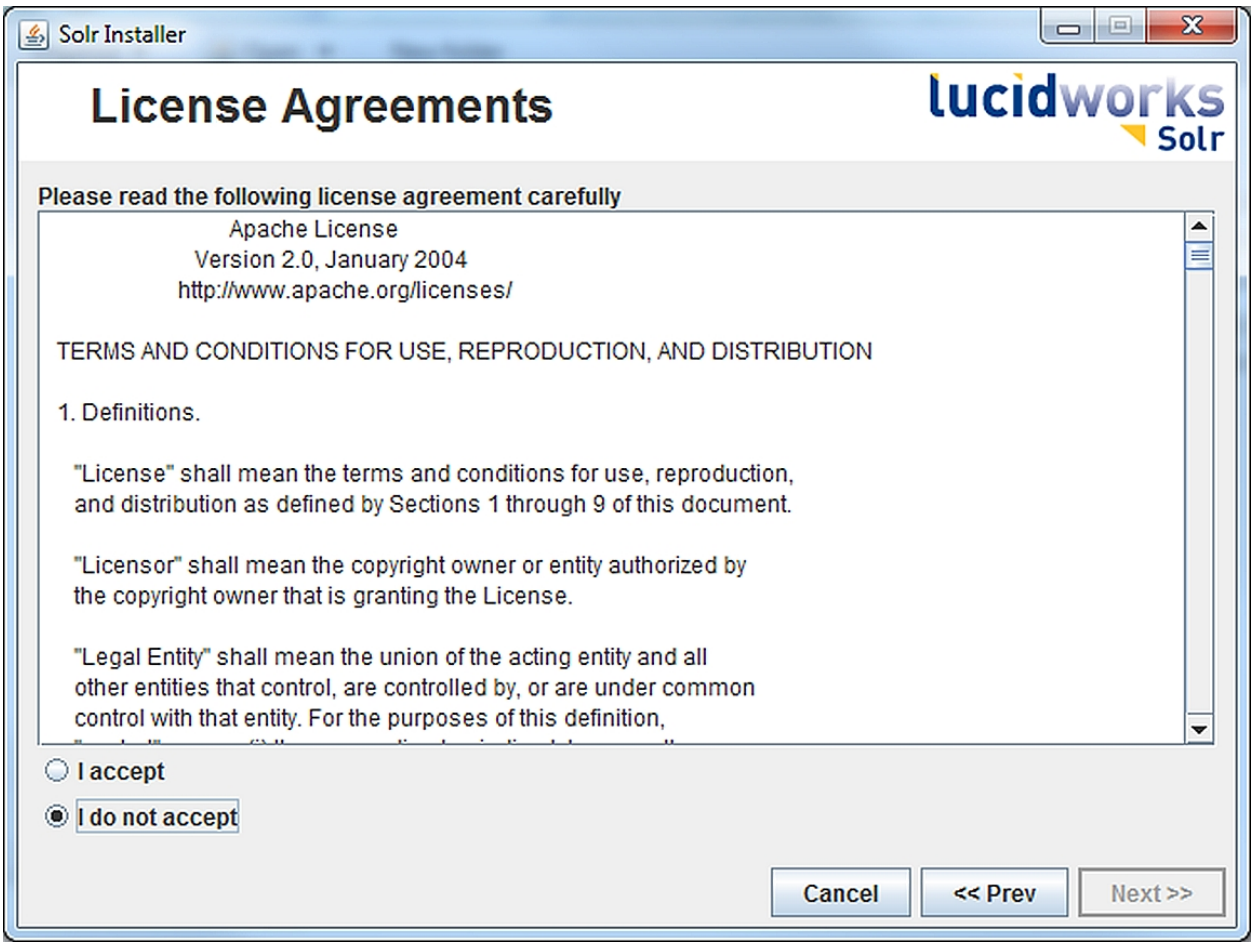
You will see the welcome screen.

For environments that do not support a graphical window system, like a headless Linux server, run the installer like this instead:

```
$ java -jar SolrInstaller.jar -console
```

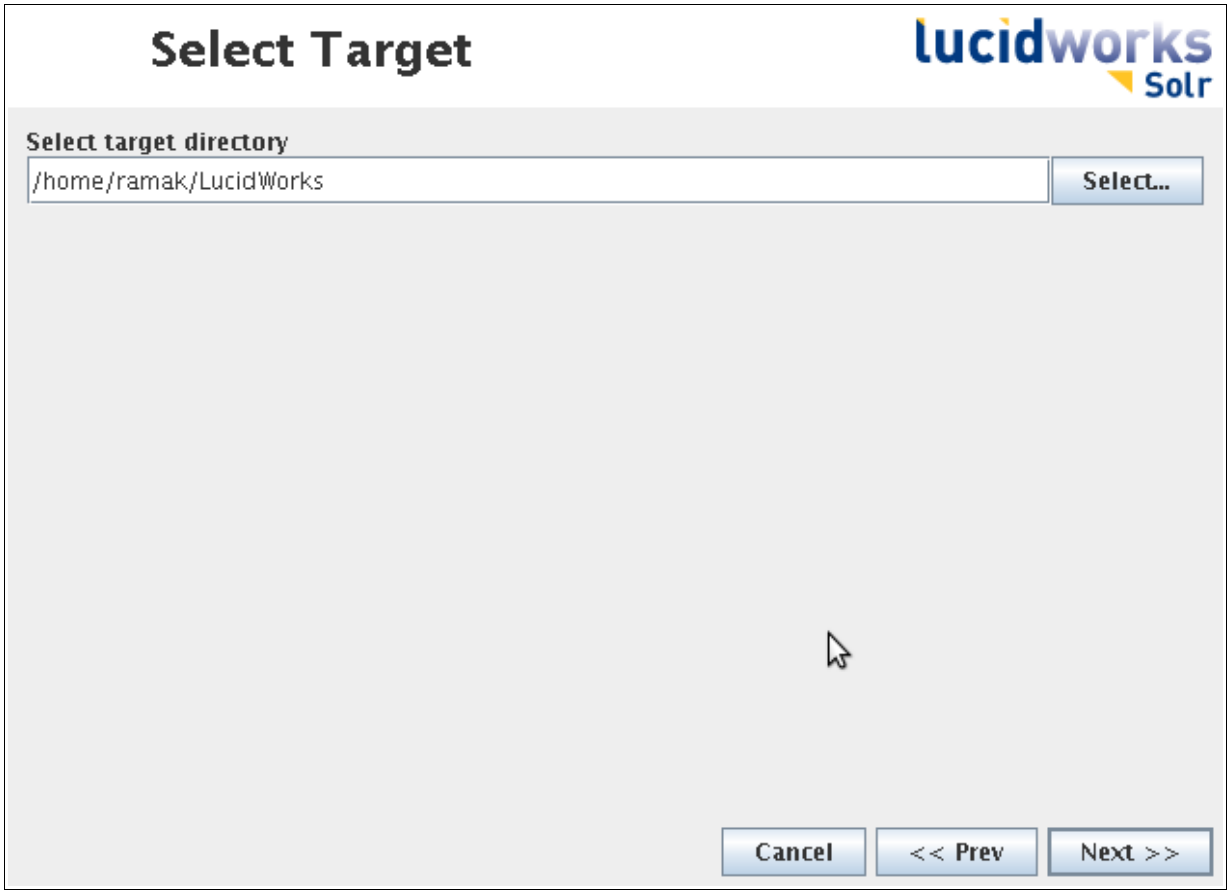
The rest of this chapter describes the graphical installer, but you can expect a similar flow from the console version.

Press **Next>>**. You will see the license agreement.



The License Agreements screen.

If you agree to the terms of the license, click **I accept** and press **Next>>**.

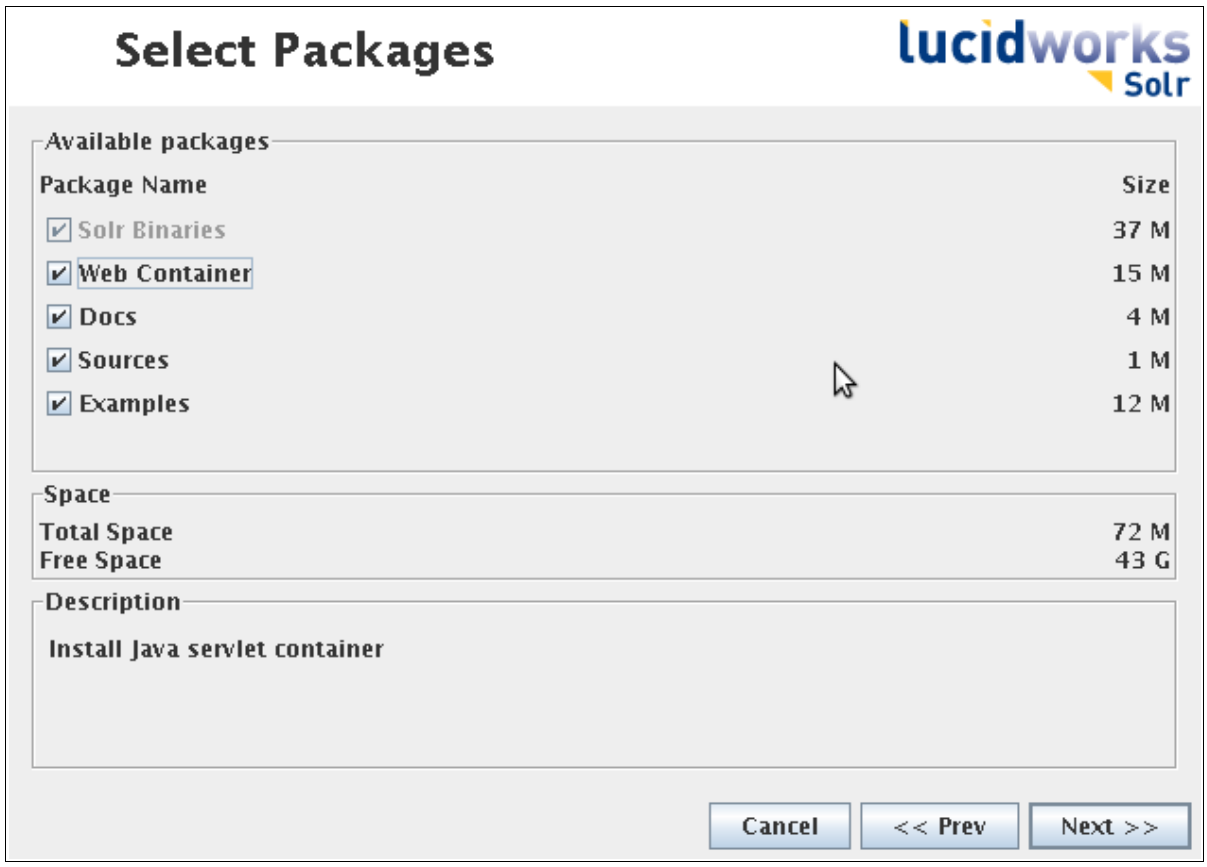


Selecting a target directory for the installation.

If you run the installer on Windows Vista, it will offer a default target installation directory under the user home directory (e.g.. `C:\users\<username>`). If you want to install LucidWorks for Solr outside of your user directory, you will have to run the installer in elevated mode.

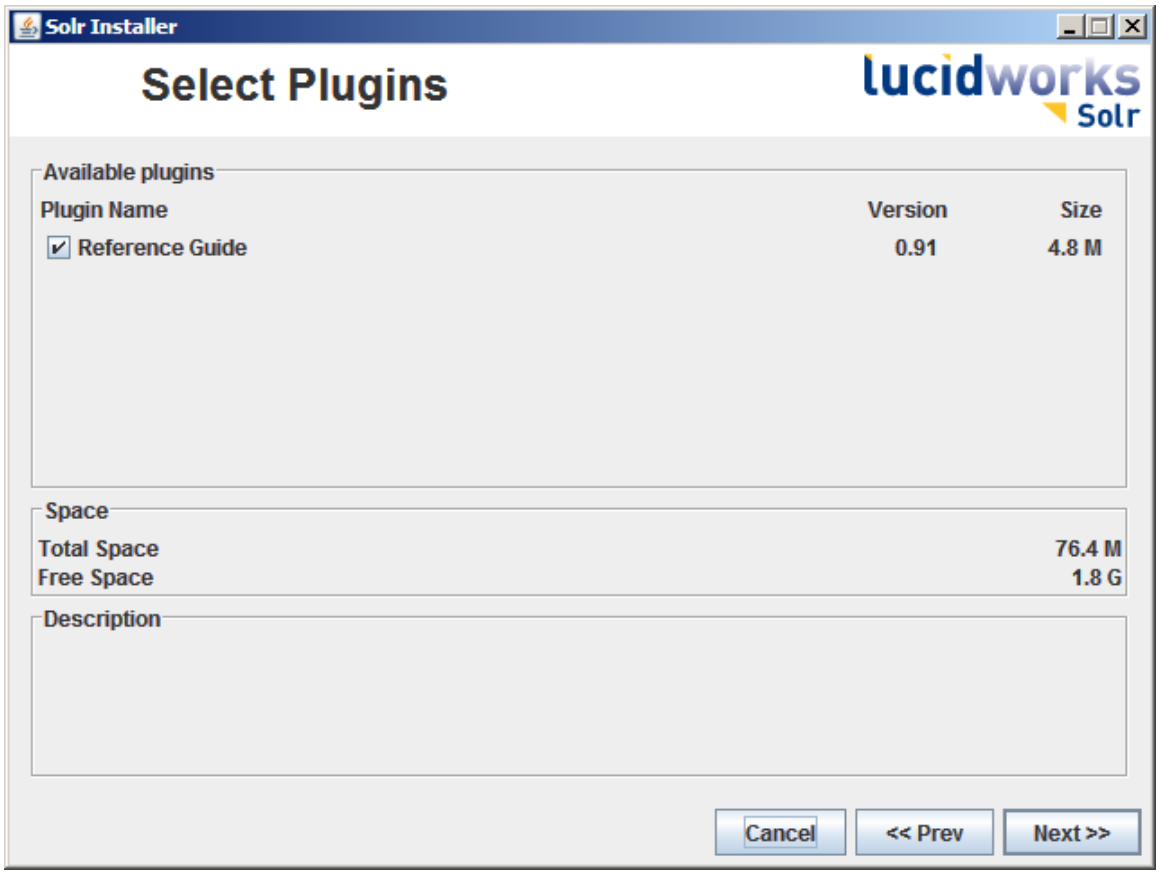
Choose where you want LucidWorks installed and press **Next>>**. The installer will create the directory if it does not already exist. If it does exist, you will get a courteous warning.

Next, you'll see the installer's package selection window.



The Select Packages screen.

Here you choose the different packages to install. By default, all packages are selected for installation. Make your choices, and press **Next>>**.

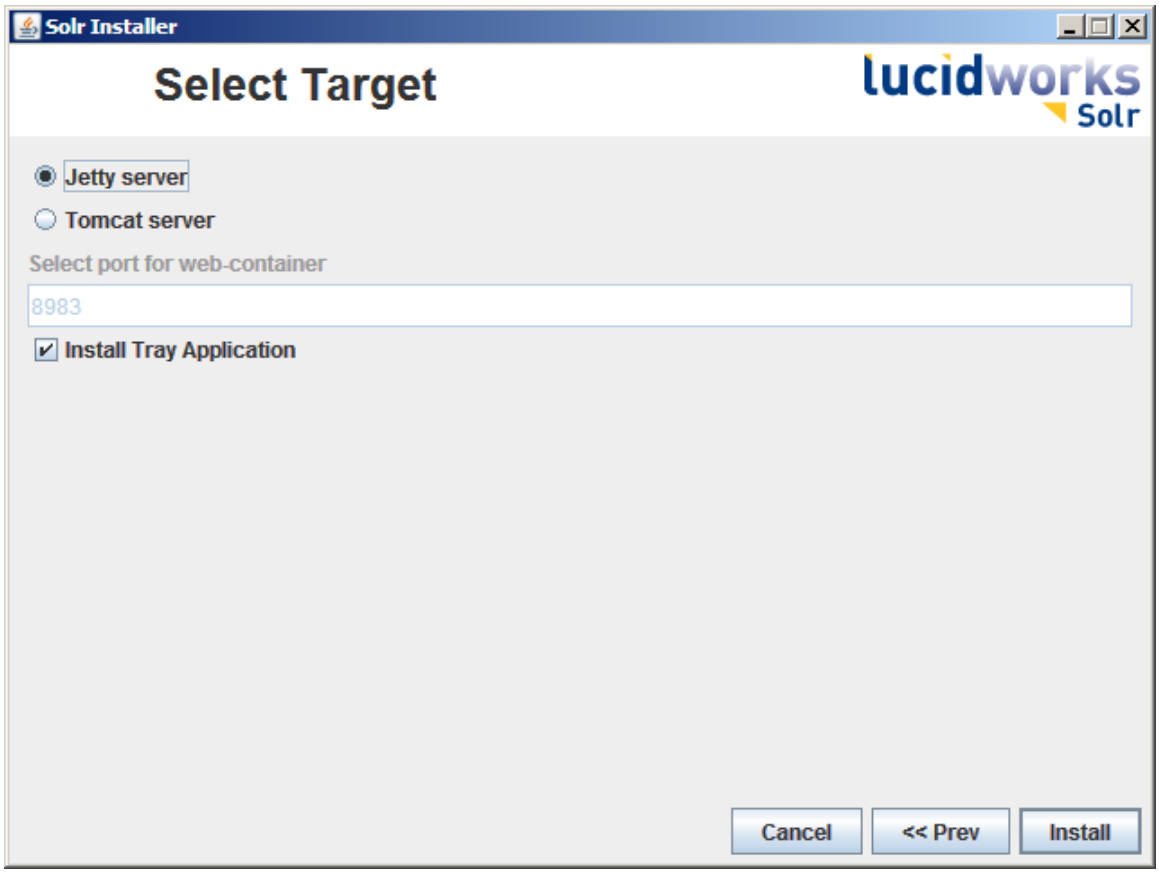


The Select Plugins window.

The installer connects to the Lucid Imagination update service to see if there are additional or updated plugins to be offered. By default, the installer checks the "public" repository for new and updated plugins. Lucid Imagination maintains additional repositories for beta customers, early adapters and paying customers. Please consult with your Lucid Imagination contacts if you would like your installer to check one of these additional repositories.

Click off the plugins you want installed with LucidWorks. Press **Next>>**.

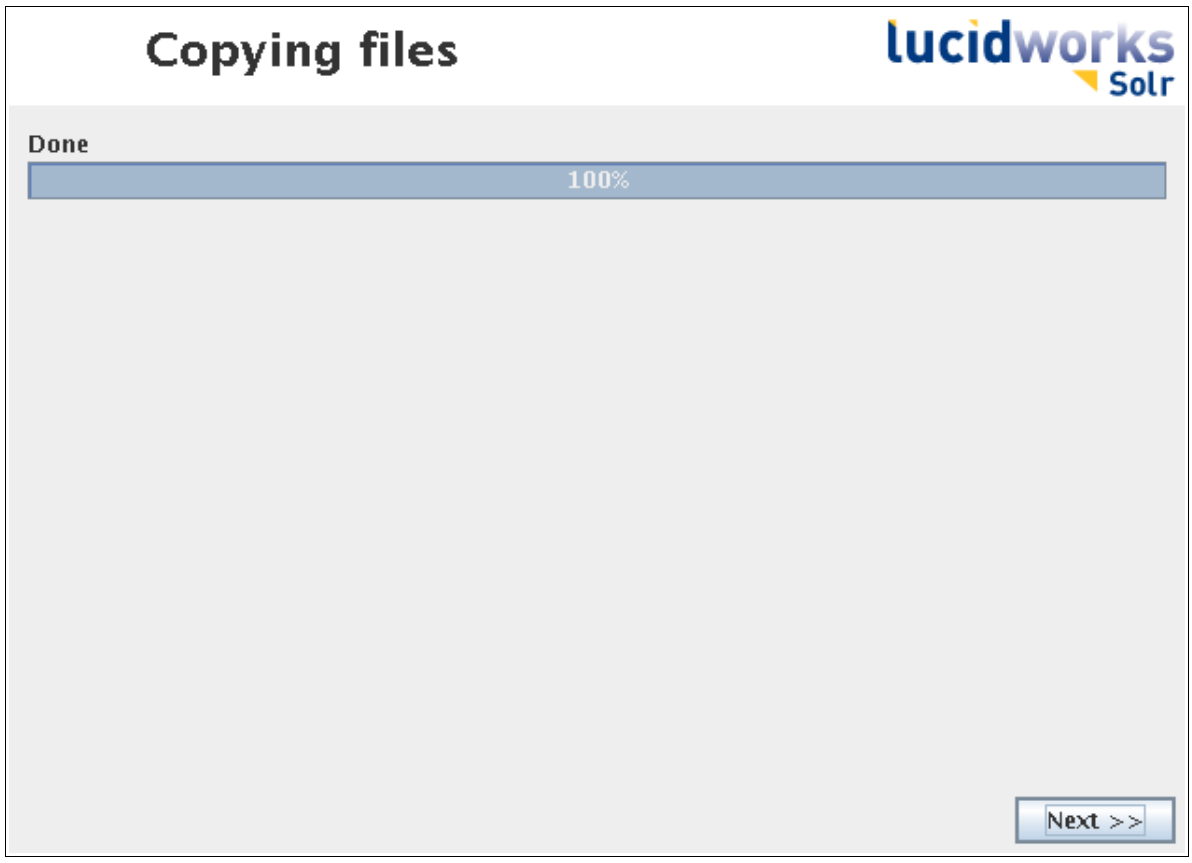
The installer displays a screen for selecting the Web container to be used with LucidWorks for Solr.



Selecting a target Web application container.

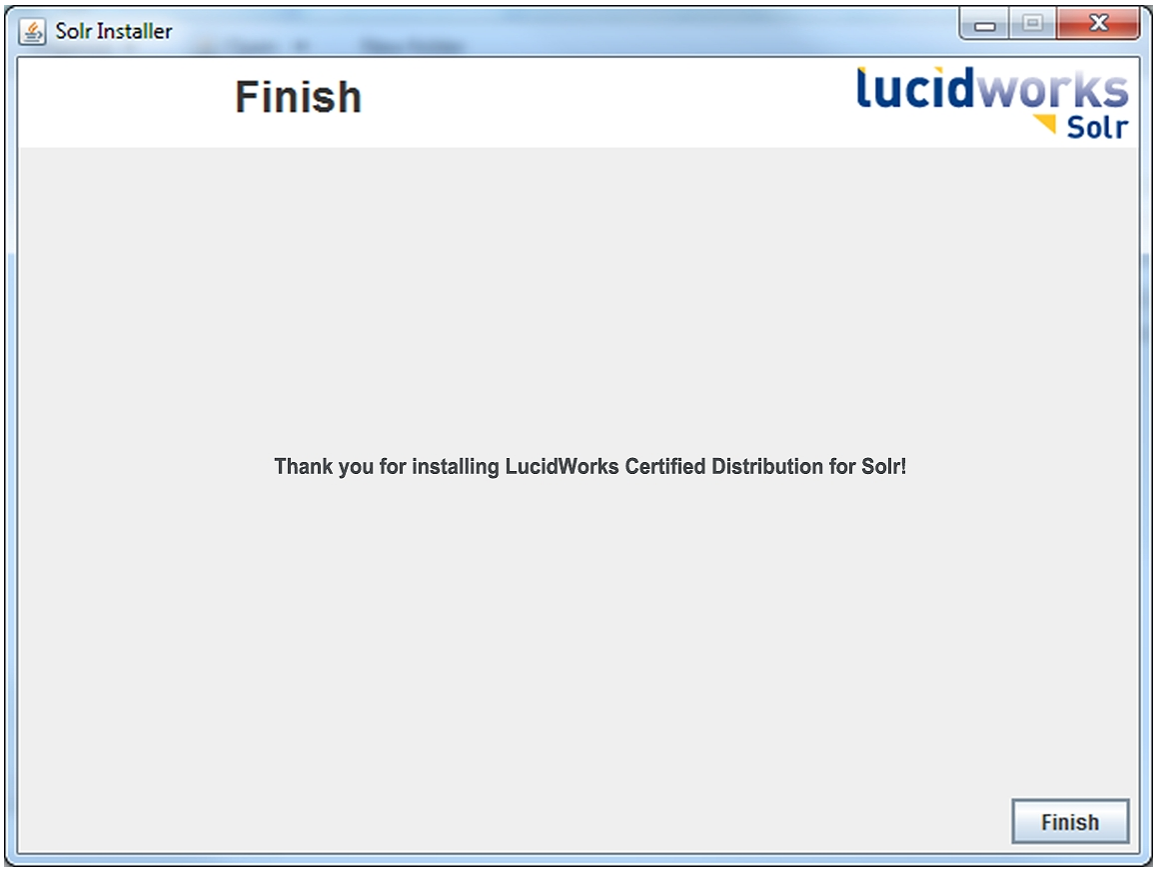
Choose which Web application container (Jetty or Tomcat) you want installed with LucidWorks. (Whichever container you select, it will be configured to run at the default port, 8983.) When you have made your selection, press **Next>>**.

The installer chugs away for a few seconds, copying files, and tells you when it's finished.



A screen showing the installer's progress as it copies files.

Press **Next>>**. The installer tells you it is finished.



The Finish screen.

Press **Finish** to complete the installation process and exit the installer.

2.2 Running LucidWorks for Solr

This section describes how to run LucidWorks with an example schema, how to add documents, and how to run queries.

2.2.1 Fire Up the Server

In the directory where you installed LucidWorks, run `start.sh` to start the Web server.

```
$ ./start.sh
```

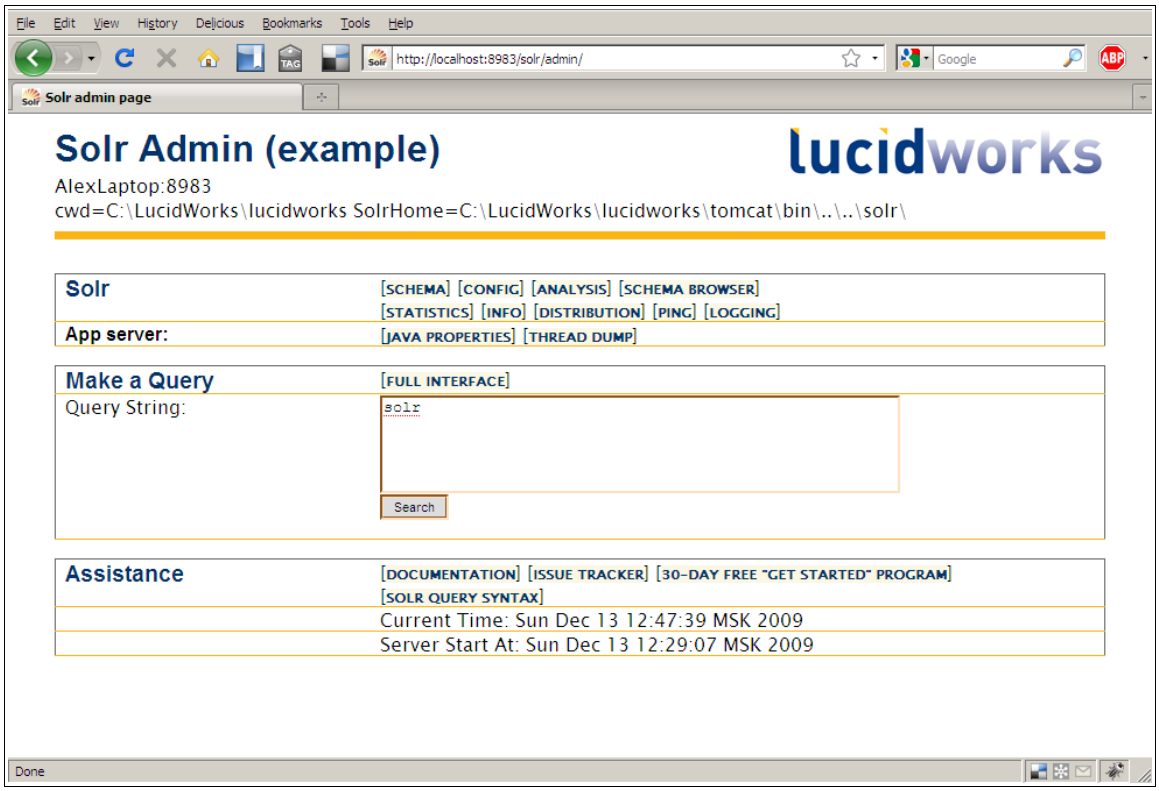
If you are running Windows, you can start the Web server by running `start.bat` instead.

```
C:\Applications\LucidWorks>start.bat
```

That's it! LucidWorks is running.

If you need convincing, use a Web browser to see the Admin Console.

http://localhost:8983/solr/admin



The Solr Admin interface.

If LucidWorks is not running, your browser will complain that it cannot connect to the server. Check your port number and try again.

2.2.2 Add Documents

LucidWorks is built to find documents that match queries. LucidWorks has some idea what the world looks like from its schema, but it doesn't know about any documents. Like Johnny 5, LucidWorks needs input before it can do anything wonderful.

You can quench LucidWorks' thirst for knowledge with example documents located in the `example/exampledocs` directory of your installation.

In that directory is a Java-based command line tool, `post.jar`, which you can use to ask Solr to index the documents. Don't worry too much about the details for now. Chapter 6 has all the details on indexing.

To see some information about the usage of `post.jar`, use the `-help` option.

```
$ java -jar post.jar -help
SimplePostTool: version 1.2
This is a simple command line tool for POSTing raw XML to a Solr
port.  XML data can be read from files specified as commandline
args; as raw commandline arg strings; or via STDIN.
Examples:
  java -Ddata=files -jar post.jar *.xml
  java -Ddata=args  -jar post.jar '<delete><id>42</id></delete>'
  java -Ddata=stdin -jar post.jar < hd.xml
Other options controlled by System Properties include the Solr
URL to POST to, and whether a commit should be executed.  These
are the defaults for all System Properties...
  -Ddata=files
  -Durl=http://localhost:8983/solr/update
  -Dcommit=yes
```

Go ahead and add all the documents in the directory as follows.

```
$ java -Durl=http://localhost:8983/solr/update -jar post.jar *.xml
SimplePostTool: version 1.2
SimplePostTool: WARNING: Make sure your XML documents are encoded in UTF-
8, other encodings are not currently supported
SimplePostTool: POSTing files to http://10.211.55.8:8983/solr/update..
SimplePostTool: POSTing file hd.xml
SimplePostTool: POSTing file ipod_other.xml
SimplePostTool: POSTing file ipod_video.xml
SimplePostTool: POSTing file mem.xml
SimplePostTool: POSTing file monitor.xml
SimplePostTool: POSTing file monitor2.xml
SimplePostTool: POSTing file mp500.xml
SimplePostTool: POSTing file sd500.xml
SimplePostTool: POSTing file solr.xml
SimplePostTool: POSTing file spellchecker.xml
SimplePostTool: POSTing file utf8-example.xml
SimplePostTool: POSTing file vidcard.xml
SimplePostTool: COMMITting Solr index changes..
$
```

That's it! Solr has indexed the documents contained in the files.

2.2.3 Ask Questions

Now that you've indexed documents, you can perform queries. The simplest way is by building a URL that includes the query parameters. This is exactly the same as building any other HTTP URL.

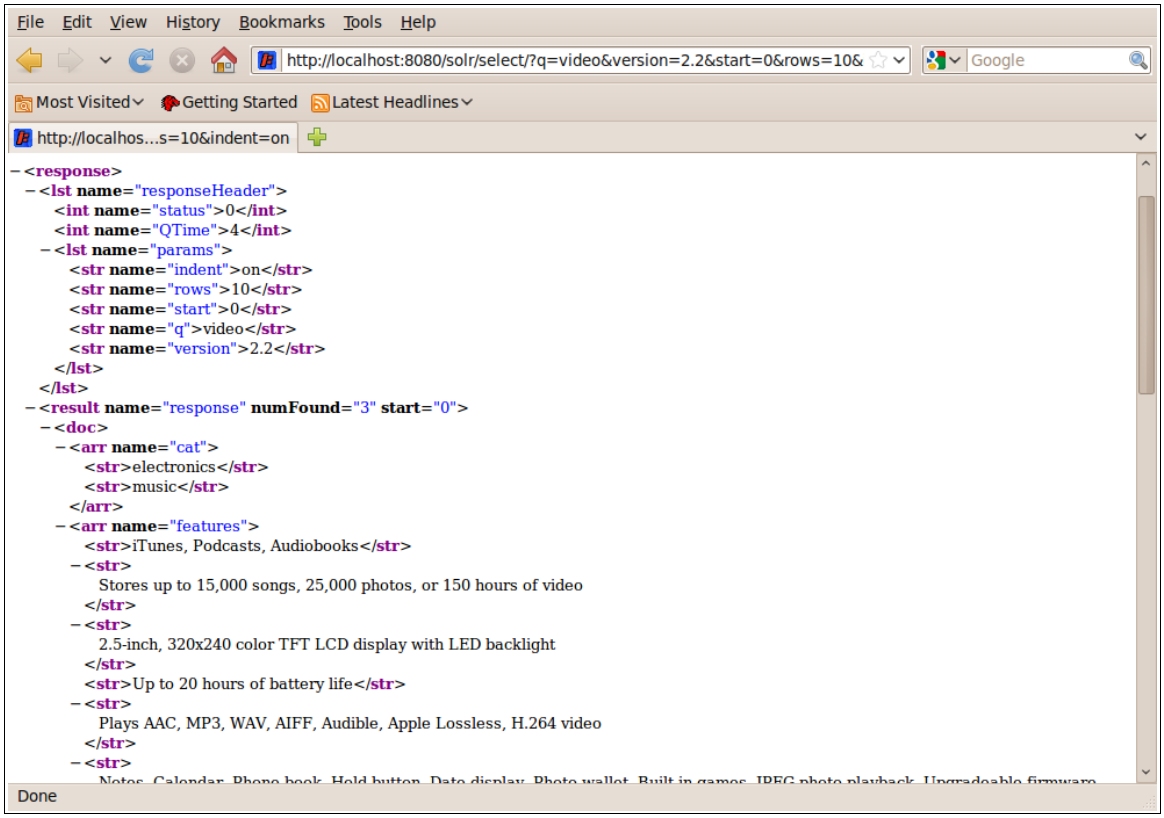
For example, the following query searches all document fields for "video":

http://localhost:8983/solr/select?q=video

Notice how the URL includes the host name (`localhost`), the port number where the server is listening (`8983`), the application name (`solr`), the request handler for queries (`select`), and finally, the query itself (`q=video`).

The results are contained in an XML document, which you can examine directly by clicking on the link above. The document contains two parts. The first part is the `responseHeader`, which contains information about the response itself. The beefy part of the reply is in the `result` tag, which contains one or more `doc` tags, each of which contains fields from documents that match the query. You can use standard XML transformation techniques to mold Solr's results into a form that is suitable for displaying to users. Alternatively, Solr can output the results in JSON, PHP, Ruby and even user-defined formats.

Just in case you are not running Solr as you read, the following screen capture shows the result of a query (the next example, actually) as viewed in Mozilla Firefox. The top-level response contains a `lst` named `responseHeader` and a `result` named `response`. Inside `result`, you can see the three `doc`s that represent the search results.



An XML response to a query.

Once you've mastered the basic idea of a query, it's easy to add enhancements to explore the query syntax. This one is the same as before but the results only contain the `id`, `name` and `price` for each returned document. If you don't specify which fields you want, all of them are returned.

http://localhost:8983/solr/select?q=video&fl=id,name,price

Here is another example which searches for "black" in the `name` field only. If you don't tell Solr which field to search, it will search default fields, as specified in the schema.

http://localhost:8983/solr/select?q=name:black

You can provide ranges for fields. The following query finds every document whose price is between $0 and $400.

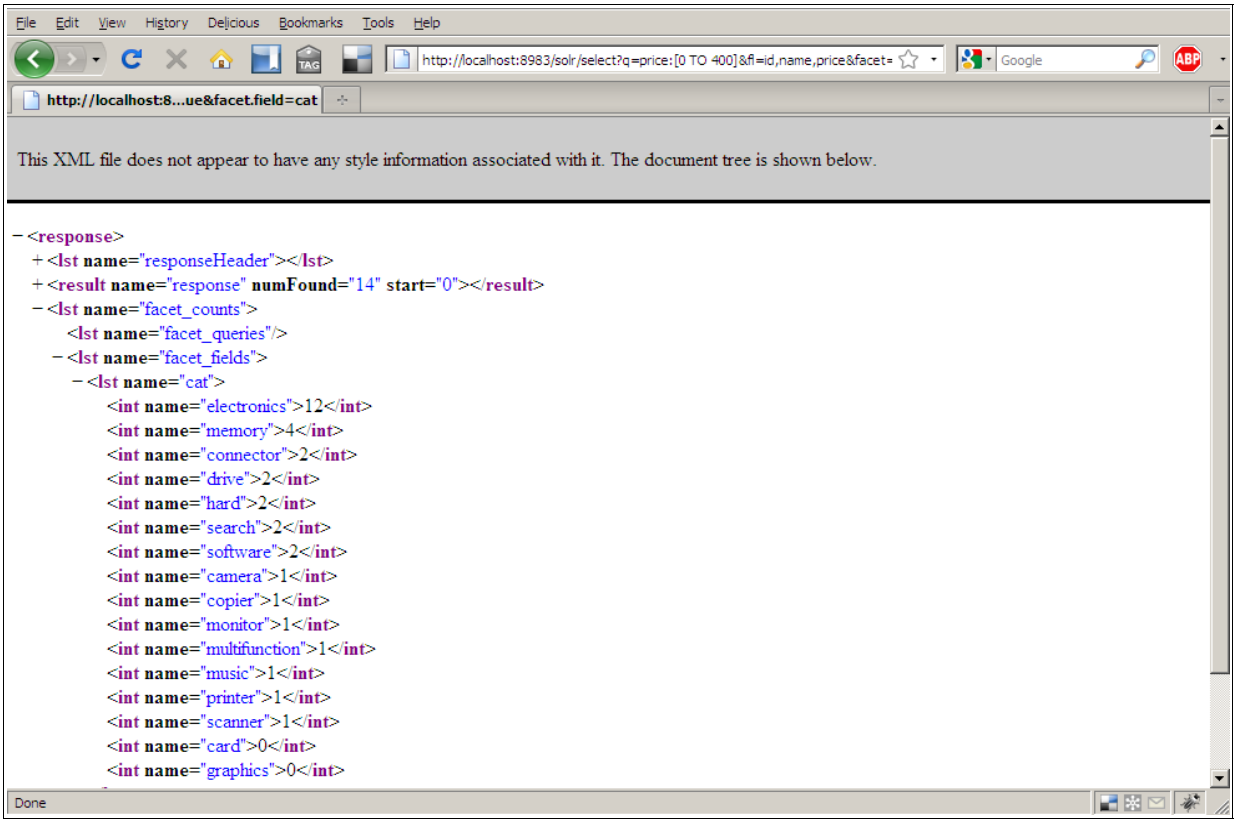
http://localhost:8983/solr/select?q=price:[0 TO 400]&fl=id,name,price

Faceted browsing is one of Solr's key features. It allows users to narrow search results in ways that are meaningful to your application. For example, a shopping site could provide facets to narrow search results by manufacturer or price.

Faceting information is returned as a third part of Solr's query response. To get a taste of this power, take a look at the following query. It adds `facet=true` and `facet.field=cat`.

http://localhost:8983/solr/select?q=price:[0 TO 400]&fl=id,name,price&facet=true&facet.field=cat

In addition to the familiar `responseHeader` and `response` from Solr, a `facet_counts` element is also present. Here is a view with the `responseHeader` and `response` collapsed so you can see the faceting information clearly.



An XML Response with faceting.

The facet information shows how many of the query results have each possible value of the `cat` field. You could easily use this information to provide users with a quick way to narrow their query results. You can filter results by adding one or more filter queries to the Solr request. Here is a request further constraining the request to documents with a category of "software".

http://localhost:8983/solr/select?q=price:[0 TO 400]&fl=id,name,price&facet=true&facet.field=cat&fq=cat:software

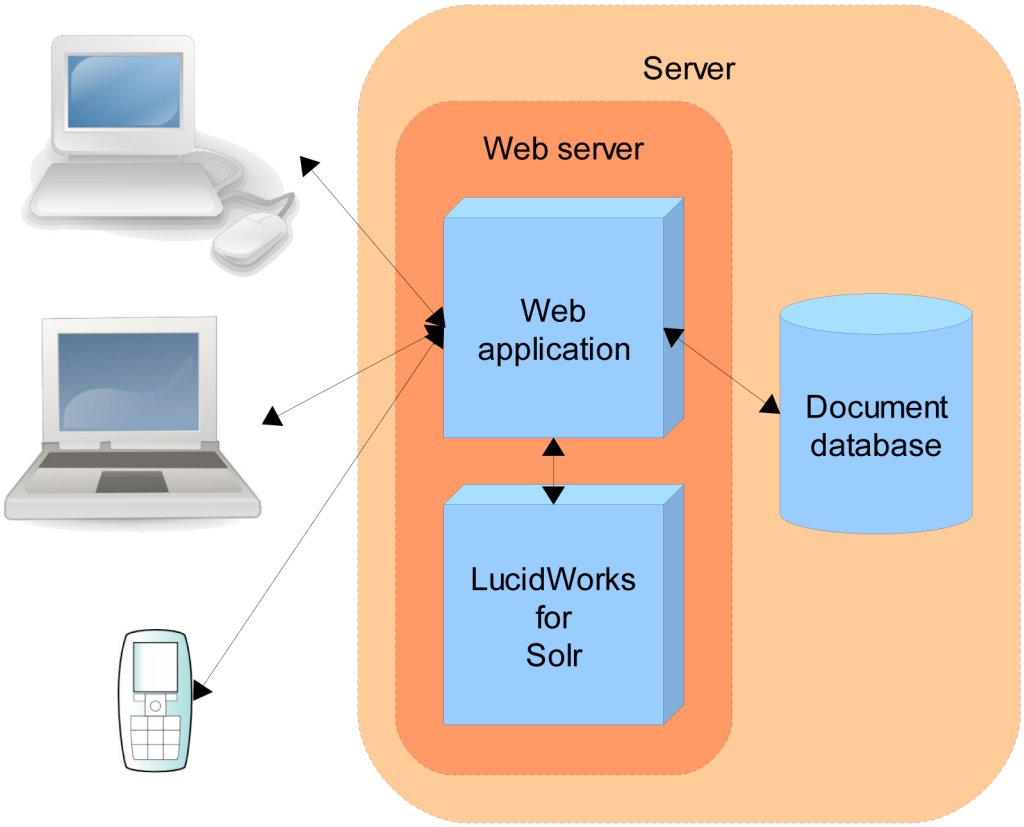
2.2.4 Clean Up

When you are finished running LucidWorks, execute `stop.sh` to shut down the server. If you are on Windows, use `stop.bat` instead.

2.3 A Quick Overview

Having had some fun with Solr, you'll now learn (at a high level) about all the cool things it can do.

Here is a typical configuration:



In the above scenario, LucidWorks runs alongside another application in a Web server like Tomcat. For example, an online store application would provide a user interface, a shopping cart, and a way to make purchases. The store items would be kept in some kind of database.

Solr makes it easy to add the capability to search through the online store through the following steps:

Define a *schema*. The schema tells Solr about the contents of documents it will be indexing. In the online store example, the schema would define fields for the product name, description, price, manufacturer,

and so on. Solr's schema is powerful and flexible and allows you to tailor Solr's behavior to your application. See Chapter 4 for all the details.

Deploy Solr to your application server.

Feed Solr the documents for which your users will search.

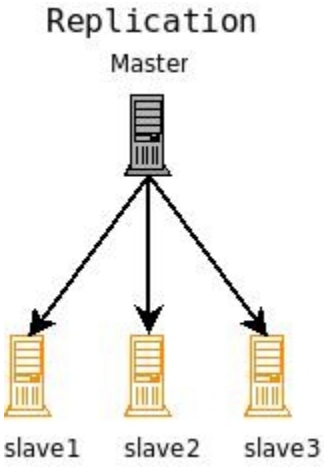
Expose search functionality in your application.

Because Solr is based on open standards, it is highly extensible. Solr queries are RESTful, which means, in essence, that a query is a simple HTTP request URL and the response is a structured document—mainly XML, but possibly JSON or some other format. This means that a wide variety of clients will be able to use Solr, from other web applications to browser clients, rich client applications, and mobile devices. Any platform capable of HTTP can talk to Solr. See Chapter 11 for details on client APIs.

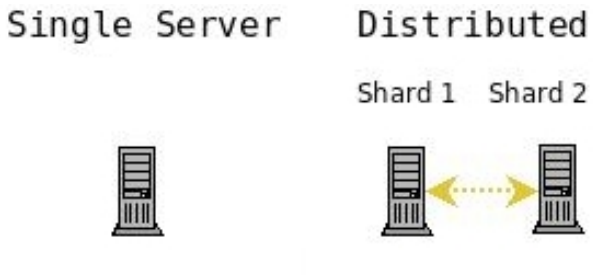
Solr is based around the Apache Lucene project, a high-performance, full-featured search engine. Solr offers support for the simplest keyword searching through to complex queries on multiple fields and faceted search results. Chapter 7 has more information about searching and queries.

If Solr's impressive capabilities aren't enough to blow your hat off, its ability to handle outrageously high-volume applications should do the trick.

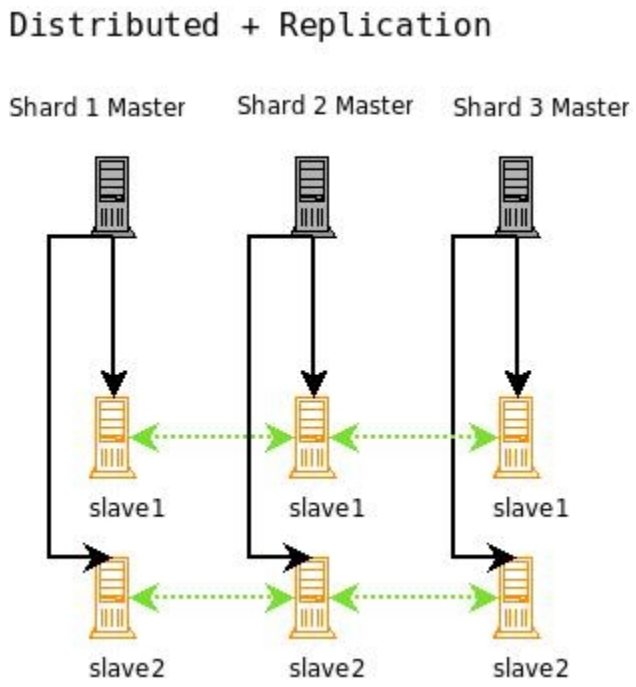
A relatively common scenario is that you have so many queries that the server is unable to respond fast enough to each one. In this case, you can make copies of an index. This is called replication. Then you can distribute incoming queries among the copies in any way you see fit. A round robin mechanism is one simple way to do this.



Another useful technique, less common than replication, is sharding. If you have so many documents that you simply can't fit them all on a single box for RAM or index size reasons, you can split an index into multiple pieces, called *shards*. Each shard lives on its own physical server. An incoming query is sent to all the shard servers, which respond with matching results.



If you are fortunate enough to have oodles of documents and oodles of users, you might need to combine the techniques of sharding and replication. In this case, you create some number of shards, then replicate the shards. Incoming queries are sent to one server for each shard.



For full details on sharding and replication, see Chapter 10.

Best of all, this talk about high-volume applications is not just hot air. Some of the famous Internet sites that use Solr today are CNET, Netflix, and digg.com.

For more information, take a look at Lucid Imagination's Application Showcase:

http://www.lucidimagination.com/Community/Marketplace/Application-Showcase-Wiki

2.4 A Step Closer

You already have some idea of Solr's schema. This section describes Solr's home directory and other configuration options.

When Solr runs in an application server, it needs access to a home directory. The home directory contains important configuration information and is the place where Solr will store its index.

The crucial parts of the Solr home directory are shown here:

```
<solr-home-directory>/
  conf/
    schema.xml
    solrconfig.xml
  data/
```

You supply `solrconfig.xml` and `schema.xml` to tell Solr how to behave. By default, Solr stores its index inside `data`.

`solrconfig.xml` controls high-level behavior. You can, for example, specify an alternate location for the `data` directory. For more information on `solrconfig.xml`, see Chapter 8.

`schema.xml` describes the documents you will ask Solr to index. Inside `schema.xml`, you define a document as a collection of fields. You get to define both the field types and the fields themselves. Field type definitions are powerful and include information about how Solr processes incoming field values and query values. For more information on `schema.xml`, see Chapter 4.

This page is intentionally left blank.

3 The Solr Admin Web Interface

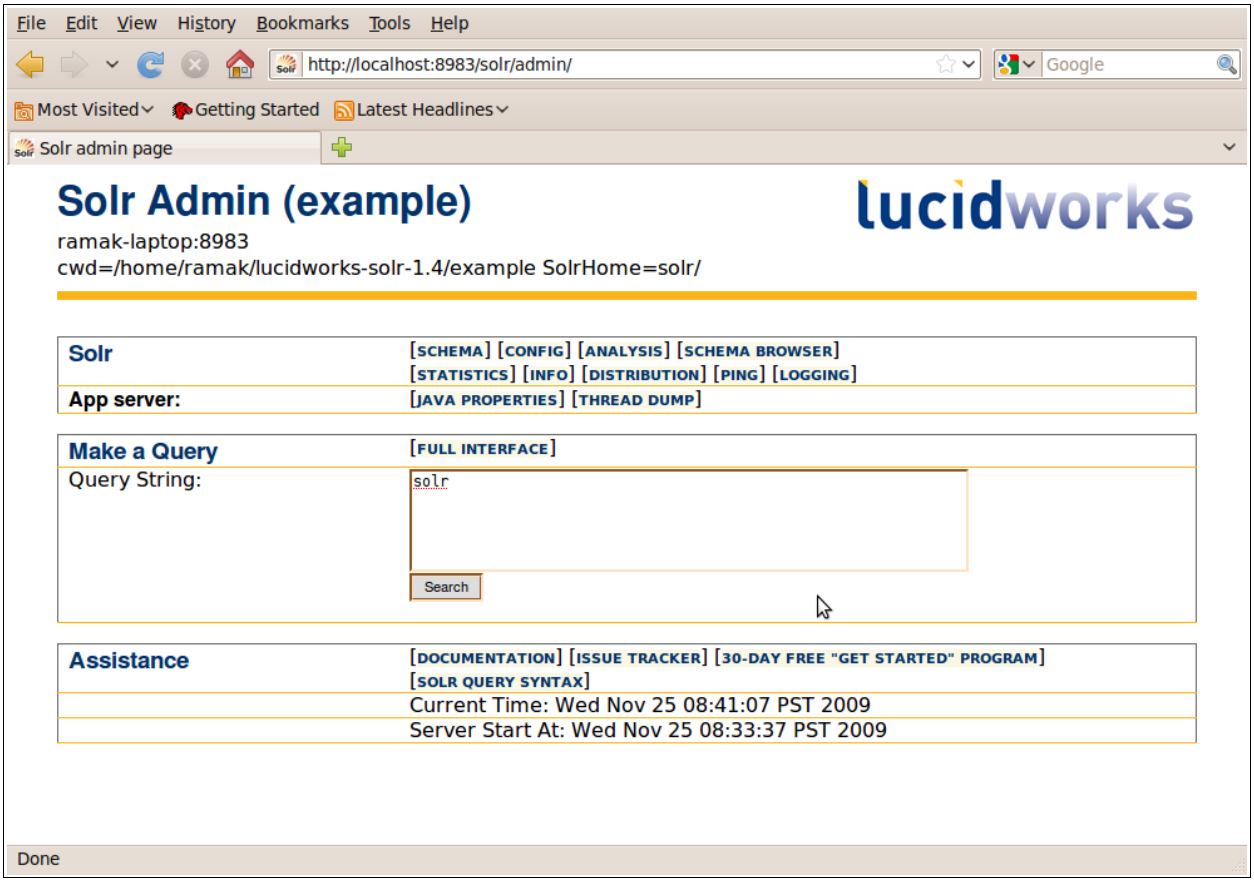
3.1 Introduction

Solr features a Web interface that makes it easy for Solr administrators and programmers to:

- view Solr configuration details
- run queries and analyze document fields in order to fine-tune a Solr configuration
- access online documentation and other help

Users access the Admin Web interface through the `solr/admin/` page, which by default is located at `http://[hostname]:8983/solr/admin/`.

The image at the top of the next page shows the Solr Admin Web interface. The name of the Solr installation's top directory appears in parentheses at the top of the page.



The LucidWorks for Solr Admin Web interface.

The main page of the Web interface is divided into three parts:

- a section for exploring the Solr server and its application server
- a section for running queries
- a section on getting assistance, either by accessing documentation or the Solr issue tracker, or by contacting the Apache Solr project team

NOTE: If you're running Solr on a Macintosh, you should access the Admin Web interface in a browser other than Safari, since Safari will not display raw XML content, such as the contents of the Solr `schema.xml` file.

3.1.1 Configuring the Admin Web Interface in solrconfig.xml

You can configure the Solr Admin Web interface by editing the file `solrconfig.xml`. The `<admin>` block on the `solrconfig.xml` file determines:

- Which files the Web interface can access
- How the interface's PING link should call the `ping` command
- Whether or not the interface displays the ENABLE/DISABLE link in the App Server section

In its default configuration, which is shown below, the Web interface is configured to access `solrconfig.xml` and `schema.xml`. It also specifies the parameters the interface should pass to the `ping` command when a user clicks on the interface's PING link. It also creates a file called `server-enabled`, which will be created or deleted depending on the server's status.

```
<admin>
    <defaultQuery>solr</defaultQuery>
    <gettableFiles>
         solrconfig.xml
         schema.xml
    </gettableFiles>
    <pingQuery>q=solr&amp;version=2.0&amp;start=0&amp;rows=0</pingQuery>

    <!-- configure a healthcheck file for servers behind a loadbalancer
      -->
    <healthcheck type="file">server-enabled</healthcheck>
  </admin>
```

3.2 The Solr Section of the Admin Web Interface

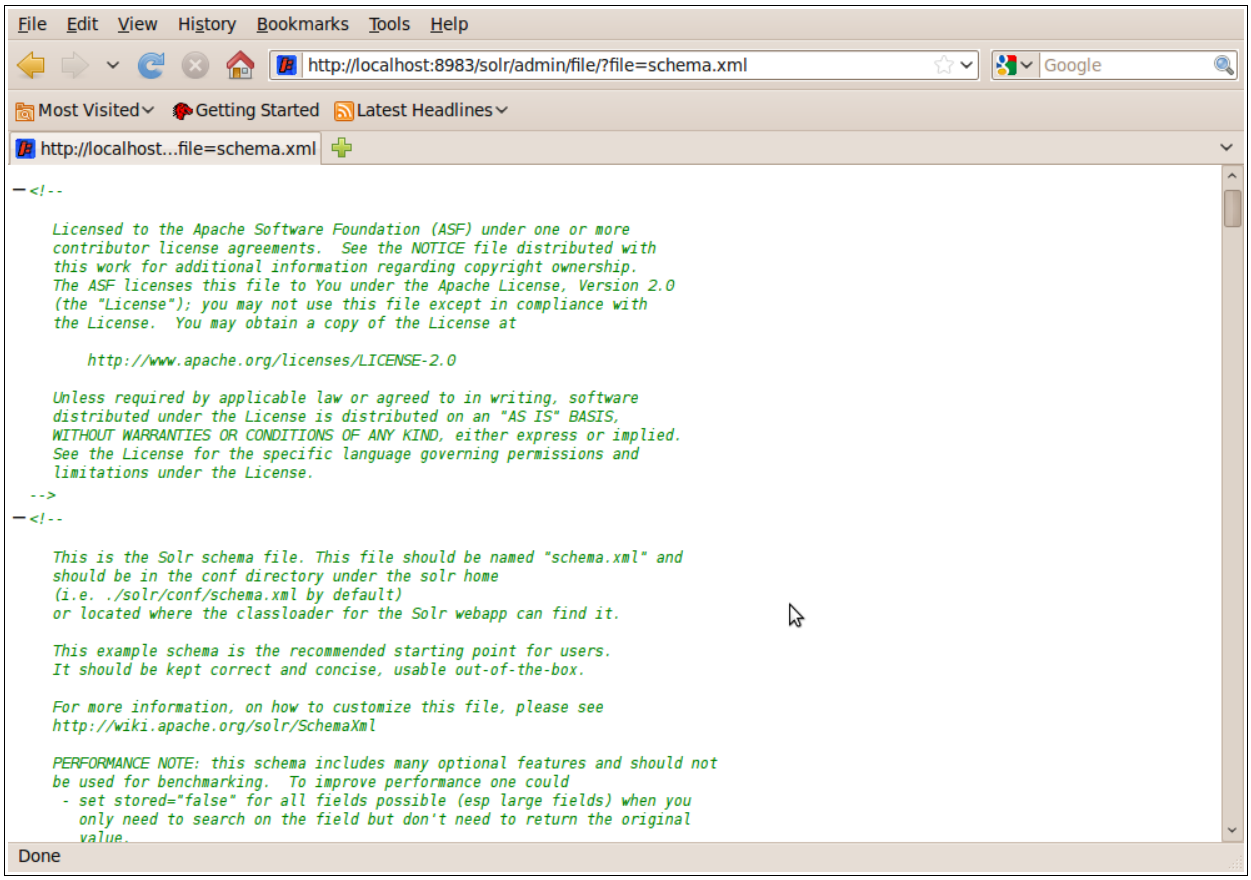
The Solr section of the Admin Web interface includes the following links.

Link	Description
SCHEMA	Displays the `schema.xml` file, a configuration file that describes the data to be indexed and searched.
CONFIG	Displays the `solrconfig.xml` file, a file that contains most of the parameters for configuring Solr itself.
ANALYSIS	Displays a Field Analysis form, which is useful for testing the behavior of Analyzers, Tokenizers, and TokenFilters on different fields.
SCHEMA BROWSER	Displays a dynamic HTML interface for exploring the `schema.xml` settings of the Solr server.
STATISTICS	Displays configuration details and statistics about the following aspects of the Solr server: CORE CACHE QUERY handlers UPDATE handlers HIGHLIGHTING OTHER (reserved for future use) The Solr server continually updates the statistics presented on this page.
INFO	Displays startup-time data about the following categories: CORE CACHE QUERY handlers UPDATE handlers OTHER (reserved for future use) Unlike the statistics presented on the STATISTICS page, the statistics presented on the INFO page do not change after startup.

Link	Description
DISTRIBUTION	Displays details about a distributed Solr configuration, if the Solr server is configured as either a Master or Slave server. On a Master instance, each row displays the name of the slave and the snapshots the slave has retrieved. On a Slave instance, the page displays a single line showing the name of its last attempt to retrieve a snapshot from its master.
PING	Runs the `ping` command against the Solr server in order to confirm that the server is running and responsive to network requests. If the command is successful, it returns HTTP 200 to the browser but displays nothing. If unsuccessful, the command returns HTTP 500 (an error) and displays an exception message.
LOGGING	Displays an interactive form for setting and viewing the effective logging levels of the JDK Log hierarchy.

3.2.1 Displaying the Solr Schema

To display the Solr `schema.xml` file in your browser, click the SCHEMA link. The browser will then display then `schema.xml` file, as shown in the image below.

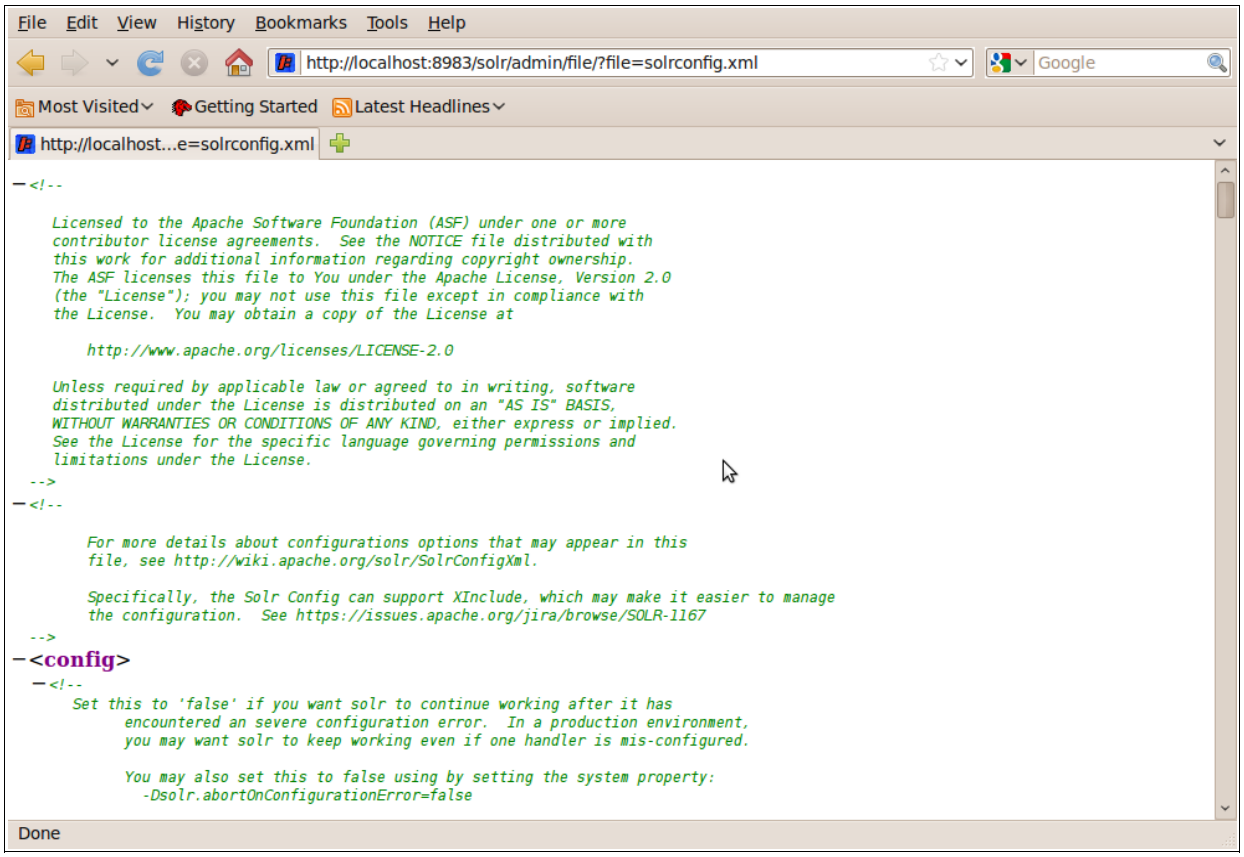


The schema.xml File.

For more information on the `schema.xml` file, please see Chapter 4.

3.2.2 Displaying the Solr Configuration File

To display the `solrconfig.xml` file, click the CONFIG link. Solr displays the file in the browser, as shown below.

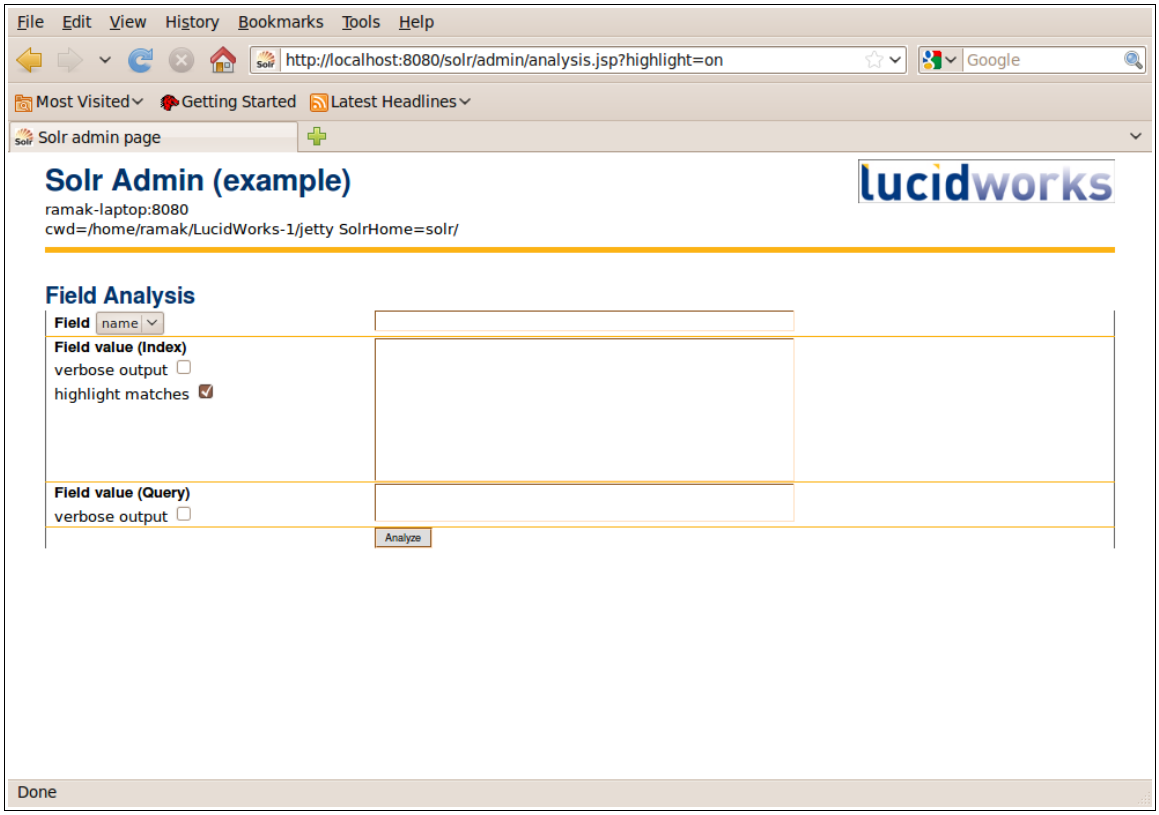


The solrconfig.xml File.

3.2.3 Running Field Analysis to Test Analyzers, Tokenizers, and TokenFilters

When defining fields and field types, and configuring Analyzers, Tokenizers, and TokenFilters, it's helpful to see how the current configuration of Solr indexes a sample text and processes a sample query. The Field Analysis feature of the Solr Admin Web interface makes it easy to run queries against sample text, so you can assess the current configuration of the Solr server.

Click the ANALYSIS link to display the Field Analysis form, shown below.

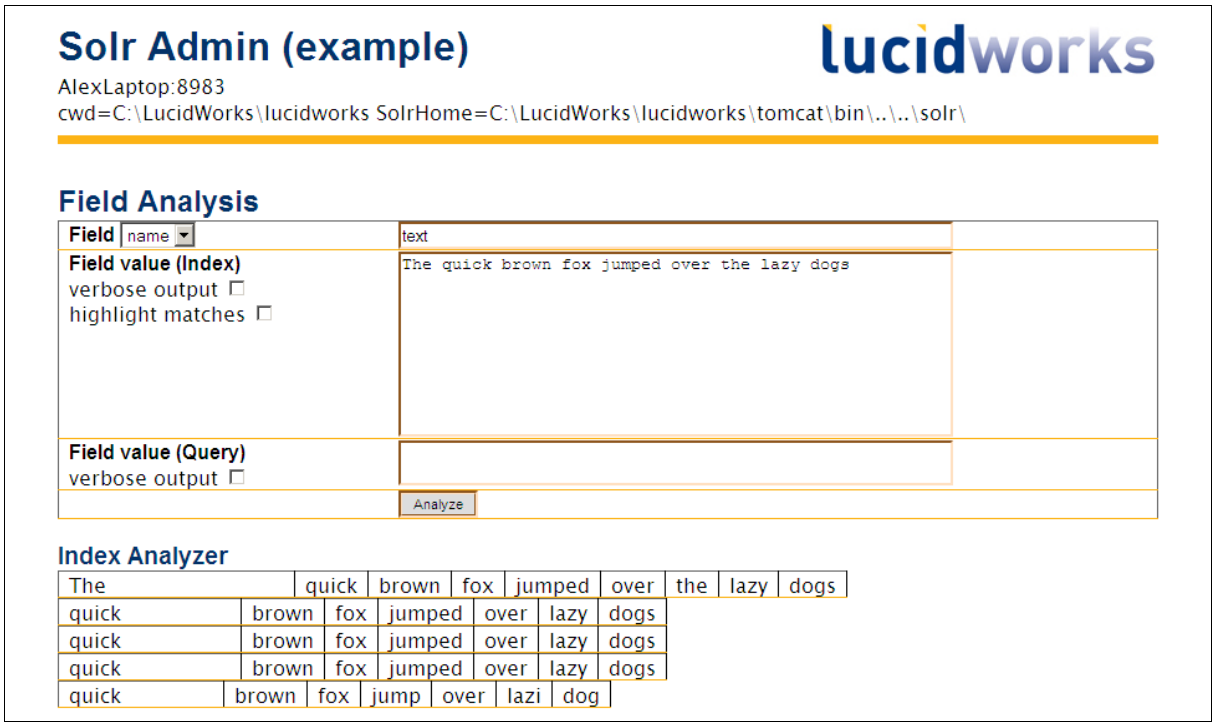


The Field Analysis form.

The Field Analysis form includes three main parts:

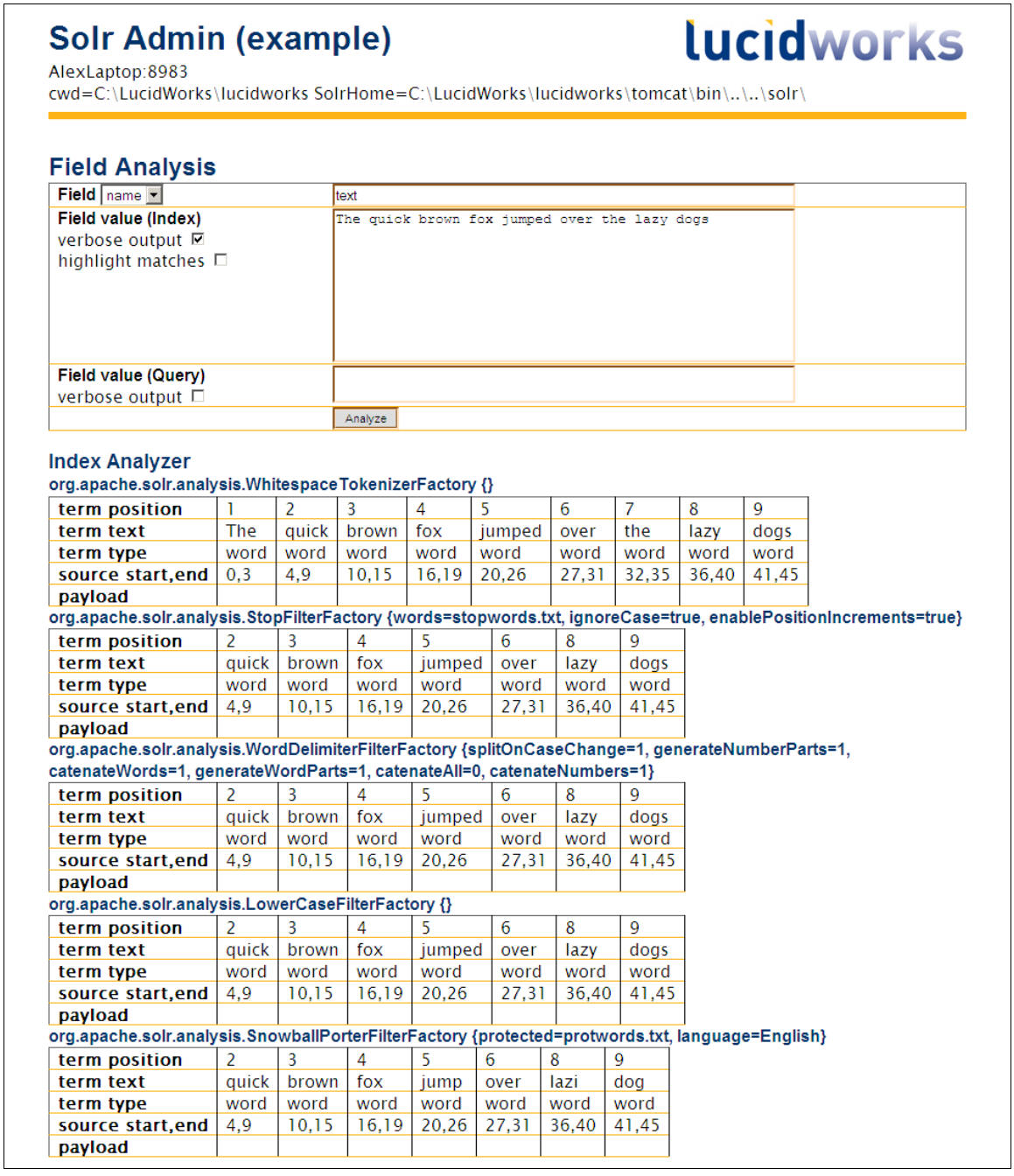
- A “Field name/type” field, in which you toggle a drop-down menu to select “name” or “type,” then enter the name of the field name or field type in the text box to the right. The value you enter must correspond to a field name or field type defined in the Solr server's `schema.xml` file.
- A “Field value (Index)” text box, in which you type sample text for a field, as though it were a field in a document indexed by Solr. To see a detailed analysis of how the Solr server calls Analyzers, Tokenizers, and TokenFilters to index the text, click in the checkbox to select “verbose output.”
- A “Field value (Query)” text box, in which you type the text to be used in a query performed by the Server against the text you entered in the “Field value (Index)” text box. To see details of how the Server processes the query, click in the checkbox to select “verbose output.”

The image below shows the Field Analysis form performing a query against text entered in the "Field value (Index)" field.



Field Analysis form performing a query against text entered in the "Field value (Index)" field.

To see these processes in detail, one can re-run the analysis, selecting the "verbose output" options. The following image shows the verbose output for the indexing process. You can see the order in which Tokenizers and TokenFilters are called, beginning with the WhiteSpaceTokenizerFactory, which demarcates words by identifying the white spaces around them.



Solr Admin (example)

lucidworks

AlexLaptop:8983
cwd=C:\LucidWorks\lucidworks SolrHome=C:\LucidWorks\lucidworks\tomcat\bin\..\..\solr\

Field Analysis

Field name	text
Field value (Index) verbose output ☑ highlight matches ☐	The quick brown fox jumped over the lazy dogs
Field value (Query) verbose output ☐	
	Analyze

Index Analyzer

org.apache.solr.analysis.WhitespaceTokenizerFactory {}

term position	1	2	3	4	5	6	7	8	9
term text	The	quick	brown	fox	jumped	over	the	lazy	dogs
term type	word	word	word	word	word	word	word	word	word
source start,end	0,3	4,9	10,15	16,19	20,26	27,31	32,35	36,40	41,45
payload									

org.apache.solr.analysis.StopFilterFactory {words=stopwords.txt, ignoreCase=true, enablePositionIncrements=true}

term position	2	3	4	5	6	8	9
term text	quick	brown	fox	jumped	over	lazy	dogs
term type	word	word	word	word	word	word	word
source start,end	4,9	10,15	16,19	20,26	27,31	36,40	41,45
payload							

org.apache.solr.analysis.WordDelimiterFilterFactory {splitOnCaseChange=1, generateNumberParts=1, catenateWords=1, generateWordParts=1, catenateAll=0, catenateNumbers=1}

term position	2	3	4	5	6	8	9
term text	quick	brown	fox	jumped	over	lazy	dogs
term type	word	word	word	word	word	word	word
source start,end	4,9	10,15	16,19	20,26	27,31	36,40	41,45
payload							

org.apache.solr.analysis.LowerCaseFilterFactory {}

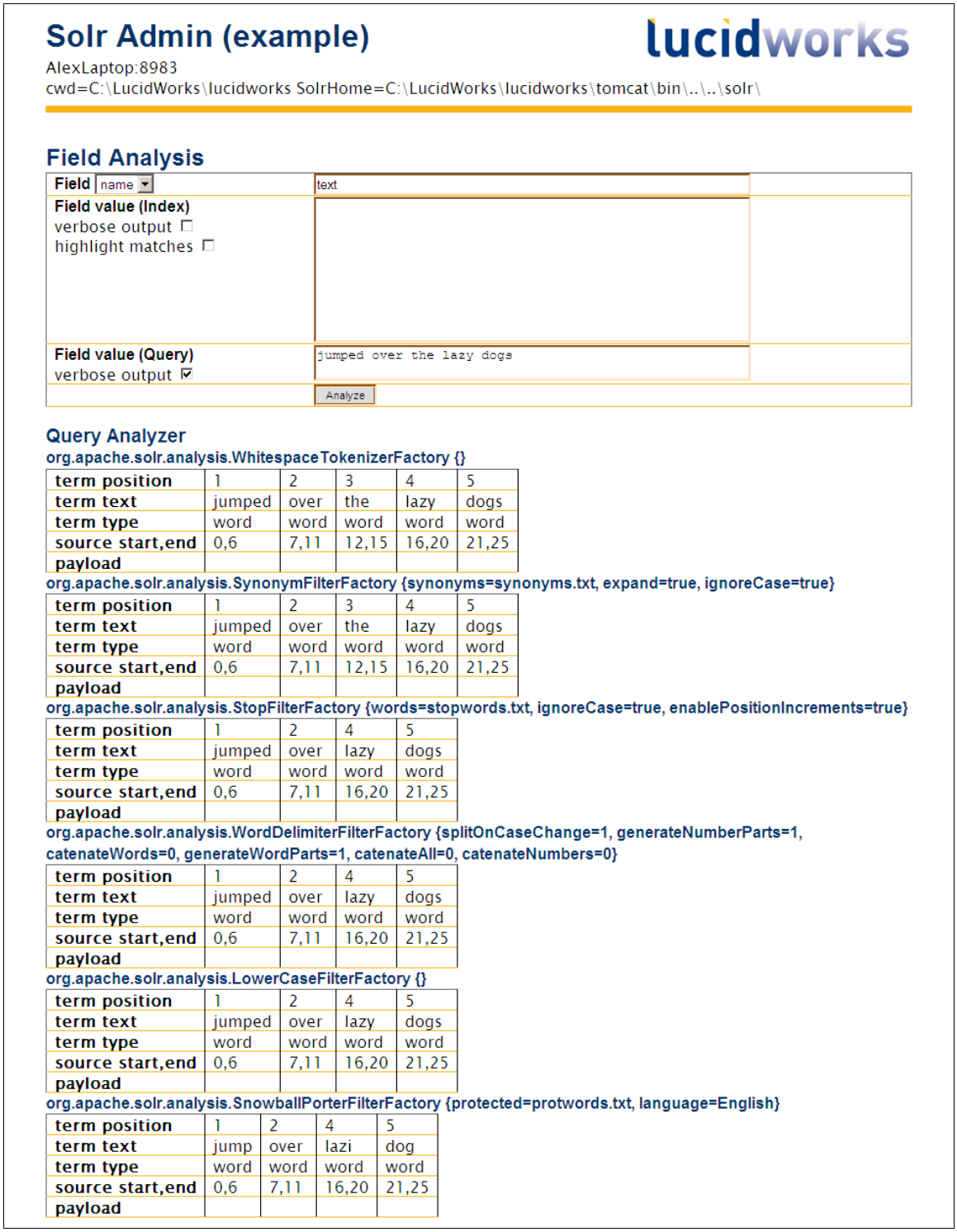
term position	2	3	4	5	6	8	9
term text	quick	brown	fox	jumped	over	lazy	dogs
term type	word	word	word	word	word	word	word
source start,end	4,9	10,15	16,19	20,26	27,31	36,40	41,45
payload							

org.apache.solr.analysis.SnowballPorterFilterFactory {protected=protwords.txt, language=English}

term position	2	3	4	5	6	8	9
term text	quick	brown	fox	jump	over	lazi	dog
term type	word	word	word	word	word	word	word
source start,end	4,9	10,15	16,19	20,26	27,31	36,40	41,45
payload							

The "verbose output" option reveals the steps involved in the indexing process.

The next image shows the “verbose output” option selected for the querying process. You can see that Solr's Query Analyzer invokes org.apache.solr.analysis.WhitespaceTokenizerFactory. The “verbose output” option shows you all the analyzers in the order in which they are invoked.



Solr Admin (example)

lucidworks

AlexLaptop:8983
cwd=C:\LucidWorks\lucidworks SolrHome=C:\LucidWorks\lucidworks\tomcat\bin\..\..\solr\

Field Analysis

Field name	text
Field value (Index) verbose output ☐ highlight matches ☐	
Field value (Query) verbose output ☑	jumped over the lazy dogs
	Analyze

Query Analyzer

org.apache.solr.analysis.WhitespaceTokenizerFactory {}

term position	1	2	3	4	5
term text	jumped	over	the	lazy	dogs
term type	word	word	word	word	word
source start,end	0,6	7,11	12,15	16,20	21,25
payload					

org.apache.solr.analysis.SynonymFilterFactory {synonyms=synonyms.txt, expand=true, ignoreCase=true}

term position	1	2	3	4	5
term text	jumped	over	the	lazy	dogs
term type	word	word	word	word	word
source start,end	0,6	7,11	12,15	16,20	21,25
payload					

org.apache.solr.analysis.StopFilterFactory {words=stopwords.txt, ignoreCase=true, enablePositionIncrements=true}

term position	1	2	4	5
term text	jumped	over	lazy	dogs
term type	word	word	word	word
source start,end	0,6	7,11	16,20	21,25
payload				

org.apache.solr.analysis.WordDelimiterFilterFactory {splitOnCaseChange=1, generateNumberParts=1, catenateWords=0, generateWordParts=1, catenateAll=0, catenateNumbers=0}

term position	1	2	4	5
term text	jumped	over	lazy	dogs
term type	word	word	word	word
source start,end	0,6	7,11	16,20	21,25
payload				

org.apache.solr.analysis.LowerCaseFilterFactory {}

term position	1	2	4	5
term text	jumped	over	lazy	dogs
term type	word	word	word	word
source start,end	0,6	7,11	16,20	21,25
payload				

org.apache.solr.analysis.SnowballPorterFilterFactory {protected=protwords.txt, language=English}

term position	1	2	4	5
term text	jump	over	lazi	dog
term type	word	word	word	word
source start,end	0,6	7,11	16,20	21,25
payload				

The "verbose output" option for the query process.

3.2.4 Using the Schema Browser

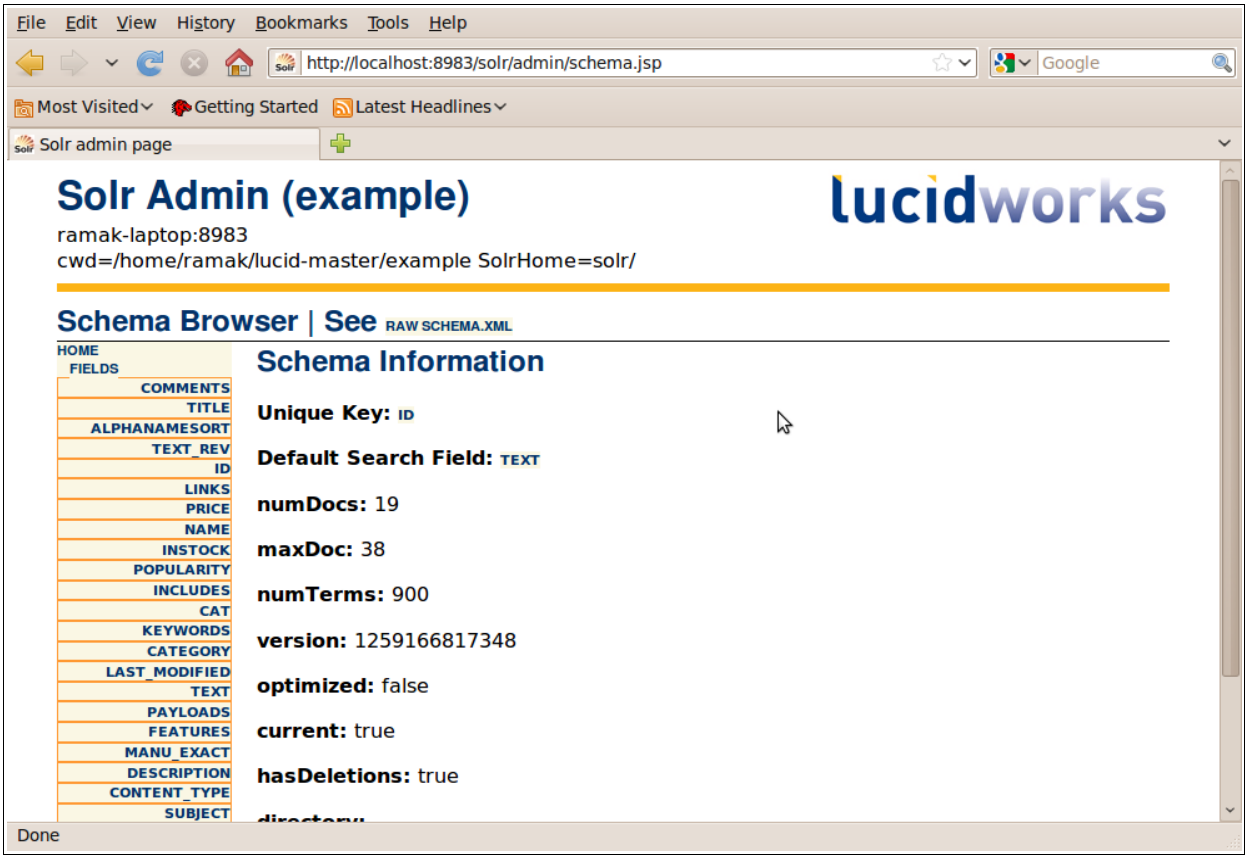
The Schema Browser is a dynamic Ajax-based window for viewing details of the Solr server's schema, which defines fields, dynamic fields, and field types used for indexing. When you first open the browser, it displays three categories on the left side of the screen: fields, dynamic fields, and field types, as shown below.

The Schema Browser.

3.2.4.1 Displaying the Configuration of a Field

The Schema Browser makes it easy to explore the definitions of fields, dynamic fields, and field types. To display the Schema Browser, click the SCHEMA BROWSER link in the Solr Admin Web interface.

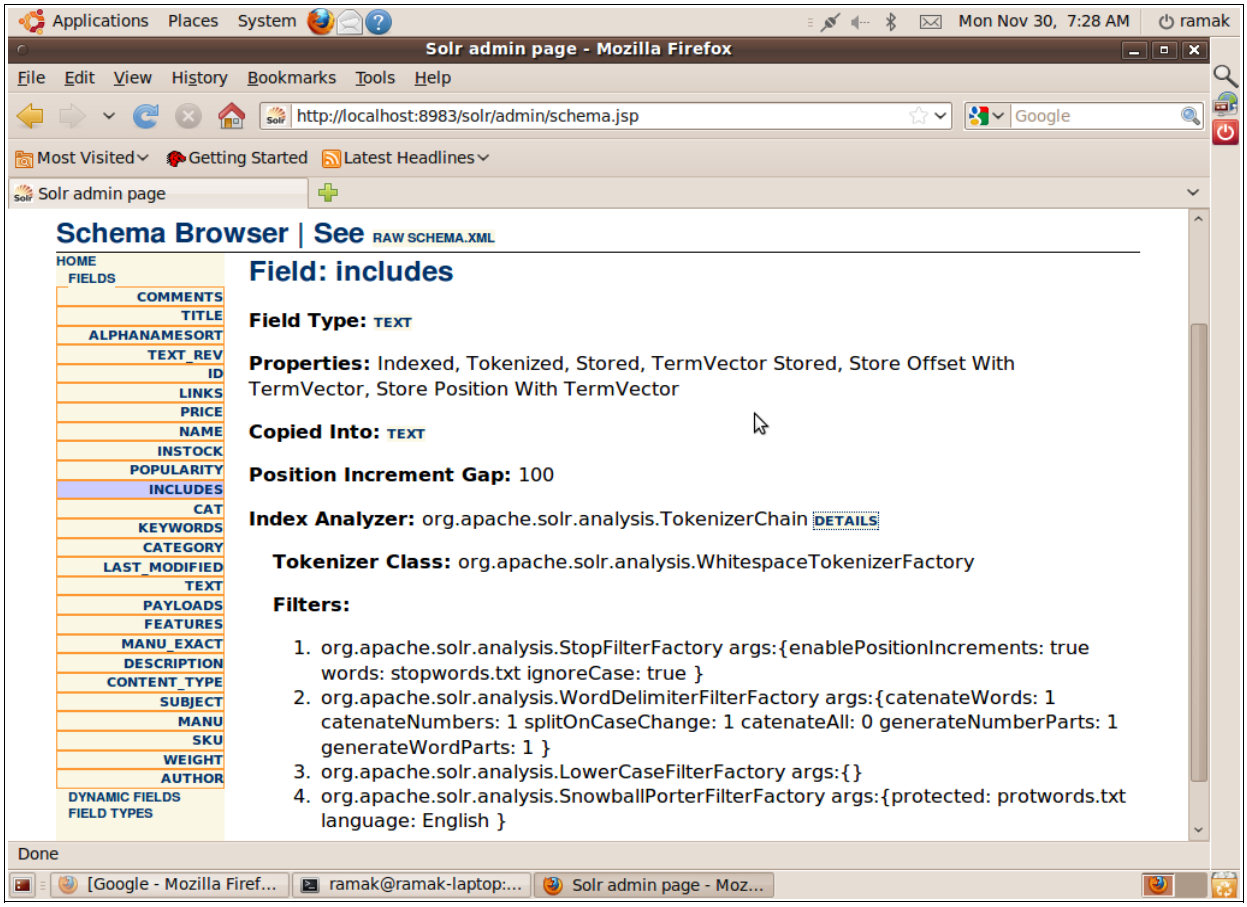
In the left hand navigation bar, click on the word "Fields" to see a list of fields defined in the `schema.xml` file. Then click on a specific field's name to see details about that particular field.



The Schema Browser displaying information about a selected field.

3.2.4.2 Displaying Additional Details about a Parameter

The schema information for some fields includes low-level details which are not displayed by default. If an item includes a DETAILS link, you can click the link to see additional details. To hide the additional details, click on the DETAILS link again.

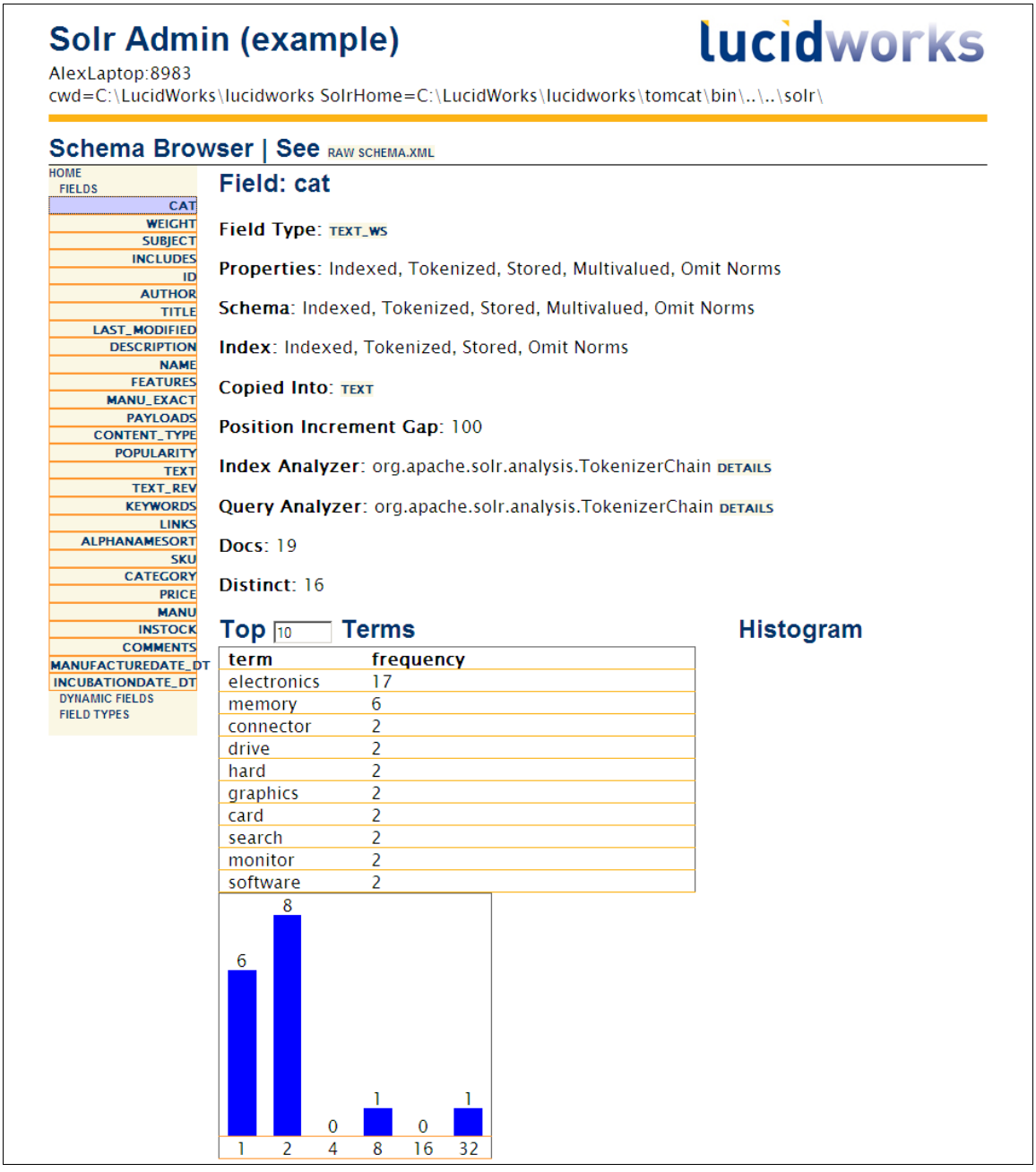


Clicking on the DETAILS link to see additional details about a configuration parameter.

3.2.4.3 Exploring the Most Popular Terms for a Field

Toward the bottom of the page, the Schema Browser presents a table of terms and a bar chart related to the selected field. The table, Top *n* Terms, where *n* is by default 10, lets you see the most popular *n* terms in that field in the index. You can enter a different number for *n* in the form and see a shortened or lengthened list of terms (depending on whether you enter a lower or higher number for *n*). If you enter a

number that exceeds the number of terms found in that field, the form automatically substitutes the total number of terms and displays only that number of terms. The image below shows an example of this display.

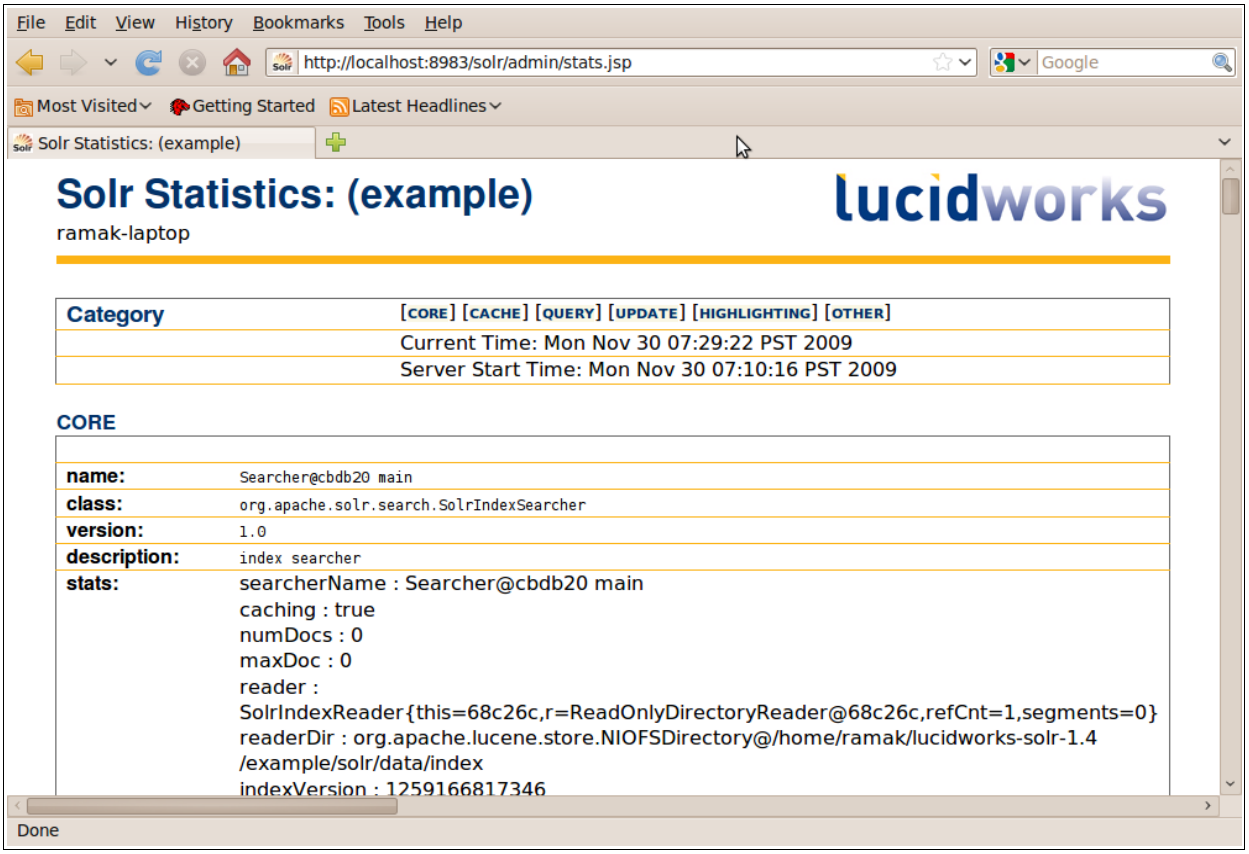


Displaying the Top n Terms.

A histogram shows the number of terms with a given frequency in the field. For example, in the image above, there are six terms that appear once, eight terms that appear twice, and so on.

3.2.5 Displaying Statistics of the Solr Server

The STATISTICS link displays statistics related to the Solr server's performance. The server continually updates these statistics. The image below shows an example of the statistics reported by the Statistics page.

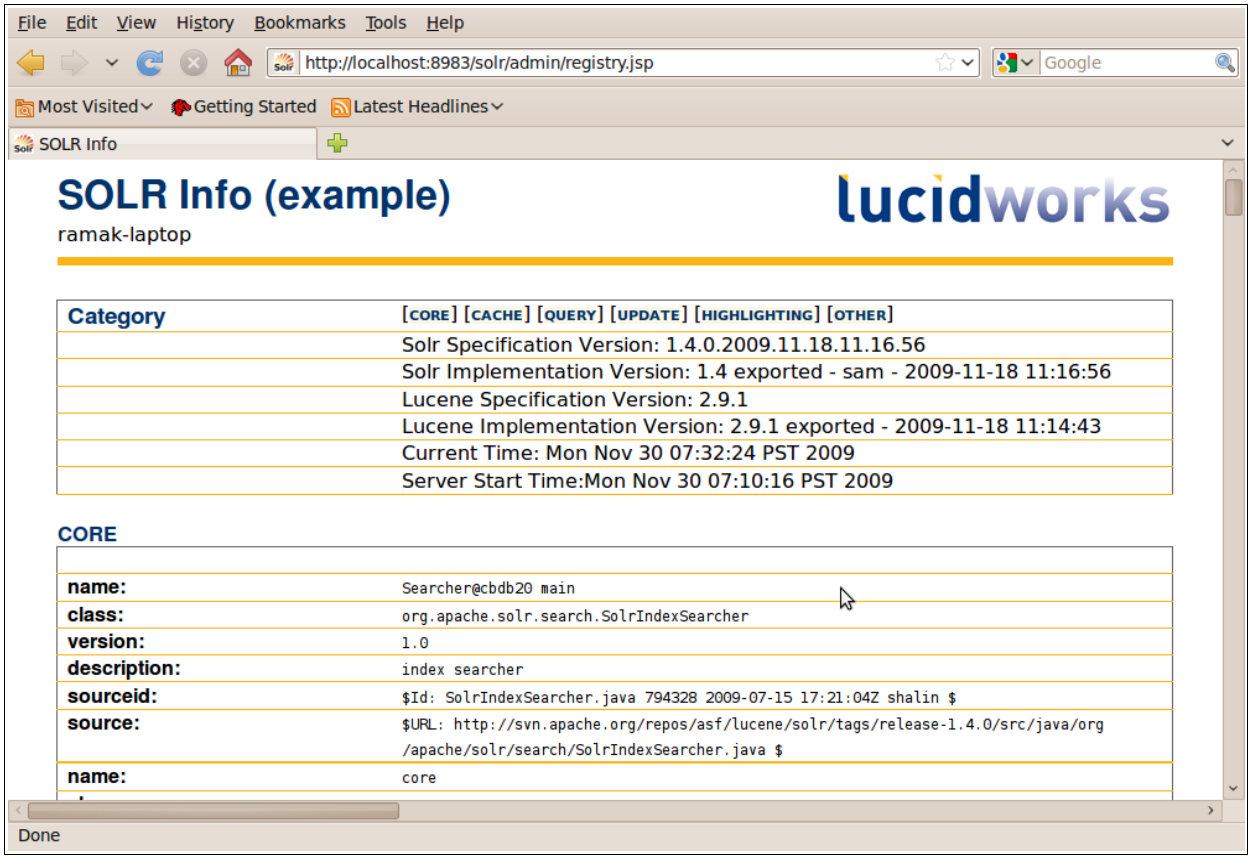


The Solr Statistics page.

The Solr Statistics page groups its data into several sections: core, cache, query, update, highlighting, and other. To jump to the reported data about a particular topic, click on that topic's link (for example, CORE) at the top of the Solr Statistics page.

3.2.6 Displaying Start-up Time Statistics about the Solr Server

To display statistics about the server at start-up time, click the INFO link. Unlike the information displayed by the STATISTICS link, the Solr information displayed by INFO is not continuously updated.

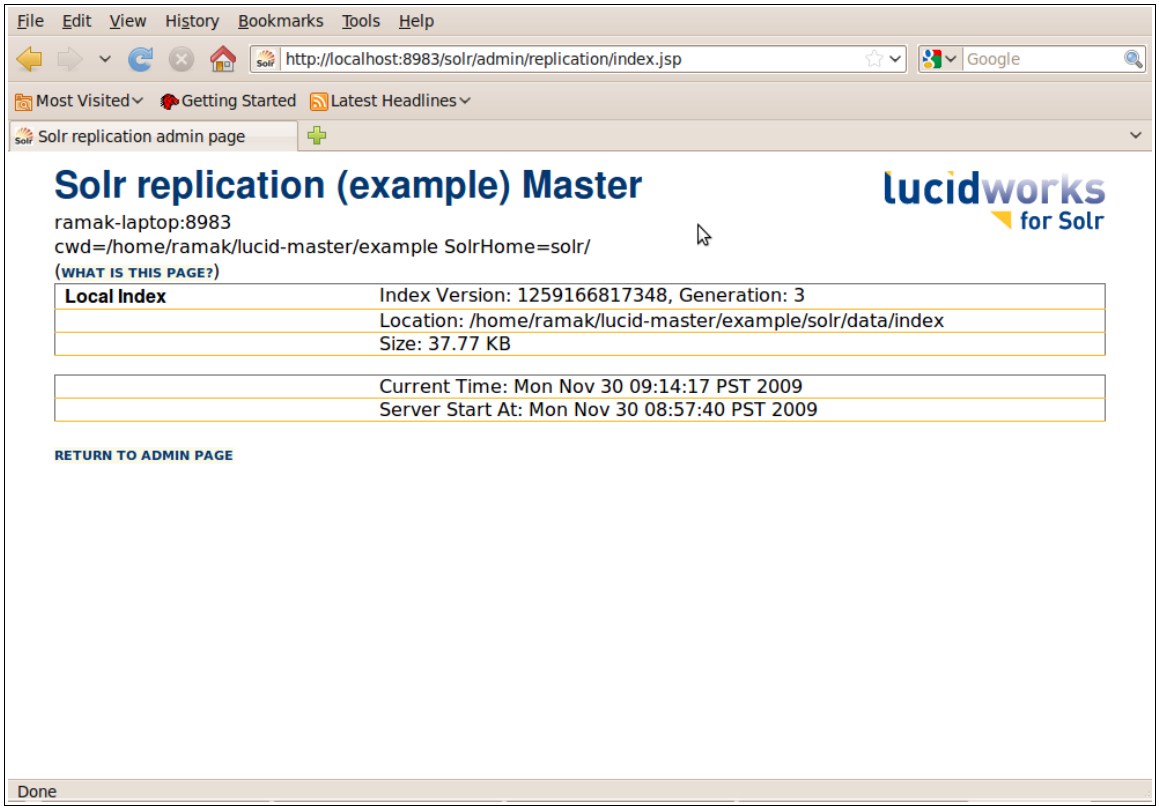


The Solr Info page reports configuration details and statistics.

3.2.7 Displaying Information about a Distributed Solr Configuration

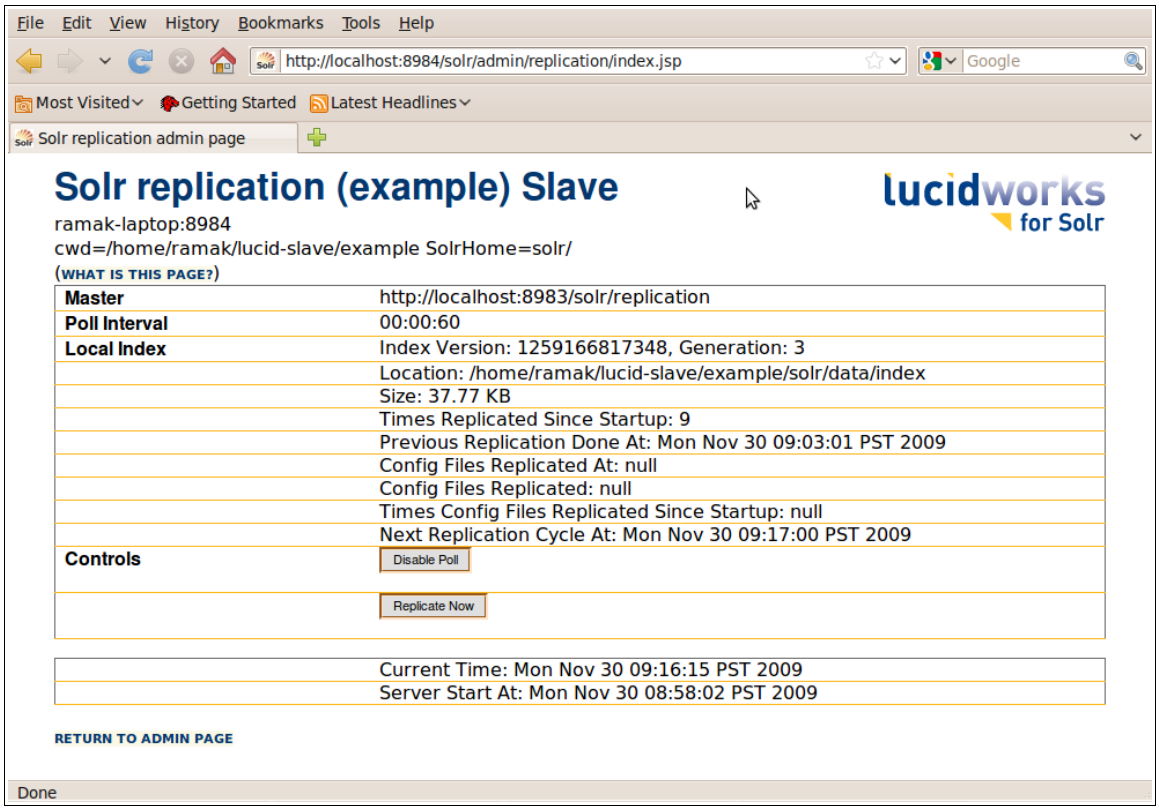
Click the DISTRIBUTION link to see information about master and slave servers. In master/slave configurations, the master server's index is replicated on one or more slave servers, which process queries. (For more information about replicated indexes, see Chapter 10.)

On a master server, the Admin Web interface's Distribution Info reports information about the snapshot of the index being distributed to slave servers.



On a master server, the Distribution Info page identifies the filename of the master server index snapshot and reports on the replication of this snapshot to any slave servers.

On a slave server, the Distribution Info page shows simply information for the slave server itself, as shown below. The page identifies which version of the replicated index the slave server is using. It also reports on the status of the most recent replication process.

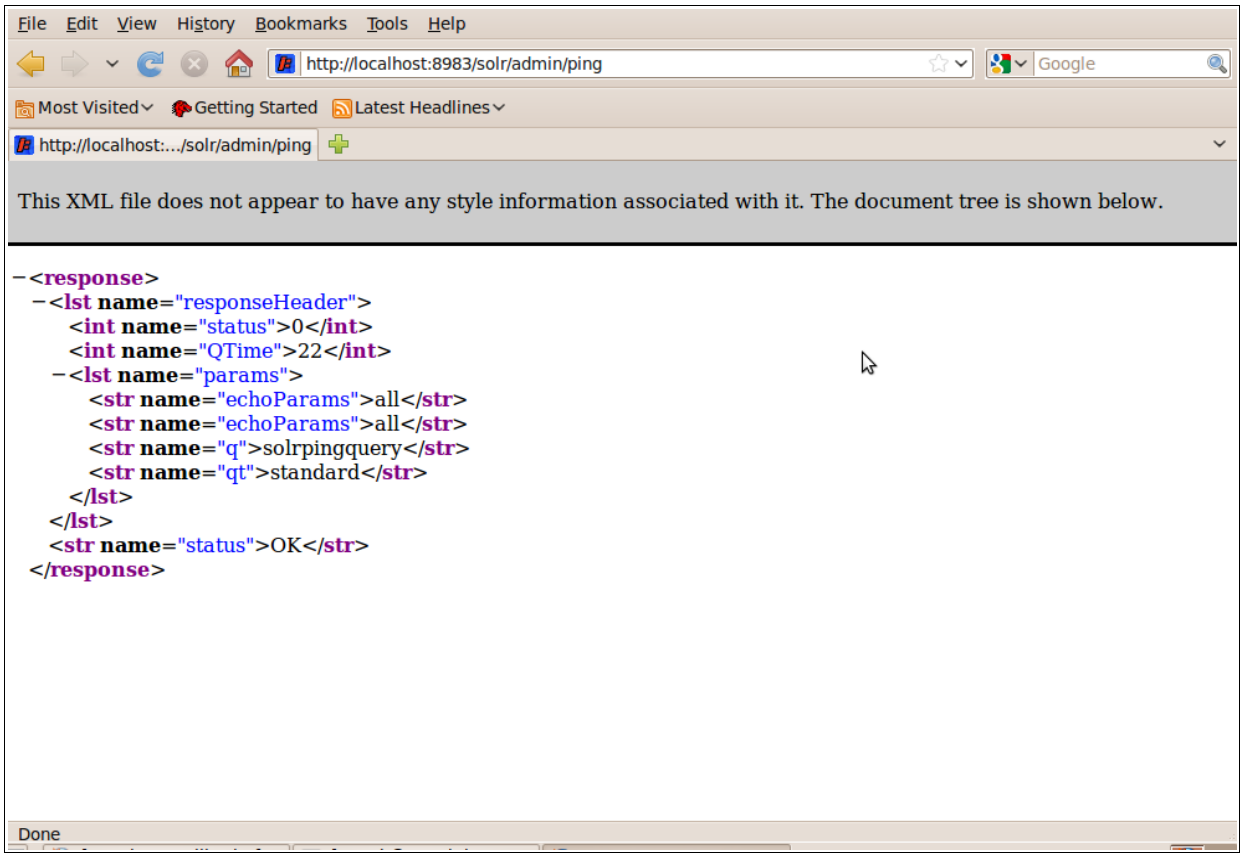


The Distribution Info page for a slave server.

3.2.8 Pinging the Solr Server to Test Its Responsiveness

The `ping` command, which is supported by Windows, Linux, and MacOS, sends a signal to a network-accessible server and reports the time it takes the server to respond, if it responds at all. The command executable is stored at `/admin/ping` on the Solr server. The `ping` command is a straightforward, convenient tool for checking whether or not a server is running.

To run `ping` against the Solr server, click the PING link. If the server is running, the Admin Web interface displays an XML-formatted response like that shown below.

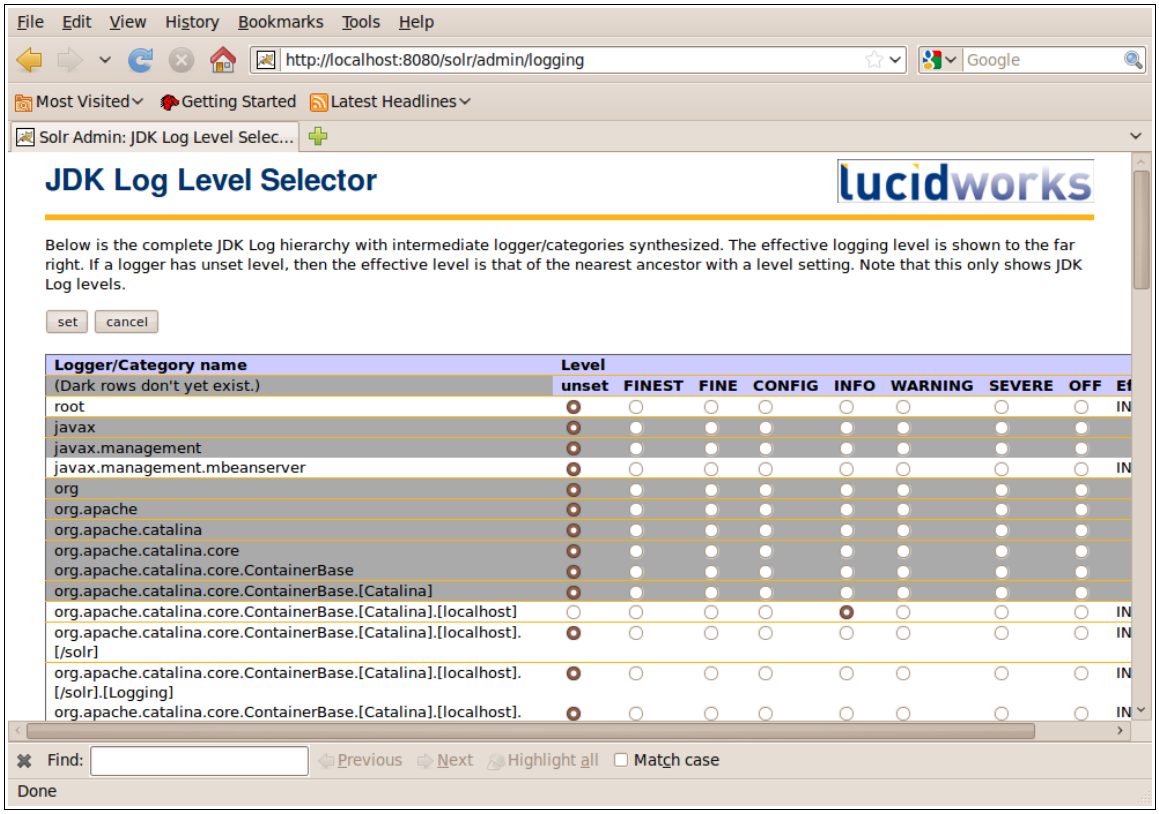


An XML-formatted response to the ping command.

3.2.9 Viewing and Configuring Logfile Settings

Click the LOGGING link to display a long page that offers radio-button settings for JDK logfiles.

NOTE: Any changes you make to logfile settings through Admin Web interface will last only as long as the current Solr session. Once the server is shut down and restarted, settings will revert to the configuration specified in the logfile configuration files.



The JDK Log Level Selector page.

The table below describes the various levels for logging used in JDK logfiles.[1]

Level	Usage
SEVERE	The highest value; intended for extremely important messages (e.g., fatal program errors).
WARNING	Intended for warning messages.
INFO	Informational run-time messages.
CONFIG	Informational messages about configuration settings.
FINE	Used for greater detail when debugging/diagnosing problems.
FINER	Used for even greater detail.
FINEST	The lowest value; provides the greatest detail.
ALL	All messages.
OFF	No messages.

1 See "An Introduction to the Java Logging API," O'Reilly Media, http://www.onjava.com/pub/a/onjava/2002/06/19/log.html

3.3 The App Server Section

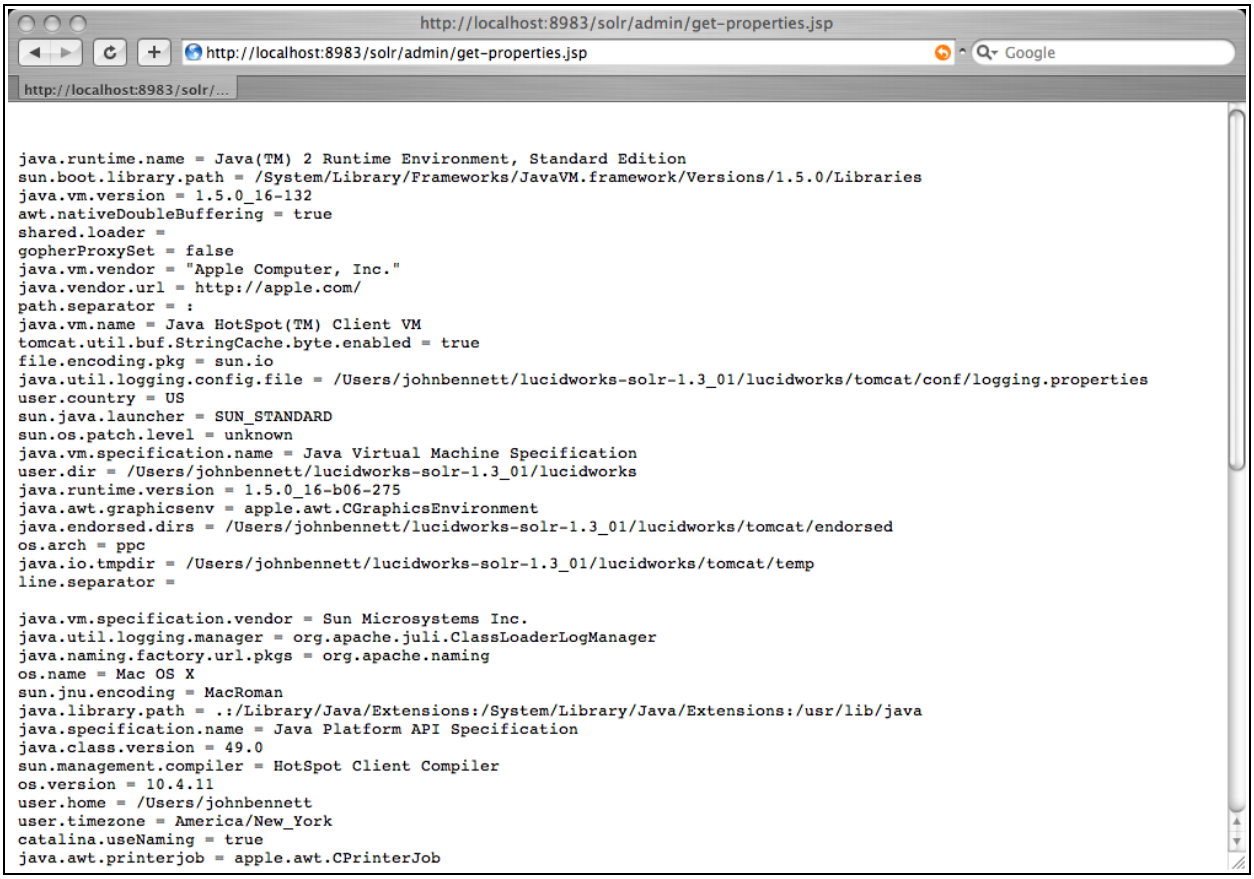
The App Server section of the Admin Web interface always displays a JAVA PROPERTIES link and a THREAD DUMP link. It may also display an ENABLE/DISABLE link, depending on the configuration of the `<admin>` block in the `solrconfig.xml` file.

The table below describes the links in the App Server section.

Link	Description
JAVA PROPERTIES	Displays the properties of the Solr server's Java environment.
THREAD DUMP	Displays a thread dump of the Solr server's Java HotSpot VM.
ENABLE/DISABLE	Enables or disables the Solr application server by creating or removing the file specified in the optional `<healthcheck>` tag in the `<admin>` block of `solrconfig.xml`. (If the `<healthcheck>` tag is absent, the ENABLE/DISABLE link does not appear in the Web interface.) When using load balancers, this feature makes it easy to take a server in or out of rotation by enabling or disabling the server and causing its `healthcheck` to succeed or fail.

3.3.1 Displaying Java Properties

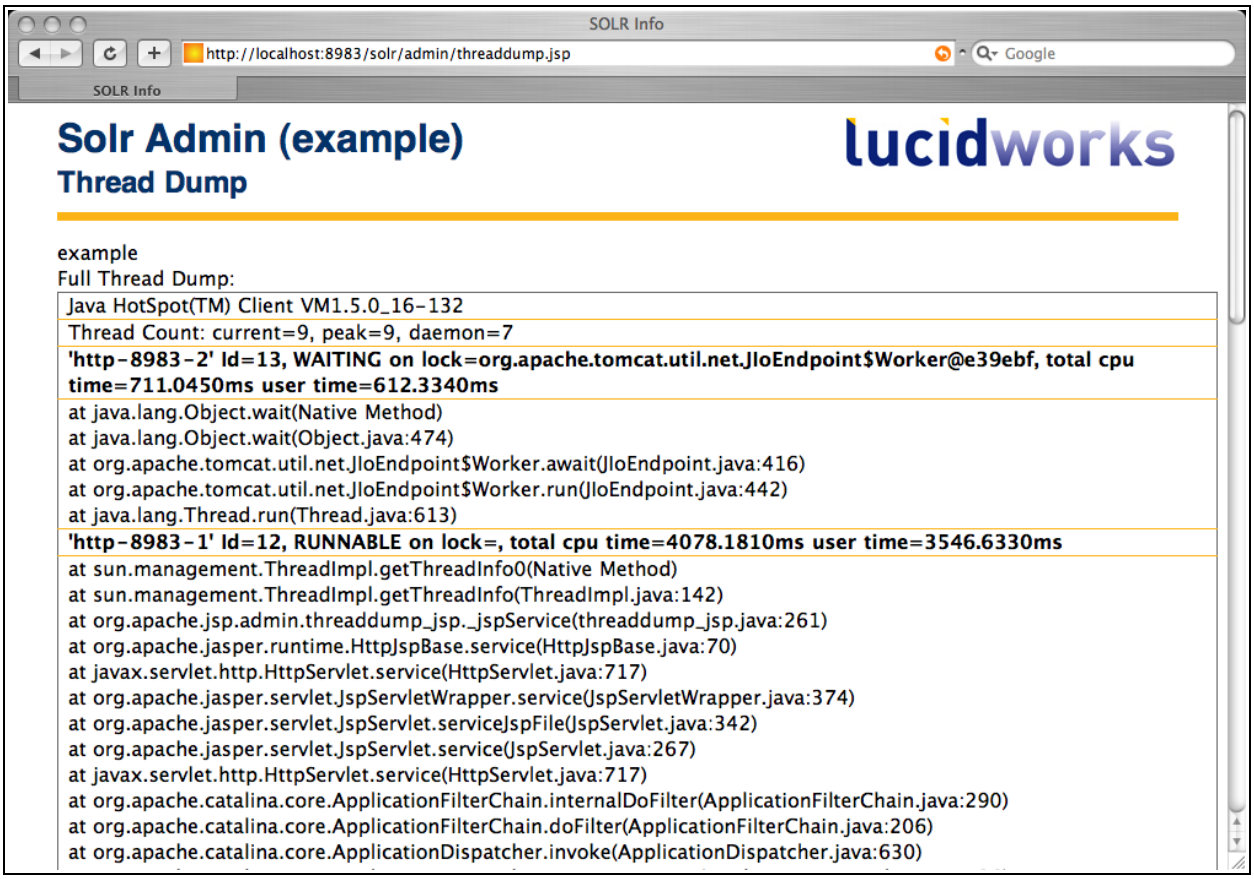
To see the properties of the Java Runtime Environment in which the Solr server is running, click the JAVA PROPERTIES link. The server reports Java configuration details, as shown below.



The Java Properties display.

3.3.2 Displaying the Active Threads in the Java Environment

To see which threads are active in the Java Runtime Environment, click the THREAD DUMP link.



The Thread Dump display.

3.3.3 Enabling or Disabling the Server in a Load-balanced Configuration

This link is only displayed if a `<healthcheck>` directive appears in the `<admin>` block of the `solrconrfig.xml` file. For example:

```
<healthcheck type="file">solr/conf/healthcheck.txt</healthcheck>
```

When using load balancers, the ENABLE/DISABLE link makes it easy to take a server in or out of rotation by making a healthcheck succeed or fail.

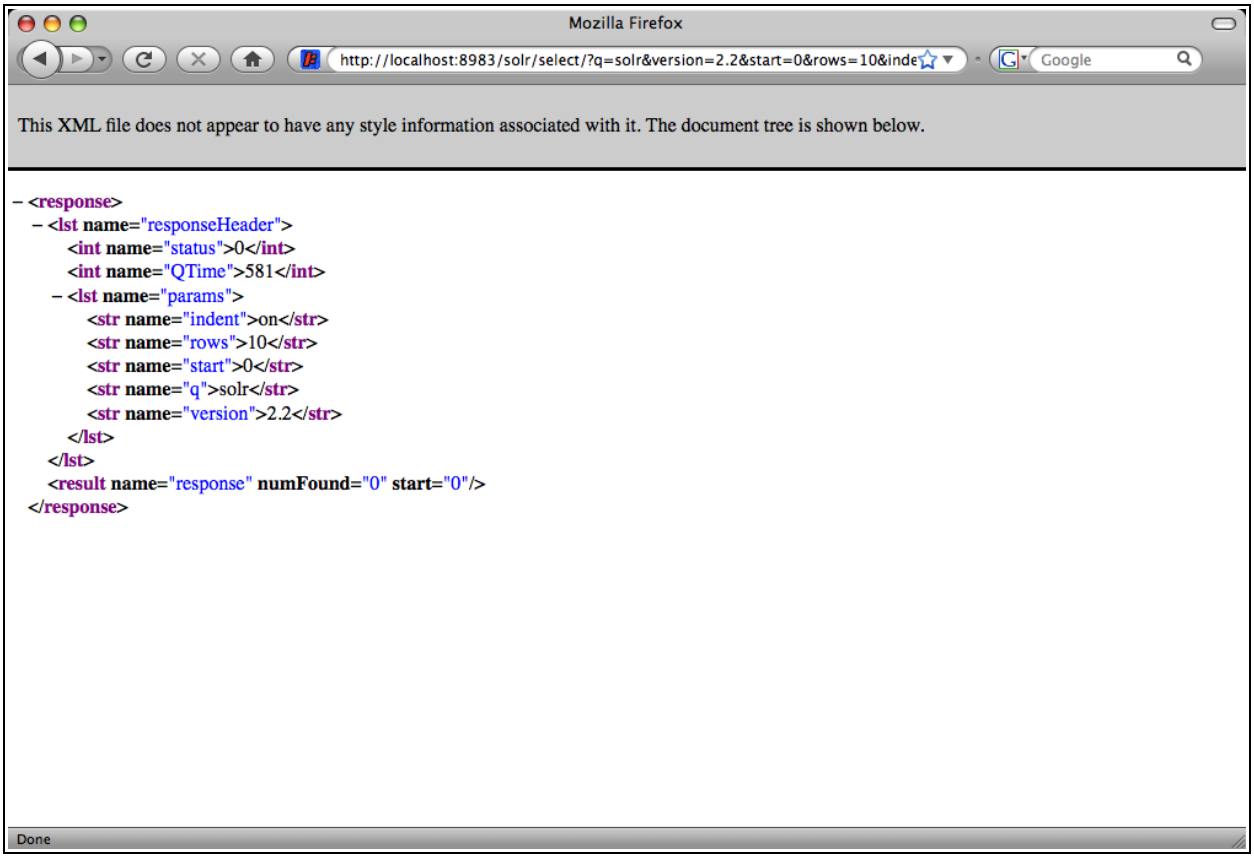
Clicking on ENABLE/DISABLE changes the contents of the `healthcheck` file:

```
http://localhost:8983/solr/admin/file/?file=healthcheck.txt
```

Changing the file toggles the function of the server, either enabling or disabling it for rotation with the load balancer.

3.4 The Make a Query Section

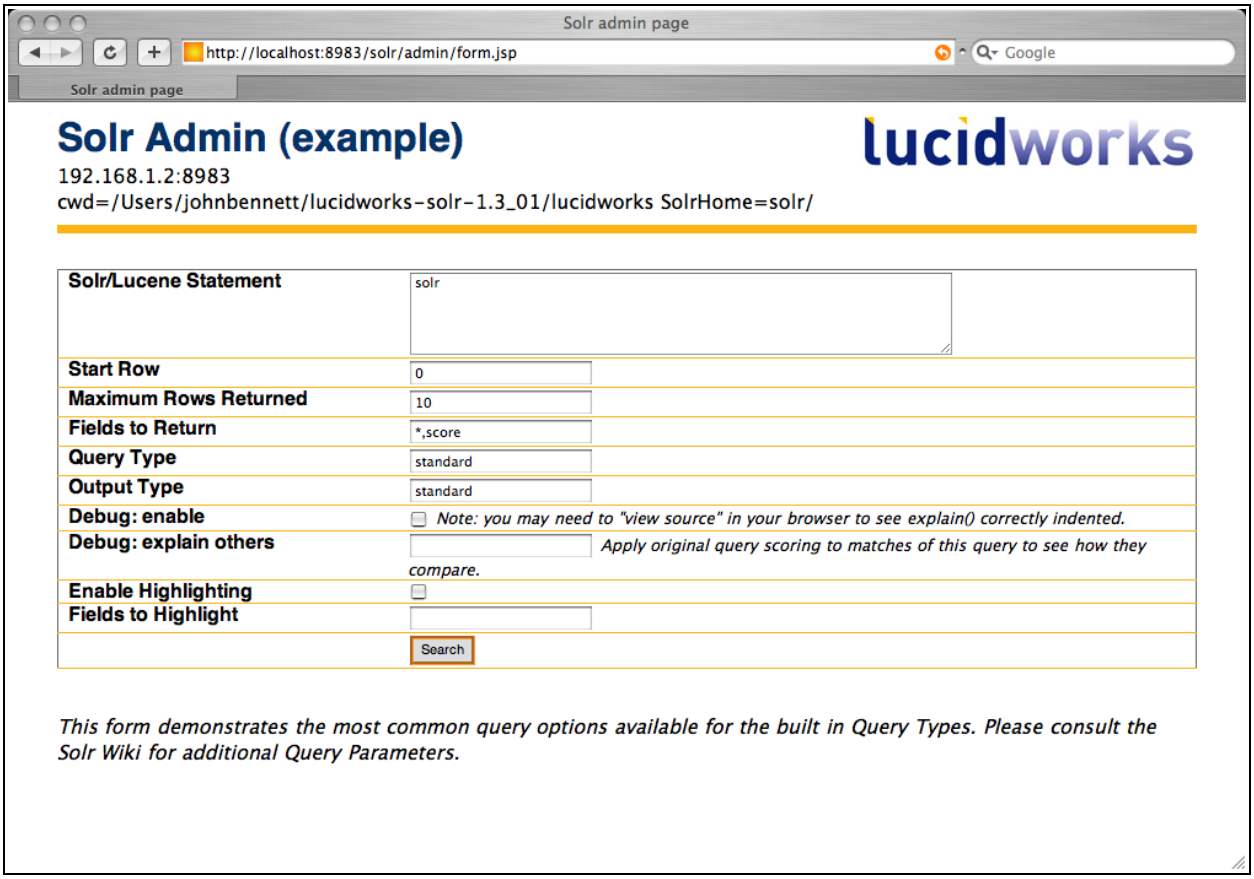
You can use the Make a Query section of the Web interface to submit a search query to the Solr server and analyze the results. The server returns the query results to the browser as XML, as shown in the screenshot below.



Query results are displayed in XML.

3.4.1 Using the Full Interface to Submit Queries

For more control over the details of the query and its response, click the FULL INTERFACE link. The Solr server displays a new page like that shown below.



The Full Search query interface.

The table below explains the fields in this form.

Field	Description
Solr/Lucene Statement	The Lucene/Solr query to be submitted. For a description of query syntax, please see Chapter 7.
Start Row	The offset into the query result starting at which documents should be returned. The default value is `0`, meaning that the query should return results starting with the first document that matches. This field accepts the same syntax as the `start` query parameter, which is described in Chapter 7
Maximum Rows Returned	The number of rows of results that should be displayed at one time for pagination. The default is `10`. Accepts the same syntax as the `rows` query parameter.
Fields to Return	Specifies a list of fields to return. Accepts the same syntax as the `fl` query parameter.
Query Type	Specifies the query handler for the request. If a query handler is not specified, Solr processes the query with the standard query handler.
Response Type	Specifies a response handler for the request. If a response handler is not specified, Solr processes the response with the standard response handler.
Debug: enable	Augments the query response with debugging information, including "explain info" for each document returned. This debugging information is intended to be intelligible to the administrator or programmer.
Debug: explain others	Accepts a Lucene query identifying a set of documents. If non-blank, the "explain info" data of each document matching this query, relative the main query (specified in the *Solr/Lucene Statement* field) will be returned along with the rest of the debugging information.
Enable Highlighting	Causes the query response to highlight the fields specified in the *Fields to Highlight* box in the form.
Fields to Highlight	Specifies which fields in the response to highlight, if highlighting is enabled.

3.5 The Assistance Section

The Assistance section includes the following links.

Link	Description
DOCUMENTATION	Navigates to the Apache Solr documentation hosted on `http://lucene.apache.org/solr/`.
ISSUES	Navigates to the JIRA issue tracking server for the Apache Solr project. This server resides at `http://issues.apache.org/jira/browse/SOLR`.
SEND EMAIL	Invokes the local email client to send email to `solr-user@lucene.apache.org`.
SOLR QUERY SYNTAX	Navigates to the Apache Wiki page describing the Solr query syntax: `http://wiki.apache.org/solr/SolrQuerySyntax`

3.5.1 Summary

The Solr Admin Web interface, which is accessible at the address `http://[hostname]:8983/solr/admin/`, enables you to:

- view Solr configuration details, including the contents of the `schema.xml` file, the `solrconfig.xml` file, master/slave configurations for index replication, and Java logfile settings
- run queries and analyze document fields in order to fine-tune a Solr configuration
- access online documentation and other help

You can configure the Web interface using XML directives and parameters in the `solrconfig.xml` file.

This page is intentionally left blank.

4 Documents, Fields, and Schema Design

4.1 Introduction

The fundamental premise of Solr is simple. You feed it a heaping pile of information, then later you can ask it questions and find the piece of information you want. The part where you feed in all the information is called *indexing* or *updating*. When you ask a question, it's called a *query*.

One way to understand how Solr works is to think of a loose-leaf book of recipes. Every time you add a recipe to the book, you update the index at the back. You list each ingredient and the page number of the recipe you just added. Suppose you add one hundred recipes. Using the index, you can very quickly find all the recipes that use garbanzo beans, or artichokes, or coffee, as an ingredient. Using the index is much faster than looking through each recipe one by one. Imagine a book of one thousand recipes, or one million.

Solr allows you to build an index with many different fields, or types of entries. The example above shows how to build an index with just one field, ingredients. You could have other fields in the index for the recipe's cooking style, like Asian, Cajun, or vegan, and you could have an index field for preparation times. Solr can answer questions like "What Cajun-style recipes that have blood oranges as an ingredient can be prepared in fewer than 30 minutes?"

The schema is the place where you tell Solr how it should build indexes from input documents.

4.2 How Solr Sees the World

Solr's basic unit of information is a *document*, which is a set of information that describes something. A recipe document would contain the ingredients, the instructions, the preparation time, the cooking time, the tools needed, and so on. A document about a person, for example, might contain the person's name, biography, favorite color, and shoe size. A document about a book could contain the title, author, year of publication, number of pages, and so on.

In the Solr universe, documents are composed of *fields*, which are more specific pieces of information. Shoe size could be a field. First name and last name could be fields.

Fields can contain different kinds of data. A name field, for example, is text (character data). A shoe size field might be a floating point number so that it could contain values like 6 and 9.5. Obviously, the definition of fields is flexible—you could define a shoe size field as a text field—but if you define your fields correctly, Solr will be able to interpret them correctly and ultimately, your users will get better results when they perform a query.

You can tell Solr about the kind of data a field contains by specifying its *field type*. The field type tells Solr how to interpret the field and how it can be queried.

When you add a document, Solr takes the information in the document's fields and adds that information to an index. When you perform a query, Solr can quickly consult the index and return the matching documents.

4.3 Field Analysis

Field analysis tells Solr what to do with incoming data when building an index. A more accurate name for this process would be *processing* or even *digestion*, but the official name is *analysis*.

Consider, for example, a biography field in a person document. Every word of the biography must be indexed so that you can quickly find people whose lives have had anything to do with ketchup, or dragonflies, or cryptography.

However, a biography will likely contains lots of words you don't care about and don't want clogging up your index—words like "the," "a," "to," and so forth. Furthermore, suppose the biography contains the word "Ketchup," capitalized at the beginning of a sentence. If a user makes a query for "ketchup," you want Solr to tell you about the person even though the biography contains the capitalized word.

The solution to both these problems is field analysis. For the biography field, you can tell Solr how to break apart the biography into words. You can tell Solr that you want to make all the words lower case, and you can tell Solr to remove accents marks.

Field analysis is an important part of a field type. Chapter 5 is a detailed description of field analysis.

4.4 Solr Field Types

A field type includes four types of information:

- The name of the field type
- An implementation class name
- If the field type is a `TextField`, a description of the field analysis for the field type
- Field attributes

4.4.1 Field Type Definitions in `schema.xml`

In `schema.xml`, the field types are defined in the `types` element. Here is an example of a field type definition:

```
    <fieldType name="textTight" class="solr.TextField"
               positionIncrementGap="100" >
      <analyzer>
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.SynonymFilterFactory"
                synonyms="synonyms.txt" ignoreCase="true" expand="false"/>
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="0" generateNumberParts="0"
                catenateWords="1" catenateNumbers="1" catenateAll="0"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.SnowballPorterFilterFactory"
                language="English" protected="protwords.txt"/>
```

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

The first line in the example above contains the field type name, `textTight`, and the name of the implementing class, `solr.TextField`. The rest of the definition is about field analysis, which will be described in Chapter 5.

The implementing class is responsible for making sure the field is handled correctly. In the class names in `schema.xml`, the string `solr` is shorthand for `org.apache.solr.schema` or `org.apache.solr.analysis`. Therefore, `solr.TextField` is really `org.apache.solr.schema.TextField`.

4.4.2 Field Types Included with Solr

The following table lists the field types that are available in Solr 1.4. The `org.apache.solr.schema` package includes all the classes listed in this table.

Class	Description
`BCDIntField`	Binary-coded decimal (BCD) integer. BCD is a relatively inefficient encoding that offers the benefits of quick decimal calculations and quick conversion to a string.
`BCDLongField`	BCD long integer
`BCDStrField`	BCD string
`BinaryField`	Binary data
`BoolField`	Contains either true or false. Values of "1", "t", or "T" in the first character are interpreted as true. Any other values in the first character are interpreted as false.
`ByteField`	Contains an array of bytes.
`DateField`	Represents a point in time with millisecond precision. See the section below.
`DoubleField`	Double
`ExternalFileField`	Pulls values from a file on disk. See the section below on working with external files.
`FloatField`	Floating point
`IntField`	Integer
`LongField`	Long integer
`RandomSortField`	Does not contain a value. Queries that sort on this field type will return results in random order. Use a dynamic field to use this feature.
`ShortField`	Short integer
`SortableDoubleField`	The `Sortable*` fields provide correct numeric sorting. If you use the plain types (`DoubleField`, `IntField`, etc.) sorting will be lexicographical instead of numeric.
`SortableFloatField`	Numerically sorted floating point
`SortableIntField`	Numerically sorted integer
`SortableLongField`	Numerically sorted long integer
`StrField`	String
`TextField`	Text, usually multiple words or tokens
`TrieDateField`	Date field accessible for Lucene TrieRange processing
`TrieDoubleField`	Double field accessible Lucene TrieRange processing

Class	Description
`TrieField`	If this type is used, a "type" attribute must also be specified, with a value of either: integer, long, float, double, date. Using this field is the same as using any of the Trie*Fields.
`TrieFloatField`	Floating point field accessible Lucene TrieRange processing
`TrieIntField`	Int field accessible Lucene TrieRange processing
`TrieLongField`	Long field accessible Lucene TrieRange processing
`UUIDField`	Universally Unique Identifier (UUID). Pass in a value of "NEW" and Solr will create a new UUID.

4.4.3 Working with Dates

`DateField` represents a point in time with millisecond precision. The format is as follows:

```
YYYY-MM-DD Thh:mm:ssZ

YYYY is the year
MM is the month
DD is the day of the month
hh is the hour of the day as on a 24-hour clock
mm is minutes
ss is seconds
```

Note that no time zone can be specified; the time given should be expressed in Coordinated Universal Time (UTC). Here is an example value:

```
1972-05-20 T17:33:18Z
```

You can include fractional seconds if you wish, although trailing zeros are not allowed and any precision beyond milliseconds will be ignored. Here is another example value with milliseconds included:

```
1972-05-20 T17:33:18.772Z
```

In addition, `DateField` also supports *date math*. This makes it easy to create times relative to the current time. This represents a point in time two months from now:

```
+2MONTHS
```

This is one day ago:

```
-1DAY
```

Use a slash to indicate rounding. This represents the beginning of the current hour:

```
/HOUR
```

You can combine terms. The following is six months and three days in the future, at the beginning of the day:

```
+6MONTHS+3DAYS/DAY
```

4.4.4 Working with External Files

`ExternalFileField` makes it possible to specify field values for documents in a file. For such a field, the file contains mappings from a key field to the field value. Another way to think of this is that, instead of specifying the field in documents as they are indexed, Solr finds values of this field in the external file.

> **NOTE:** External fields are not searchable. They can be used only for function queries. (For more information on function queries, see Chapter 7.)

`ExternalFileField` is handy for cases where you want to update a particular field in many documents more often than you want to update the rest of the documents. For example, suppose you have some kind of document rank based on number of views. You might want to update the rank of all the documents daily or hourly, while the rest of the contents of the documents might be updated much less frequently.

Without `ExternalFileField`, you would need to update each document just to change the rank. Using `ExternalFileField` is much more efficient because all document values for a particular field are stored in an external file that can be updated as frequently as you wish.

An attribute in the field type declaration, `valType`, specifies the actual type of the values that will be found in the file. Note that only float fields are currently supported (`FloatField` type or a subclass).

```
<fieldType name="entryRankFile" keyField="pkId" defVal="0"
```

```
           stored="false" indexed="false"
           class="solr.ExternalFileField" valType="float"/>
```

The file itself is located in Solr's index directory, which by default is `data/index` in the Solr home directory. The name of the file should be `external_<fieldname>` or `external_<fieldname>.*`. For the example above, then, the file could be named `external_entryRankFile` or `external_entryRankFile.txt`.

NOTE: If any files using the name pattern .* appear, the last (after being sorted by name) will be used and previous versions will be deleted. This behavior supports implementations on systems where one may not be able to overwrite a file (for example, on Windows, if the file is in use).

The file contains entries that map a key field, on the left of the equals sign, to a value, on the right. Here are a few example entries:

```
doc33=1.414
doc34=3.14159
doc40=42
```

4.4.5 Field Type Properties

The field type class determines most of the behavior of a field type, but optional properties can also be defined in `schema.xml`. For example, the following definition of a `date` field type defines two properties, `sortMissingLast` and `omitNorms`.

```
<fieldType name="date" class="solr.DateField"
       sortMissingLast="true" omitNorms="true"/>
```

Most properties are either `true` or `false`.

Here are some commonly used properties:

Field Property	Description	Values
`indexed`	If `true`, the value of the field can be used in queries to retrieve matching documents	`true` or `false`
`stored`	If `true`, the actual value of the field can be retrieved by queries	`true` or `false`
`sortMissingFirst` `sortMissingLast`	Control the placement of documents when a sort field is not present	`true` or `false`
`multiValued`	If `true`, indicates that a single document might contain multiple values for this field type	`true` or `false`
`positionIncrementGap`	For multivalued fields, specifies a distance between multiple values, which prevents spurious phrase matches	integer
`omitNorms`	If `true`, omits the norms associated with this field (this disables length normalization and index-time boosting for the field, and saves some memory). Only full-text fields or fields that need an index-time boost need norms.	`true` or `false`
`omitTermFreqAndPositions`	If `true`, omits term frequency, positions, and payloads from postings for this field. This can be a performance boost for fields that don't require that information. It also reduces the storage space required for the index. Queries that rely on position that are issued on a field with this option will silently fail to find documents. This property defaults to true for all fields that are not text fields.	`true` or `false`

4.4.6 Field Properties by Use Case

Here is a summary of available options on a field, broken down by use case. A `true` or `false` indicates that the option must be set to the given value for the use case to function correctly.

Use Case	indexed	stored	multiValued	omitNorms	termVectors	termPositions
search within field	`true`					
retrieve contents		`true`				
use as unique key	`true`		`false`			
sort on field	`true`		`false`	`true` [1]		
use field boosts [5]				`false`		
document boosts affect searches within field				`false`		
highlighting	`true`[4]	`true`			[2]	`true` [3]
faceting [5]	`true`					
add multiple values, maintaining order			`true`			
field length affects doc score				`false`		
`MoreLike`					`true` [6]	

Use Case	indexed	stored	multiValued	omitNorms	termVectors	termPositions
This [5]						

Notes:

1. Recommended but not necessary.
2. Will be used if present, but not necessary.
3. (if `termVectors=true`)
4. A tokenizer must be defined for the field, but it doesn't need to be indexed.
5. Described in Chapter 7
6. Term vectors are not mandatory here If not true, then a stored field is analyzed. So term vectors are recommended, but only required if stored=false.

4.5 Defining Fields

Once you have the field types set up just the way you like, defining the fields themselves is simple. All you do is supply a name and a field type. If you wish, you can also provide options that will override the options for the field type.

Fields are defined in the `fields` element of `schema.xml`. The following example defines a field named `price` with a type of `sfloat`.

```
<field name="price" type="sfloat" indexed="true" stored="true"/>
```

Fields can have the same options as field types. The field type options serve as defaults which can be overridden by options defined per field.

4.6 Copying Fields

You might want to interpret some document fields in more than one way. Solr has a mechanism for making copies of fields so that you can apply several distinct field types to a single piece of incoming information. For you Linux shell geeks, this is something like `tee`.

The name of the field you want to copy is the *source*, and the name of the copy is the *destination*. In `schema.xml`, it's very simple to make copies of fields:

```
<copyField source="cat" dest="text" maxChars="30000" />
```

If the `text` field has data of its own in input documents, the contents of `cat` will be added to the index for `text`. The `maxChars` parameter, a new `int` parameter introduced in Solr 1.4, establishes an upper limit for the number of characters to be copied. This limit is useful for situations in which you want to control the size of index files.

Both the `source` and the `destination` of `copyField` can contain asterisks, which will match anything. For example, the following line will copy the contents of all incoming fields that match the wildcard pattern *t to `the text field.`:

```
<copyField source="*_t" dest="text" maxChars="25000" />
```

NOTE: The `copyField` command can use a wildcard (*) character in the `dest` parameter only if the `source` parameter contains one as well. `copyField` uses the matching glob from the `source` field for the `dest` field name into which the `source` content is copied..

4.7 Dynamic Fields

Dynamic fields allow Solr to index fields that you did not explicitly define in your schema. This is handy if you discover you have forgotten to define one or more fields. Dynamic fields can make your application less brittle by providing some flexibility in the documents you can add to Solr.

A dynamic field is just like a regular field except it has a name with a wildcard in it. When you are indexing documents, a field that does not match any explicitly defined fields can be matched with a dynamic field.

For example, suppose your schema includes a dynamic field with a name of `*_i`. If you attempt to index a document with a `cost_i` field, but no explicit `cost_i` field is defined in the schema, then the `cost_i` field will have the field type and analysis defined for `*_i`.

Dynamic fields are also defined in the `fields` element of `schema.xml`. Like fields, they have a name, a field type, and options.

```
<dynamicField name="*_i" type="sint" indexed="true"  stored="true"/>
```

Lucid Imagination recommends that you include basic dynamic field mappings (like that shown above) in your `schema.xml`. The mappings can come in handy.

4.8 Other Schema Elements

This section describes several other important elements of `schema.xml`.

4.8.1 Unique Key

The `uniqueKey` element specifies which field is a unique identifier for documents. Although `uniqueKey` is not required, it is nearly always warranted by your application design. For example, `uniqueKey` should be used if you will ever update a document in the index.

For more information, consult the Wiki:

http://wiki.apache.org/solr/UniqueKey

You can define the unique key field by naming it:

```
<uniqueKey>id</uniqueKey>
```

4.8.2 Default Search Field

If you are using the Lucene query parser, queries that don't specify a field name will use the `defaultSearchField`. The `dismax` query parser does not use this value. (For more information about query parsers, see Chapter 7.)

Just name a field to use it as the default search field:

```
<defaultSearchField>text</defaultSearchField>
```

4.8.3 Query Parser Operator

In queries with multiple terms, Solr can either return results where all conditions are met or where one or more conditions are met. The *operator* controls this behavior. An operator of `AND` means that all conditions must be fulfilled, while an operator of `OR` means that one or more conditions must be true.

In `schema.xml`, use the `solrQueryParser` element to control what operator is used if an operator is not specified in the query. The default operator setting only applies to the Lucene query parser (not the DisMax query parser, which internally hard-codes its operator to OR)

```
<solrQueryParser defaultOperator="OR"/>
```

4.9 Putting the Pieces Together

At the highest level, `schema.xml` is structured as follows. This example is not real XML, but it gives you an idea of the important parts of the file.

```
<schema>
  <types>
  <fields>

  <uniqueKey>
  <defaultSearchField>
  <solrQueryParser defaultOperator>

  <copyField>
</schema>
```

Obviously, most of the excitement is in `types` and `fields`, where the field types and the actual field definitions live. These are supplemented by `copyFields`. Sandwiched between `fields` and the `copyField` section are the unique key, default search field, and the default query operator.

4.9.1 Choosing Appropriate Numeric Types

For general numeric needs, use the sortable field types, `SortableIntField`, `SortableLongField`, `SortableFloatField`, and `SortableDoubleField`. These field types will sort numerically instead of lexicographically, which is the main reason they are preferable over their simpler cousins, `IntField`, `LongField`, `FloatField`, and `DoubleField`.

If you expect users to make frequent range queries on numeric types, consider using `TrieField`. It offers faster speed for range queries at the expense of increasing index size. This feature is new in Solr 1.4.

4.9.2 Working With Text

Handling text properly will make your users happy by providing them with the best possible results for text searches.

One technique is using a text field as a catch-all for keyword searching. Most users are not sophisticated about their searches and the most common search is likely to be a simple keyword search. You can use `copyField` to take a variety of fields and funnel them all into a single text field for keyword searches. In the example schema representing a store, `copyField` is used to dump the contents of `cat`, `name`, `manu`, `features`, and `includes` into a single field, `text`. In addition, it could be a good idea to copy `id` into `text` in case users wanted to search for a particular product by passing its product number to a keyword search.

Another technique is using `copyField` to use the same field in different ways. Suppose you have a field that is a list of authors, like this:

```
Schildt, Herbert; Wolpert, Lewis; Davies, P.
```

For searching by author, you could tokenize the field, convert to lower case, and strip out punctuation:

```
schildt / herbert / wolpert / lewis / davies / p
```

For sorting, just use an untokenized field, converted to lower case, with punctuation stripped:

```
schildt herbert wolpert lewis davies p
```

Finally, for faceting, use the primary author only via a `StringField`:

```
Schildt, Herbert
```

4.10 Summary

To build a searchable index, Solr takes in documents composed of data fields of specific field types. The configuration file `schema.xml` defines field types and specific fields that your documents can contain. The `schema.xml` file also describes how Solr should handle those fields when adding documents to the index or when querying those fields.

Now that you have a basic understanding of fields, field types, and documents, you're ready to learn about documents and filters, which are the topics of the next chapter.

5 Understanding Analyzers, Tokenizers, and Filters

5.1 Introduction

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

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

Although the analysis process is used for both indexing and querying, the same analysis process need not be used for both operations (see Section 5.2.1). For indexing, you often want to simplify, or normalize, words. For example, setting all letters to lowercase, eliminating punctuation and accents, mapping words to their stems, and so on. Doing so can increase recall because, for example, “ram”, “Ram” and “RAM” would all match a query for “ram”. To increase query-time precision, a filter could be employed to narrow the matches by, for example, ignoring all-cap acronyms if you're interested in male sheep, but not Random Access Memory.

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

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

5.2 What Is An Analyzer?

Analyzers are specified as a child of the `<fieldType>` element in the `schema.xml` config file that can be found in the `solr/conf` directory, or wherever `solrconfig.xml` is located.

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

For example:

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

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

Even the most complex analysis requirements can usually be decomposed into a series of discrete, relatively simple processing steps. As you will soon discover in Sections 5.5 and 5.6 , the LucidWorks Solr distribution comes with a large selection of tokenizers and filters that covers most scenarios you are likely to encounter. Setting up an analyzer chain is very straightforward; you specify a simple

`<analyzer>` element (no `class` attribute) with child elements that name factory classes for the tokenizer and filters to use, in the order you want them to run.

For example:

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

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

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

5.2.1 Analysis Phases

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

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

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

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

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

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

5.3 What Is A Tokenizer?

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

Characters in the input stream may be discarded, such as whitespace or other delimiters. They may also be added to or replaced, such as mapping aliases or abbreviations to normalized forms. A token contains various metadata in addition to its text value, such as the location at which the token occurs in the field. Because a tokenizer may produce tokens that diverge from the input text, you should not assume that the

text of the token is the same text that occurs in the field, or that its length is the same as the original text. It's also possible for more than one token to have the same position or refer to the same offset in the original text. Keep this in mind if you use token metadata for things like highlighting search results in the field text.

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

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

5.4 What Is a Filter?

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

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

Because filters consume one `TokenStream` and produce a new `TokenStream`, they can be chained one after another indefinitely. Each filter in the chain in turn processes the tokens produced by its

predecessor. The order in which you specify the filters is therefore significant. Typically, the most general filtering is done first, and later filtering stages are more specialized.

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

This example starts with Solr's standard tokenizer (Section 5.5.1) which breaks the field's text into tokens. Those tokens then pass through Solr's standard filter (Section 5.6.17) which removes dots from acronyms, and a few other common operations. All the tokens are then set to lowercase (Section 5.6.8), which will facilitate case-insensitive matching at query time.

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

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

Word stemming is, obviously, very language specific. LucidWorks for Solr includes several language-specific stemmers created by the Snowball[2] generator that are based on the Porter stemming algorithm. The generic Snowball Porter Stemmer Filter (Section 5.6.16) can be used to configure any of these language stemmers. LucidWorks for Solr also includes a convenience wrapper for the English Snowball stemmer. There are also several purpose-built stemmers for non-English languages. These stemmers are described in Language Analysis, Section 5.8 .

2 http://snowball.tartarus.org/

5.5 Tokenizers

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

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

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

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

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

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

5.5.1 Standard Tokenizer

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

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

Factory class: `solr.StandardTokenizerFactory`

Arguments: None

Example:

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

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

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

5.5.2 HTML Strip Standard Tokenizer

This tokenizer treats the input text as HTML/XML. It strips out markup information and then tokenizes the text content with the Standard Tokenizer (see above). The field need not be well-formed HTML or XML—the tokenizer will process anything that looks like HTML or XML input.

> **NOTE:** As of Solr 1.4, the use of this tokenizer is deprecated. We recommend that you use HTMLStripCharFilter (Section 5.7) instead.

This tokenizer does the following:

- Removes HTML/XML tags, including attributes, and keeps the content. Attributes are recognized even if their values are not quoted.
- XML processing instructions are removed, such as: `<?xml version="1.0"?>`
- HTML/XML comments are removed, such as: `<!-- this is a comment -->`
  In fact, any pattern that starts with `<!` and ends with `>` is discarded.
- Removes <script> and <style> elements including embedded comments within them.
- Replaces numeric entity references with the corresponding character: `&#x58;` →X
- Replaces any HTML 4 named entities with its corresponding character: `&copy;` →©
  Exception: ` ` is replaced with a simple space, rather than `&#a0;`.

Factory class: `solr.HTMLStripStandardTokenizerFactory`

Arguments: None

Example:

Default behavior:

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

In: "<html><body><h1>Test</h1> four score </body></html>"

Out: "Test", "four", "score"

Example:

Markup removed, entity   becomes a whitespace delimiter, punctuation discarded.

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

In: "<rocky beer="Blatz">Yo Adrienne!</rocky>"

Out: "Yo", "Adrienne"

Example:

Using entities for < and >. Those chars are not stripped as HTML. Instead they become delimiters for the Standard Tokenizer and so the element names become tokens.

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

In: "Bold text"

Out: "b", "Bold", "text", "b"

5.5.3 HTML Strip White Space Tokenizer

This tokenizer performs the same HTML stripping as the HTML Strip Standard tokenizer (above), except that it then applies the Whitespace Tokenizer (Section 5.5.8) after removing markup.

> **NOTE:** As of Solr 1.4, use of this tokenizer is deprecated. We recommend that you use HTMLStripCharFilterFactory (Section 5.7) instead.

Factory class: solr.HTMLStripWhitespaceTokenizerFactory

Arguments: None

Example:

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

In: "<html><body><h1>Test</h1> four score </body></html>"

Out: "Test", "four", "score"

Example:

Markup removed, entity   becomes a whitespace delimiter, punctuation *not* discarded.

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

In: "<rocky beer="Blatz">Yo Adrienne!</rocky>"

Out: "Yo", "Adrienne!"

Example:

Using entities for < and >. Those chars are not stripped as HTML, so they become delimiters for the Whitespace Tokenizer.

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

In: "Bold text"
Out: "<", "b>", "Bold", "text<", "/b>"

5.5.4 Lower Case Tokenizer

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

Factory class: `solr.LowerCaseTokenizerFactory`

Arguments: None

Example:

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

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

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

5.5.5 N-Gram Tokenizer

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

Factory class: `solr.NGramTokenizerFactory`

Arguments:

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

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

Example:

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

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

In: "hey man"

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

Example:

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

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

In: "bicycle"

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

5.5.6 Edge N-Gram Tokenizer

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

Factory class: `solr.EdgeNGramTokenizerFactory`

Arguments:

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

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

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

Example:

Default behavior (min and max default to 1):

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

In: "`babaloo`"

Out: "`b`"

Example:

Edge n-gram range of 2 to 5

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

In: "`babaloo`"

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

Example:

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

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

In: "`babaloo`"

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

5.5.7 Regular Expression Pattern Tokenizer

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

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

Factory class: `solr.PatternTokenizerFactory`

Arguments:

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

`group`: (Optional, default -1) Specifies which regex group to extract as the token(s).

The value -1 means the regex should be treated as a delimiter that separates tokens.

Non-negative group numbers (>= 0) indicate that character sequences matching that regex group should be converted to tokens. Group zero refers to the entire regex, groups greater than zero refer to parenthesized sub-expressions of the regex, counted from left to right.

Example:

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

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

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

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

Example:

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

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

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

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

Example:

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

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

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

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

5.5.8 White Space Tokenizer

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

Factory class: `solr.WhitespaceTokenizerFactory`

Arguments: None

Example:

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

In: "`To be, or what?`"

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

5.6 Filter Descriptions

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

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

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

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

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

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

5.6.1 Double Metaphone Filter

This filter applies Lawrence Phillips' Double Metaphone algorithm[3] to produce phonetic encodings for strings.

3 Originally described in the June 2000 of the *C/C++ Users Journal*. See http://en.wikipedia.org/wiki/Double_Metaphone for an overview.

NOTE: The standalone Double Metaphone filter was deprecated in Solr 1.3. Users were encouraged to use the Phonetic Filter with the option `encoder="DoubleMetaphone"` instead (see Section 5.6.12). This new Double Metaphone filter offers more control over the Double Metaphone encoder. It exposes the `maxCodeLength` setting. It checks to see if there is an alternate encoding for a token and if there is, adds that alternate encoding to the stream, if the alternate encoding is different than the default encoding.

Factory class: solr.DoubleMetaphoneFilterFactory

Arguments:

`input`

`maxCodeLength` (integer, default 4)

`inject` (boolean, default false)

Example:

```
<fieldtype name="phonetic" stored="false" indexed="true"
class="solr.TextField" >
     <analyzer>
     <tokenizer class="solr.StandardTokenizerFactory"/>
     <filter class="solr.DoubleMetaphoneFilterFactory" inject="false"/>
   </analyzer>
</fieldtype>
```

5.6.2 Edge N-Gram Filter

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

Factory class: `solr.EdgeNGramFilterFactory`

Arguments:

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

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

Example:

Default behavior.

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

In: "four score and twenty"

T→F: "four", "score", "and", "twenty"

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

Example:

A range of 1 to 4.

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

In: "four score"

T→F: "four", "score"

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

Example:

A range of 4 to 6.

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

In: "four score and twenty"

T→F: "four", "score", "and", "twenty"

Out: "four", "sco", "scor"

5.6.3 English Porter Stemming Filter

This filter applies the Porter Stemming Algorithm for English. It is equivalent to the Snowball Porter Stemmer (Section 5.6.16) with the `language="English"` argument.

Factory class: `solr.EnglishPorterFilterFactory`

Arguments:

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

Example:

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

In: "`hop hopping hopped`"

T→F: "`hop`", "`hopping`", "`hopped`"

Out: "`hop`", "`hop`", "`hop`"

5.6.4 Hyphenated Words Filter

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

Factory class: `solr.HyphenatedWordsFilterFactory`

Arguments: None

Example:

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

In: "`A hyphen- ated word`"

T→F: "`A`", "`hyphen-`", "`ated`", "`word`"

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

5.6.5 Keep Words Filter

This filter discards all tokens except those that are listed in the given word list. This is the

inverse of the Stop Words Filter (Section 5.6.18). This filter can be useful for building specialized indices for a constrained set of terms.

Factory class: `solr.KeepWordFilterFactory`

Arguments:

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

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

Example:

With `keepwords.txt` contains:

```
happy
funny
silly
```

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

In: "Happy, sad or funny"

T→F: "Happy", "sad", "or", "funny"

Out: "funny"

Example:

Same `keepwords.txt`, case insensitive:

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

In: "Happy, sad or funny"

T→F: "Happy", "sad", "or", "funny"

Out: "Happy", "funny"

Example:

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

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

In: "Happy, sad or funny"

T→F: "Happy", "sad", "or", "funny"

F→F: "happy", "sad", "or", "funny"

Out: "happy", "funny"

5.6.6 KStemmer

KStem is an alternative to the Porter Stemmer for developers looking for a less aggressive stemmer. KStem was written by Bob Krovetz, ported to Lucene by Sergio Guzman-Lara (UMASS Amherst).

5.6.6.1 LucidKStemmer

LucidWorks for Solr contains an already-integrated Kstemmer version which has been heavily optimized. Large field performance shows a 220% performance increase, while small fields show a 1140% increase compared to the original UMASS code.

The `schema.xml` file included with LucidWorks for Solr includes the following definition of a text field using KStemmer:

```
 <!-- a basic general purpose text field utilizing KStemmer -->
    <fieldType name="text_kstem" class="solr.TextField"
positionIncrementGap="100">
      <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
words="stopwords.txt" enablePositionIncrements="false" />
        <filter class="solr.WordDelimiterFilterFactory"
generateWordParts="1" generateNumberParts="1" catenateWords="1"
catenateNumbers="1" catenateAll="0" splitOnCaseChange="1"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- The LucidKStemmer currently requires a lowercase filter
somewhere before it. -->
```

```
        <filter
class="com.lucidimagination.solrworks.analysis.LucidKStemFilterFactory"
protected="protwords.txt"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
words="stopwords.txt"/>
        <filter class="solr.WordDelimiterFilterFactory"
generateWordParts="1" generateNumberParts="1" catenateWords="0"
catenateNumbers="0" catenateAll="0" splitOnCaseChange="1"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- The LucidKStemmer currently requires a lowercase filter
somewhere before it. -->
        <filter
class="com.lucidimagination.solrworks.analysis.LucidKStemFilterFactory"
protected="protwords.txt"/>
      </analyzer>
    </fieldType>
```

5.6.7 Length Filter

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

Factory class: `solr.LengthFilterFactory`

Arguments:

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

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

Example:

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

In: "`turn right at Albuquerque`"

T→F: "`turn`", "`right`", "`at`", "`Albuquerque`"

Out: "`turn`", "`right`"

5.6.8 Lower Case Filter

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

Factory class: `solr.LowerCaseFilterFactory`

Arguments: (None)

Example:

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

In: "`Down With CamelCase`"

T→F: "`Down`", "`With`", "`CamelCase`"

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

5.6.9 N-Gram Filter

Generates n-gram tokens of sizes in the given range.

Factory class: `solr.NGramFilterFactory`

Arguments:

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

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

Example:

Default behavior.

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

In: "`four score`"

T→F: "`four`", "`score`"

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

Example:

A range of 1 to 4.

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

In: "four score"

T→F: "four", "score"

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

Example:

A range of 3 to 5.

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

In: "four score"

T→F: "four", "score"

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

5.6.10 Numeric Payload Token Filter

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

Factory class: `solr.NumericPayloadTokenFilterFactory`

Arguments:

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

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

Example:

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

In: "bing bang boom"

T→F: "bing", "bang", "boom"

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

5.6.11 Pattern Replace Filter

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

Factory class: `solr.PatternReplaceFilter`

Arguments:

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

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

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

Example:

Simple string replace:

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

In: "cat concatenate catycat"

T→F: "cat", "concatenate", "catycat"

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

Example:

String replacement, first occurrence only:

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

In: "cat concatenate catycat"

T→F: "cat", "concatenate", "catycat"

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

Example:

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

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

In: "cat foo1234 9987 blah1234foo"

T→F: "cat", "foo1234", "9987", "blah1234foo"

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

5.6.12 Phonetic Filter

This filter creates tokens using one of the phonetic encoding algorithms in the `org.apache.commons.codec.language` package.

Factory class: `solr.PhoneticFilterFactory`

Arguments:

`encoder`: (required) The name of the encoder to use. The encoder name must be one of the following (case insensitive):

"`DoubleMetaphone`", "`Metaphone`", "`Soundex`" or "`RefinedSoundex`"

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

Example:

Default behavior for DoubleMetaphone encoding.

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

In: "four score and twenty"

T→F: "four"(1), "score"(2), "and"(3), "twenty"(4)

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

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

Example:

Discard original token.

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

In: "four score and twenty"

T→F: "four"(1), "score"(2), "and"(3), "twenty"(4)

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

Example:

Default Soundex encoder.

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

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

In: "four score and twenty"

T→F: "four"(1), "score"(2), "and"(3), "twenty"(4)

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

5.6.13 Porter Stem Filter

This filter applies the Porter Stemming Algorithm for English. The results are similar to using the Snowball Porter Stemmer with the `language="English"` argument (Section 5.6.16). But this stemmer is coded directly in Java and is not based on Snowball. Nor does it accept a list of protected words. This stemmer is only appropriate for English language text.

Factory class: `solr.PorterStemFilterFactory`

Arguments: None

Example:

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

In: "jump jumping jumped"

T→F: "jump", "jumping", "jumped"

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

5.6.14 Remove Duplicates Token Filter

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

Factory class: `solr.RemoveDuplicatesTokenFilterFactory`

Arguments: (None)

Example:

This is an artificial example that uses the Synonym Filter (Section 5.6.19) to generate duplicate symbols, which are then removed. The file `testsyns.txt` contains the following:

```
blurt => foo,foo
blort => bar,bar
```

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

In: "`blurt blort`"

T→F: "`blurt`"(1), "`blurt`"(2)

T→F: "`foo`"(1), "`foo`"(1), "`bar`"(2), "`bar`"(2)

Out: "`foo`"(1), "`bar`"(2)

5.6.15 Shingle Filter

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

Factory class: `solr.ShingleFilterFactory`

Arguments:

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

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

Example:

Default behavior.

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

In: "`To be, or what?`"

T→F: "`To`"(1), "`be`"(2), "`or`"(3), "`what`"(4)

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

Example:

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

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

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

T→F: "To"(1), "be"(2), "or"(3), "not"(4), "to"(5), "be"(6)

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

5.6.16 Snowball Porter Stemmer Filter

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

This release of LucidWorks for Solr contains Snowball stemmers for English, Danish, Dutch, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish and Turkish.

For more information on Snowball, visit http://snowball.tartarus.org/.

Factory class: `solr.SnowballPorterFilterFactory`

Arguments:

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

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

Example:

Default behavior:

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

In: "`flip flipped flipping`"

T→F: "`flip`", "`flipped`", "`flipping`"

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

Example:

French stemmer, English words:

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

In: "`flip flipped flipping`"

T→F: "`flip`", "`flipped`", "`flipping`"

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

Example:

Spanish stemmer, Spanish words:

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

In: "`cante canta`"

T→F: "`cante`", "`canta`"

Out: "cant", "cant"

5.6.17 Standard Filter

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

Factory class: `solr.StandardFilterFactory`

Arguments: (None)

Example:

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

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

T→F: "Bob's", "I.O.U."

Out: "Bob". "IOU"

5.6.18 Stop Filter

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

Factory class: `solr.StopFilterFactory`

Arguments:

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

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

`enablePositionIncrements`: (true/false, default false) When true, if a token is stopped (discarded) then the position of the following token is incremented.

Example:

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

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

In: "To be or what?"

T→F: "To"(1), "be"(2), "or"(3), "what"(4)

Out: "To"(1), "what"(2)

Example:

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

In: "To be or what?"

T→F: "To"(1), "be"(2), "or"(3), "what"(4)

Out: "what"(1)

Example:

Position increment enabled, original positions retained. No tokens at positions of stopped words.

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

In: "You are a star"

T→F: "You"(1), "are"(2), "a"(3), "star"(4)

Out: "You"(1), "star"(4)

5.6.19 Synonym Filter

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

Factory class: `solr.SynonymFilterFactory`

Arguments:

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

There are two ways to specify synonym mappings:

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

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

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

Example:

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

In: "`teh small couch`"

T→F: "`teh`"(1), "`small`"(2), "`couch`"(3)

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

Example:

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

In: "teh ginormous, humungous sofa"

T→F: "teh"(1), "ginormous"(2), "humungous"(3), "sofa"(4)

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

5.6.20 Token Offset Payload Filter

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

Factory class: `solr.TokenOffsetPayloadTokenFilterFactory`

Arguments: None

Example:

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

In: "bing bang boom"

T→F: "bing", "bang", "boom"

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

5.6.21 Trim Filter

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

Factory class: `solr.TrimFilterFactory`

Arguments:

`updateOffsets`: (true/false, default false) If true, the token's start/end offsets are adjusted to account for any whitespace that was removed.

Example:

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

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

In: "`one, two , three ,four `"

T→F: "`one`", "` two `", "` three `", "`four `"

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

5.6.22 Type As Payload Filter

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

Factory class: `solr.TypeAsPayloadTokenFilterFactory`

Arguments: None

Example:

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

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

T→F: "`Pay`", "`Bob's`", "`I.O.U.`"

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

5.6.23 Word Delimiter Filter

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

- A change in case within a word: "`CamelCase`" → "`Camel`", "`Case`"
  This can be disabled by setting `splitOnCaseChange="0"` (see below).
- A transition from alpha to numeric characters or vice versa:
  "`Gonzo5000`" → "`Gonzo`", "`5000`"
  "`4500XL`" → "`4500`", "`XL`"
  This can be disabled by setting `splitOnNumerics="0"`.

- Non-alphanumeric characters (discarded): "`hot-spot`" → "`hot`", "`spot`"
- A trailing "`'s`" is removed: "`O'Reilly's`" → "`O`", "`Reilly`"
- Any leading or trailing delimiters are discarded: "`--hot-spot--`" → "`hot`", "`spot`"

Factory class: `solr.WordDelimiterFilterFactory`

Arguments:

`generateWordParts`: (integer, default 1) If non-zero, splits words at delimiters. For example:

"`CamelCase`", "`hot-spot`" → "`Camel`", "`Case`", "`hot`", "`spot`"

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

"`1947-32`" →"`1947`", "`32`"

`splitOnCaseChange`: (integer, default 1) If 0, words are not split on camel-case changes:

"BugBlaster`-XL`" → "`BugBlaster`", "`XL`"

Example 1 below illustrates the default (non-zero) splitting behavior.

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

"FemBot`3000`" → "`Fem`", "`Bot3000`"

`catenateWords`: (integer, default 0) If non-zero, maximal runs of word parts will be joined:

"`hot-spot-sensor's`" → "`hotspotsensor`"

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

`1947-32`" → "`194732`"

`catenateAll`: (0/1, default 0) If non-zero, runs of word and number parts will be joined:

"`Zap-Master-9000`" → "`ZapMaster9000`"

`preserveOriginal`: (integer, default 0) If non-zero, the original token is preserved:

"`Zap-Master-9000`" → "`Zap-Master-9000`", "`Zap`", "`Master`", "`9000`"

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

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

Example:

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

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

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

T→F: "hot-spot", "RoboBlaster/9000", "100XL"

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

Example:

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

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

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

T→F: "hot-spot", "RoboBlaster/9000", "100-42"

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

Example:

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

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

In: "hot-spot 100+42 XL40"

T→F: "hot-spot"(1), "100+42"(2), "XL40"(3)

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

Example:

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

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

In: "XL-4000/ES"

T→F: "XL-4000/ES"(1)

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

Example:

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

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

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

T→F: "FooBar", "AstroBlaster", "XL-5000", "==ES-34-"

Out: "Foo", "Bar", "AstroBlaster", "XL-5000", "ES", "34"

5.7 CharFilterFactories

Char Filter is a component that pre-processes input characters. Char Filters can be chained like Token Filters and placed in front of a Tokenizer. **Char Filters** can add, change, or remove characters without worrying about fault of Token offsets.

5.7.1 solr.MappingCharFilterFactory

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

5.7.2 solr.HTMLStripCharFilterFactory

This filter creates `org.apache.solr.analysis.HTMLStripCharFilter`.

`HTMLStripCharFilter` strips HTML from the input stream and passes the result to either `CharFilter` or `Tokenizer`.

This filter:

- Removes HTML/XML tags while preserving other content.
- Removes attributes within tags and supports optional attribute quoting.
- Removes XML processing instructions, such as: <?foo bar?>
- Removes XML comments.
- Removes XML elements starting with <! and ending with >
- Removes contents of <script> and <style> elements.
  - Handles XML comments inside these elements (normal comment processing won't always work)
  - Replaces numeric character entities references like `&#65;` or `&#x7f;`
- The terminating ';' is optional if the entity reference is followed by whitespace.
- Replaces all named character entity references.
  - is replaced with a space instead of 0xa0.
  - The terminating ';' is mandatory to avoid false matches on something like "Alpha&Omega Corp"

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

The table below presents examples of HTML stripping.

Input	Output
my link	my link
<?xml?> hello<!--comment-->	hello
hello<script><-- f('<--internal--></script>'); --></script>	hello
if a<b then print a;	if a<b then print a;
hello <td height=22 nowrap align="left">	hello
a<b &#65 Alpha&Omega Ω	a<b A Alpha&Omega Ω

5.8 Language Analysis

This section contains information about tokenizers and filters related to character set conversion or for use with specific languages. For the European languages, tokenization is fairly straightforward. Tokens are delimited by whitespace and/or a relatively small set of punctuation characters. In other languages the tokenization rules are often not so simple. Some European languages may require special tokenization rules as well, such as rules for decompounding German words.

5.8.1 ISO Latin Accent Filter

This filter replaces any accented characters in a token with the unaccented equivalent. This can increase recall by causing more matches. On the other hand, it can reduce precision because language-specific character differences may be lost.

Characters in the ISO Latin 1 (ISO-8859-1) character set are recognized and letter case will be preserved, so that “Â” becomes “A” and “á” becomes “a”.

> **NOTE:** This filter only looks for accented characters, it does not filter out other non-ASCII characters.

Factory class: `solr.ISOLatin1AccentFilterFactory`

Arguments: None

Example:

```
<analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.ISOLatin1AccentFilterFactory"/>
</analyzer>
```

In: "Björn Ångström"

T→F: "Björn", "Ångström"

Out: "Bjorn", "Angstrom"

5.8.2 Brazilian

5.8.2.1 Brazilian Stem Filter

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

Factory class: `solr.BrazilianStemFilterFactory`

Arguments: None

Example:

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

In: "praia praias"

T→F: "praia", "praias"

Out: "pra", "pra"

5.8.3 Chinese

5.8.3.1 Chinese Tokenizer

This tokenizer breaks Chinese language text into tokens. Chinese is not a whitespace delimited language, so each Chinese character becomes a token. This filter is generally preferable for text that is known to be

all Chinese characters. The more generic CJKTokenizer, which recognizes Chinese, Japanese and Korean characters generates compound tokens, which results in larger indices and, possibly, less query precision.

The `StandardTokenizer` (Section 5.5.1) will also tokenize CJK characters as individual tokens.

Factory class: `solr.ChineseTokenizerFactory`

Arguments: None

Example:

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

In: "你好，我不讲中文"

Out: "你", "好", "我", "不", "讲", "中", "文"

5.8.3.2 Chinese Filter Factory

This is a stop-word filter that passes Chinese characters through. The current implementation uses a hard-coded table of English stop words. It does not currently stop any Chinese characters.

Depending on your application, it may be preferable to use the standard Stop Filter (Section 5.6.18), as it allows you to specify your own stop word list.

Factory class: `solr.ChineseFilterFactory`

Arguments: None

Example:

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

In: "你好, and 我不讲中文"

T→F: "你", "好", "and", "我", "不", "讲", "中", "文"

Out: "你", "好", "我", "不", "讲", "中", "文"

5.8.4 CJK

5.8.4.1 CJK Tokenizer

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

Factory class: `solr.CJKTokenizerFactory`

Arguments: None

Example:

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

In: "你好，我不讲中文"

Out: "你好", "我不", "不讲", "讲中", "讲", "中文", ""

5.8.5 Dutch

5.8.5.1 Dutch Stem Filter

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

Another option for stemming Dutch words is to use the Snowball Porter Stemmer with an argument of `language="Dutch"` (Section 5.6.16).

Factory class: `solr.DutchStemFilterFactory`

Arguments: None

Example:

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

In: "`kanaal kanalen`"

T→F: "`kanaal`", "`kanalen`"

Out: "`kanal`", "`kanal`"

5.8.6 French

5.8.6.1 Elision Filter

Removes article elisions from a token stream. This filter primarily applies to the French language and makes use of the `ElisionFilter` class in `org.apache.lucene.analysis.fr`.

Factory class: `solr.ElisionFilterFactory`

Arguments:

`articles`: (required) The pathname of a file that contains a list of articles, one per line, to be stripped. Articles are words such as "le", which are commonly abbreviated, such as *l'avion* (the plane). This file should include the abbreviated form, which precedes the apostrophe. In this case, simply "*l*".

Example:

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

In: "`L'histoire d'art`"

T→F: "`L'histoire`", "`d'art`"

Out: "`histoire`", "`art`"

5.8.6.2 French Stem Filter

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

Another option for stemming French words is to use the Snowball Porter Stemmer with an argument of `language="French"` (Section 5.6.16).

Factory class: `solr.FrenchStemFilterFactory`

Arguments: None

Example:

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

In: "le chat, les chats"

T→F: "le", "chat", "les", "chats"

Out: "le", "chat", "le", "chat"

5.8.7 German

5.8.7.1 German Stem Filter

This is a Java filter written specifically for stemming the German language. It uses the Lucene class `org.apache.lucene.analysis.de.GermanStemmer`.

Another option for stemming German words is to use the Snowball Porter Stemmer with an argument of `language="German"` *(Section 5.6.16)*.

Factory class: `solr.GermanStemFilterFactory`

Arguments: None

Example:

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

In: "hund hunden"

T→F: "hund", "hunden"

Out: "hund", "hund"

5.8.8 Dictionary Compound Word Token Filter

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

Compound words are most commonly found in Germanic languages.

Factory class: `solr.DictionaryCompoundWordTokenFilterFactory`

Arguments:

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

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

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

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

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

Example:

Assume that `germanwords.txt` contains at least the following words:

```
dumm
kopf
donau
dampf
schiff
```

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

In: "`Donaudampfschiff dummkopf`"

T→F: "`Donaudampfschiff`"(1), "`dummkopf`"(2),

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

5.8.9 Greek

5.8.9.1 Greek Lower Case Filter

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

Factory class: `solr.GreekLowerCaseFilterFactory`

Arguments:

`charset`: (optional, default "`UnicodeGreek`") Specifies the name of the character set to use. Must be "`UnicodeGreek`", "`ISO`" or "`CP1253`".

NOTE: Use of custom charsets is deprecated in Solr 1.4 and will be unsupported in Solr 1.5.

Example:

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

In: "ΕλληνικήΔημοκρατία Ellīnikī Dīmokratía"

T→F: "Ελληνική", "Δημοκρατία", "Ell□ik ", "Dīmokratía"

Out: "ελληνικη", "δημοκρατια", "ellīnikī", "dīmokratía"

5.8.10 Russian

5.8.10.1 Russian Letter Tokenizer

This tokenizer breaks Russian language text into tokens. It is similar to `LetterTokenizer`, but additionally looks up letters in the appropriate Russian character set.

Factory class: `solr.RussianLetterTokenizerFactory`

Arguments:

`charset`: (optional, default "`UnicodeRussian`") The name of the character set to use. Must be "`UnicodeRussian`", "`KOI8`" or "`CP1251`".

NOTE: Use of custom charsets is deprecated in Solr 1.4 and will be unsupported in Solr 1.5.

Example:

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

In: "Здравствулте!. Я не говорю русского."

Out: "Здравствулте", "Я", "не", "говорю", "русского"

5.8.10.2 Russian Lower Case Filter

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

Factory class: `solr.RussianLowerCaseFilterFactory`

Arguments:

`charset`: (optional, default "`UnicodeRussian`") Specifies the name of the character set to use.

Must be "`UnicodeRussian`", "`KOI8`" or "`CP1251`".

NOTE: Use of custom charsets is deprecated in Solr 1.4 and will be unsupported in Solr 1.5. If you need to index text in these encodings, please use Java's character set conversion facilities (InputStreamReader, etc.) during I/O, so that Lucene can analyze this text as Unicode instead.

Example:

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

In: "Здравствулте!. Я не говорю русского."

T→F: "Здравствулте", "Я", "не", "говорю", "русского"

Out: "здравствулте", "я", "не", "говорю", "русского"

5.8.10.3 Russian Stem Filter

This is a Java filter written specifically for stemming the Russian language. It uses the Lucene class `org.apache.lucene.analysis.ru.RussianStemmer`.

This filter only works with Russian lowercase letters. Tokens should first be passed through the Russian Lowercase Filter (above) for this filter to work reliably.

Another option for stemming Russian words is to use the Snowball Porter Stemmer with an argument of `language="Russian"` *(Section 5.6.16)*.

Factory class: `solr.RussianStemFilterFactory`

Arguments:

`charset`: (optional, default "`UnicodeRussian`") Specifies the name of the character set to use. Must be "`UnicodeRussian`", "`KOI8`" or "`CP1251`".

NOTE: Use of custom charsets is deprecated in Solr 1.4 and will be unsupported in Solr 1.5.

Example:

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

In: "вал валы"

T→F: "вал", "валы"

T→F: "вал", "валы"

Out: "вал", "вал"

5.8.11 Thai

5.8.11.1 Thai Word Filter

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

Factory class: `solr.ThaiWordFilterFactory`

Arguments: None

Example:

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

In: "ช้างสิบสามเชือก"

T→F: "ช้างสิบสามเชือก"

Out: "ช้าง", "สิบ", "สาม", "เชือก"

5.8.12 Arabic

Solr 1.4 introduces a package for Arabic analysis. The package includes:

- ArabicNormalizationFilter for Arabic orthographic normalization
- ArabicStemFilter for Arabic light stemming
- Arabic stop words file, which includes a set of default Arabic stop words.

For details, please consult Lucene documentation for issue LUCENE-1406.

5.9 Running Your Analyzer

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

For example, assume that the following field type definition has been added to `schema.xml`:

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

The objective here (during indexing) is to reconstruct hyphenated words (Section 5.6.4), which may have been split across lines in the text, then to set all words to lowercase (Section 5.6.8). For queries, you want to forego the de-hyphenation step.

To test this out, point your browser at the Field Analysis page of the Solr Admin Web interface. By default, this will be at the following URL (adjust the hostname and/or port to match your configuration): `http://localhost:8983/solr/admin/analysis.jsp`. You should see a page like Figure 1.

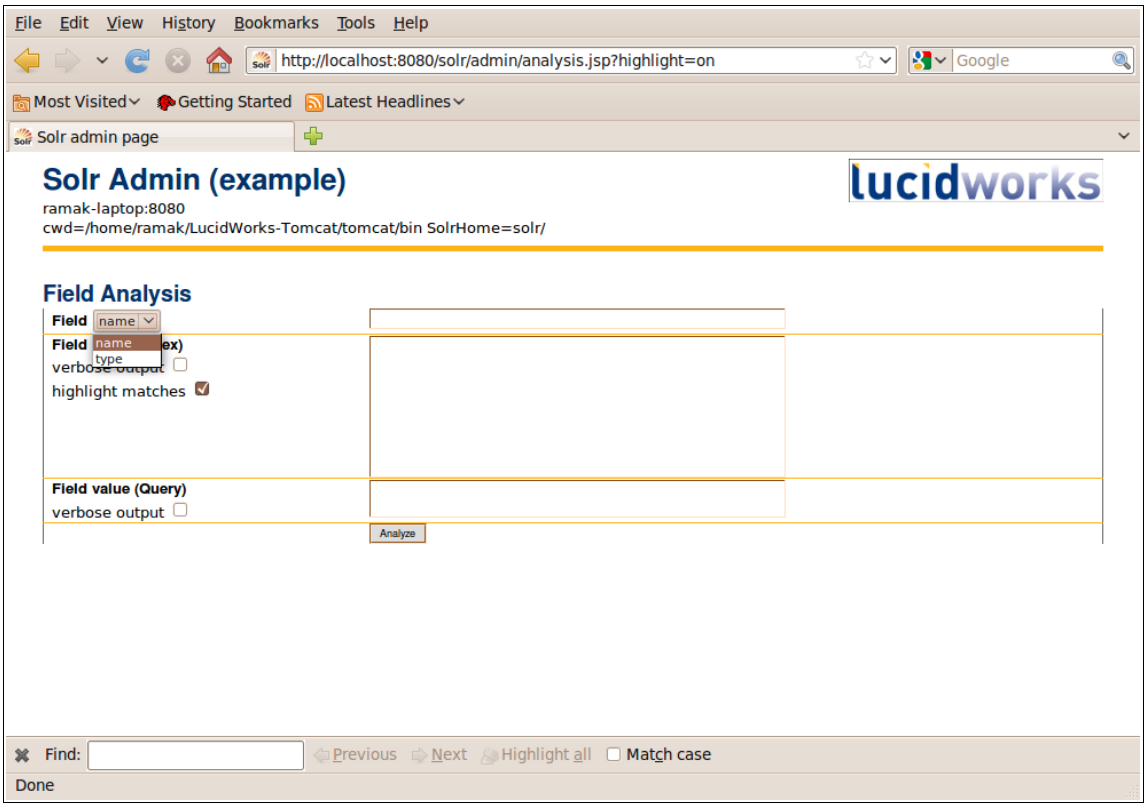


Figure 1: Empty Field Analysis screen

We want to test the field type definition for "`mytextfield`", defined above. The drop-down labeled "Field" has two values, "name" and "type". Choosing "type" allows you to give the value of the `name` attribute in a `<fieldType>` definition, "`mytextfield`" for this example.

You can also select "`name`" and provide the name of a `<field>` definition from `schema.xml`. A field definition refers to a type definition, so this is essentially an indirect way of the selecting the field's type.

In the "Field Value" box enter some sample text to be processed by the analyzer. The results of each analysis stage will be displayed when you click the "Analyze" button. Let's test the index analyzer by providing some sample text. We will leave the query field value empty for now. The result we expect is that `HyphenatedWordsFilter` will join the hyphenated pair "`Super-`" and "`computer`" into the single word "`Supercomputer`", and then `LowerCaseFilter` will set it to "`supercomputer`". Let's see what happens (Figure 2):

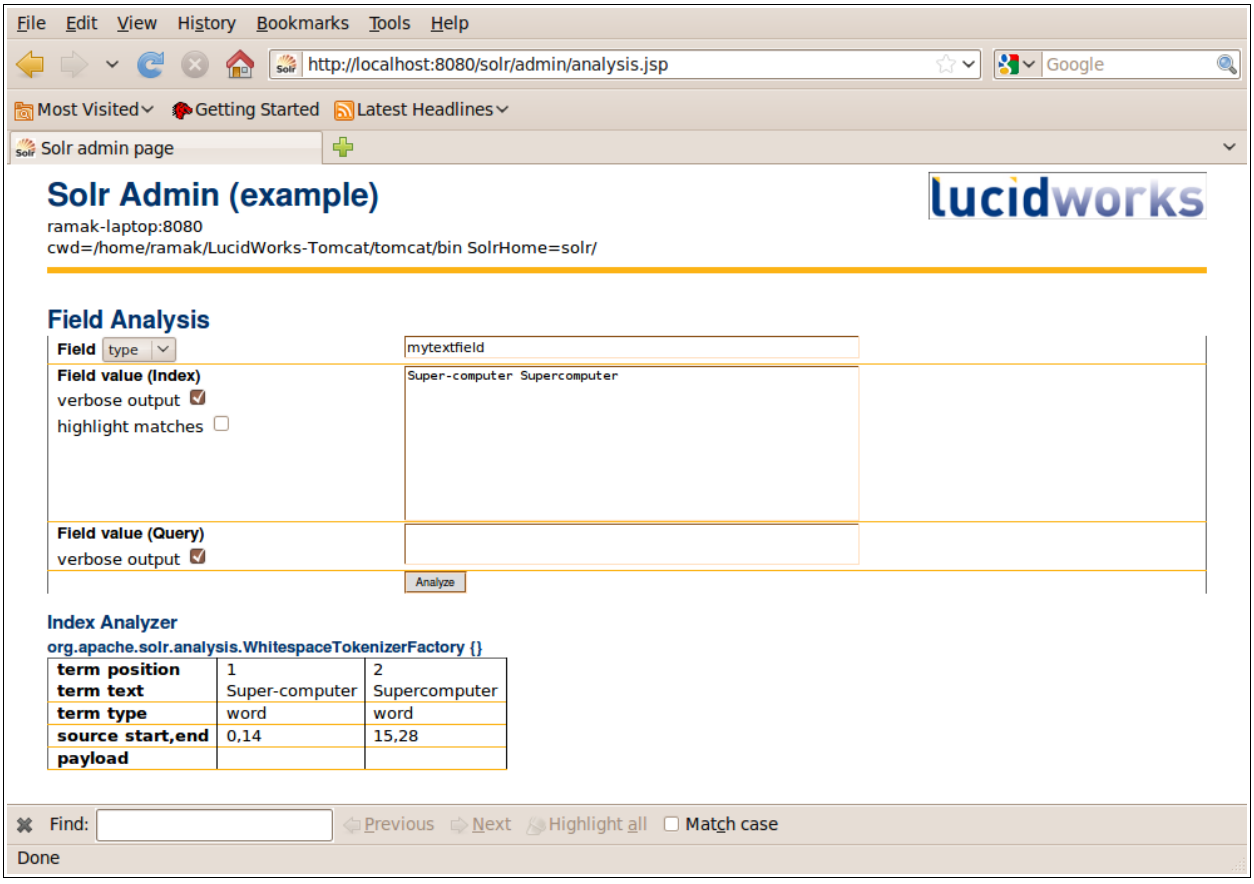


Figure 2: Running index-time analyzer, verbose output

The result is two distinct tokens rather than the one we expected. What went wrong? Looking at the first token that came out of `StandardTokenizer`, we can see the trailing hyphen has been stripped off of

"`Super-`". Checking the documentation for `StandardTokenizer` (Section 5.5.1), we see that it treats all punctuation characters as delimiters and discards them. What we really want in this case is a whitespace tokenizer that will preserve the hyphen character when it breaks the text into tokens.

Let's make this change (in bold) and try again:

```
<fieldType name="mytextfield" class="solr.TextField">
    <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.HyphenatedWordsFilterFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>
```

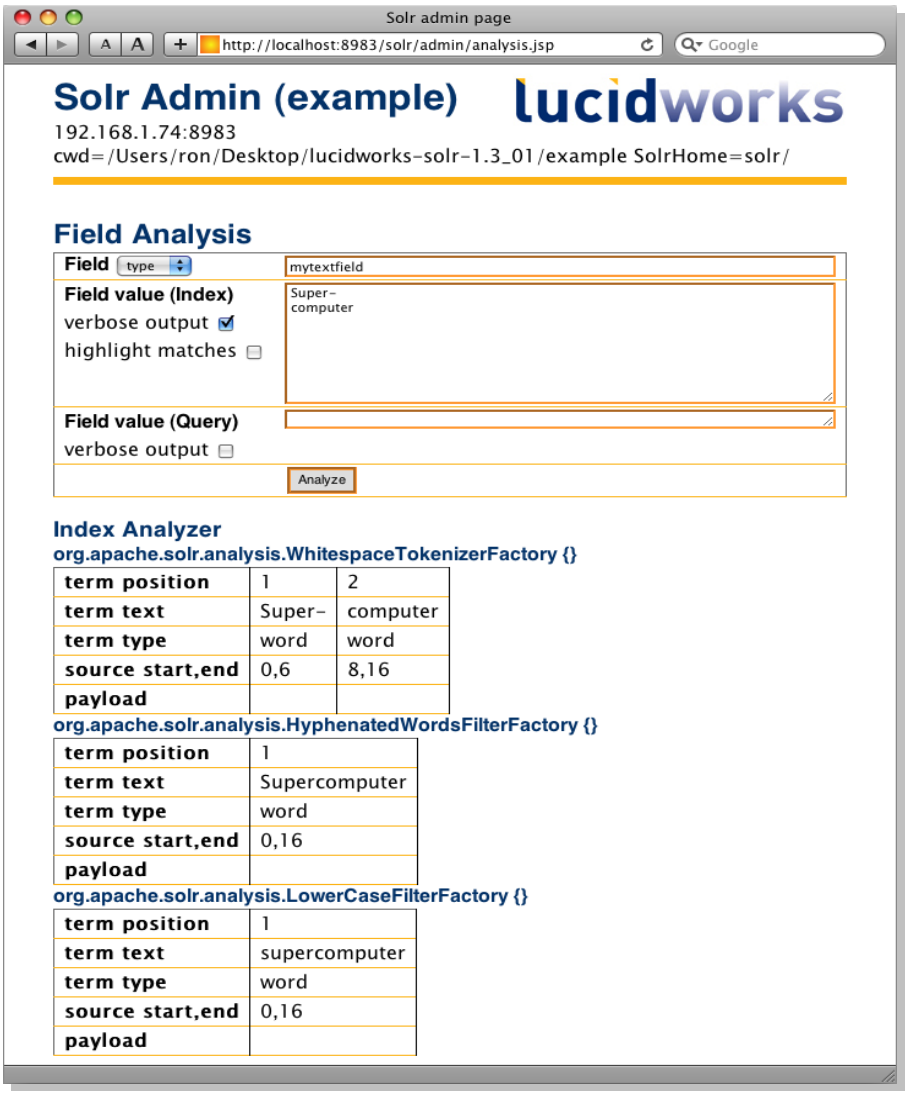


Figure 3: Using WhitespaceTokenizer, expected results

Re-submitting the form by clicking "Analyze" again, we see the result in Figure 3.

That's more like it. Because the whitespace tokenizer preserved the trailing hyphen on the first token, `HyphenatedWordsFilter` was able to reconstruct the hyphenated word, which then passed it on to `LowerCaseFilter`, where capital letters are set to lowercase.

Now let's see what happens when invoking the analyzer for query processing. For query terms, we don't want to do de-hyphenation and we *do* want to discard punctuation, so let's try the same input on it. We'll copy the same text to the "Field value (Query)" box and clear the one for index analysis. We'll also include the full, unhyphenated word as another term to make sure it is processed to lower case as we expect. Submitting again yields the results in Figure 4.

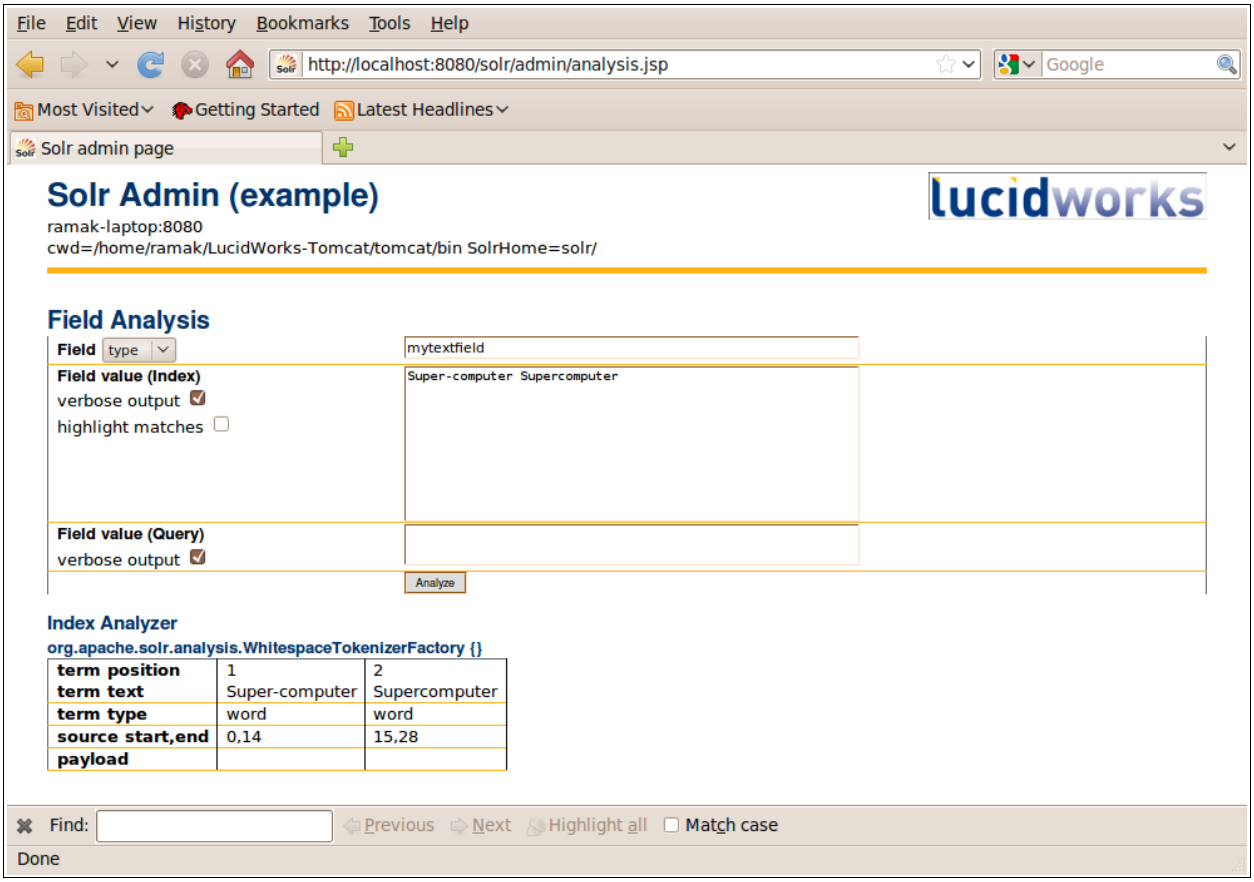


Figure 4: Query-time analyzer, good results

We can see that for queries the analyzer behaves the way we want it to. Punctuation is stripped out, `HyphenatedWordsFilter` doesn't run, and we wind up with the three tokens we expected.

Refer to the section titled "Running Field Analysis to Test Analyzers, Tokenizers, and TokenFilters" in Chapter 3 for more information about conducing field analysis through the Admin Web interface.

5.10 Summary

Solr uses field analyzers when processing documents to produce an index and when parsing a query to use to search an index. A field analyzer examines text fields and a generates a stream of lexical units called tokens. A field analyzer may be a single Lucene class, or it may comprise a tokenizer class and a series of token filter classes.

A tokenizer breaks field data into tokens.

A filter transforms a token stream in some way by changing, discarding, or creating tokens. A lower-case filter, for example, converts all upper-case letters in a token stream to lower-case letters.

In Chapter 4, we discussed defining fields and field types for your search application by use the `<field>` and `<fieldtype>` elements in `schema.xml`.

In Chapter 5, we showed you how to use other XML elements in `schema.xml` to specify the analyzers, tokenizers, and filters that will be associated with fields and field types. For any field or field type, you can specify which analyzers will be used for indexing, which for querying, and which for both. Using the Field Analysis form in the LucidWorks for Solr Web admin interface, you can test the results of your `schema.xml` configuration.

6 Indexing and Basic Data Operations

6.1 What Is Indexing?

This chapter describes the process of indexing: adding content to a Solr index and, if necessary, modifying that content or deleting it. By adding content to an index, we make it searchable by Solr.

A Solr index can accept data from many different sources, including XML files, comma-separated values (CSV) files, data extracted from tables in a database, and files in common file formats such as Microsoft Word or PDF.

Here are the three most common ways of loading data into a Solr index:

- Using the new "Solr Cell" framework built on Apache Tika for ingesting binary files or structured files such as Office, Word, PDF, and other proprietary formats.
- Uploading XML files by sending HTTP requests to the Solr server from any environment where such requests can be generated.
- Writing a custom Java application to ingest data through Solr's Java Client API (which is described in more detail in Chapter 11[4]). Using the Java API may be the best choice if you're working with an application, such as a Content Management System (CMS), that offers a Java API.

Regardless of the method used to ingest data, there is a common basic data structure for data being fed into a Solr index: a *document* containing multiple *fields,* each with a *name* and containing *content,* which

4 See also the JavaDocs for the SolrJ API: http://lucene.apache.org/solr/api/solrj/index.html

may be empty. One of the fields is usually designated as a unique ID field (analogous to a primary key in a database), although the use of a unique ID field is not strictly required by Solr.

If the field name is defined in the `schema.xml` file that is associated with the index, then the analysis steps associated with that field will be applied to its content when the content is tokenized. Fields that are not explicitly defined in the schema will either be ignored or mapped to a dynamic field definition (see Chapter 4), if one matching the field name exists.

6.1.1 The Solr 1.4 example Directory

The `example/` directory in the Solr 1.4 release includes a sample Solr implementation, along with sample documents for uploading into an index. You'll find the example docs in `<solr_home>/example/exampledocs`.

6.1.2 The curl Utility for Transferring Files

Many of the instructions and examples in this chapter make use of the `curl` utility for transferring content through a URL. `curl` posts and retrieves data over HTTP, FTP, and many other protocols. Most Linux distributions include a copy of `curl`. You'll find curl downloads for Linux, Windows, and many other operating systems at http://curl.haxx.se/download.html. Documentation for `curl` is available here: http://curl.haxx.se/docs/manpage.html.

> **NOTE:** Using `curl` or other command line tools for posting data is just fine for examples or tests, but it's not the recommended method for achieving the best performance for updates in production environments. You'll achieve better performance with Solr Cell or or the other methods described in this chapter.

Instead of curl, you can use utilities such as GNU `wget`[5] or manage GETs and POSTS with Perl, although the command line options will differ.

5 For more information about `wget`, see http://www.gnu.org/software/wget/.

6.2 Uploading Data with Solr Cell (using Apache Tika)

6.2.1 Introduction

Earlier releases of Solr could easily index data that was already in XML format, but indexing non-XML data, such as binary files or Office documents, required extra processing. Solr 1.4 uses code from the Apache Tika project[6] to provide a framework for incorporating many different file-format parsers such as Apache PDFBox[7] and Apache POI[8] into Solr itself. Working with this framework, Solr's ExtractingRequestHandler can use Tika to support uploading binary files—including files in popular formats such as Word and PDF—for data extraction and indexing.

When this framework was under development, it was called the Solr Content Extraction Library or CEL; from that abbreviation came this framework's name: Solr Cell.

6.2.2 Key Concepts

When using the Solr Cell framework, it's helpful to keep the following in mind:

- Tika will automatically attempt to determine the input document type (Word, PDF etc.) and extract the content appropriately. If you like, you can explicitly specify a MIME type for Tika with the `stream.type` parameter.
- Tika works by producing an XHTML stream that it feeds to a SAX[9] ContentHandler.
- Solr then responds to Tika's SAX events and creates the fields to index.
- Tika produces metadata such as Title, Subject, and Author according to specifications such as the DublinCore. See http://lucene.apache.org/tika/formats.html for the file types supported.
- Tika adds all the extracted text to the "content" field.
- You can map Tika's metadata fields to Solr fields. You can also boost these fields.
- You can pass in literals for field values.

6 Apache Tika is a toolkit for detecting and extracting metadata and structured text content from various documents using existing parser libraries. For more information, see http://lucene.apache.org/tika/.
7 Apache PDFBox is an open source Java PDF library for working with PDF documents. For more information, see http://incubator.apache.org/pdfbox/.
8 The POI project consists of APIs for manipulating various file formats based upon Microsoft's OLE 2 Compound Document format, and Office OpenXML format, using pure Java. For more information, see http://poi.apache.org/index.html.
9 SAX is a common interface implemented for many different XML parsers. For more information, see http://www.saxproject.org/quickstart.html.

- You can apply an XPath expression to the Tika XHTML to restrict the content that is produced.

6.2.3 Trying out Tika with the Solr Example Directory

You can try out the Tika framework using the `example` directory included in the Solr 1.4 release.

Start the Solr `example` server:

```
cd example
java -jar start.jar
```

In a separate window go to the `docs/` directory (which contains some nice example docs), or the `site` directory if you built Solr from source, and send Solr a file via HTTP POST:

```
cd docs
curl 'http://localhost:8983/solr/update/extract?
literal.id=doc1&commit=true' -F "myfile=@tutorial.html"
```

The URL above calls the Extraction Request Handler, uploads the file `tutorial.html` and assigns it the unique ID `doc1`. Here's a closer look at the components of this command:

- The `literal.id=doc1` parameter provides the necessary unique ID for the document being indexed.
- The `commit=true` parameter causes Solr to perform a commit after indexing the document, making it immediately searchable. For optimum performance when loading many documents, don't call the commit command until you are done.
- The `-F` flag instructs curl to POST data using the Content-Type multipart/form-data and supports the uploading of binary files. The `@` symbol instructs `curl` to upload the attached file.
- The argument `myfile=@tutorial.html` needs a valid path, which can be absolute or relative (e.g. "`myfile=@../../site/tutorial.html`" if you are still in `exampledocs` directory).

Now, you should be able to execute a query and find that document (open the following link in your browser): http://localhost:8983/solr/select?q=tutorial.

You may notice that although you can search on any of the text in the sample document, you may not be able to see that text when the document is retrieved. This is simply because the "content" field generated by Tika is mapped to the Solr field called `text`, which is indexed but not stored. This operation is controlled by default map rule in the `/update/extract` handler in `solrconfig.xml`, and its behavior can be easily changed or overridden. For example, to store and see all metadata and content, execute the following:

```
curl 'http://localhost:8983/solr/update/extract?
literal.id=doc1&uprefix=attr_&fmap.content=attr_content&commit=true' -F
"myfile=@tutorial.html"
```

In this command, the `uprefix=attr_` parameter causes all generated fields that aren't defined in the schema to be prefixed with `attr_` (which is a dynamic field that is stored).

The `fmap.content=attr_content` parameter overrides the default `fmap.content=text` causing the content to be added to the `attr_content` field instead.

Then run this command to query the document: http://localhost:8983/solr/select?q=attr_content:tutorial

6.2.4 Input Parameters

The table below describes the parameters accepted by the Extraction Request Handler..

Parameter	Description
`boost.<fieldname>=<float>`	Boosts the specified field. (Boosting a field alters its importance in a query response. To learn about boosting fields, see Chapter 7.)

Parameter	Description
`capture=<Tika_XHTML_name>`	Captures XHTML elements with the specified name for a supplementary addition to the Solr document. This parameter can be useful for copying chunks of the XHTML into a separate field. For instance, it could be used to grab paragraphs (<p>) and index them into a separate field. Note that content is still also captured into the overall "content" field.
`captureAttr=true|false`	Indexes attributes of the Tika XHTML elements into separate fields, named after the element. For example, when extracting from HTML, Tika can return the href attributes in <a> tags as fields named "a". See the examples below.
`defaultField=<field_name>`	If the `uprefix` parameter (see below) is not specified and a field cannot be determined, the default field will be used.
`extractOnly=true|false`	Default is false. If true, returns the extracted content from Tika without indexing the document. This literally includes the extracted XHTML as a string in the response. When viewing manually, it may be useful to use a response format other than XML to aid in viewing the embedded XHTML tags.[10]
`extractFormat=xml|text`	Default is xml. Controls the serialization format of the extract content. The xml format is actually XHTML, the same format that results from passing the `-x` command to the Tika command line application, while the text format is like that produced by Tika's `-t` command. This parameter is valid only if `extractOnly` is set to true.
`fmap.<source_field>=<target_field>`	Maps (moves) one field name to another. Example: `fmap.content=text` causes the content field generated by Tika to be moved to the "text" field.

10 For an example, see http://wiki.apache.org/solr/TikaExtractOnlyExampleOutput.

Parameter	Description
`literal.<fieldname>=<value>`	Creates a field with the specified value. The data can be multivalued if the field is multivalued.
`lowernames=true|false`	Maps all field names to lowercase with underscores. For example, "Content-Type" would be mapped to "content_type."
`resource.name=<file_name>`	Specifies the optional name of the file. Tika can use it as a hint for detecting a file's MIME type.
`uprefix=<prefix>`	Prefixes all fields that are not defined in the schema with the given prefix. This is very useful when combined with dynamic field definitions. Example: `uprefix=ignored_` would effectively ignore all unknown fields generated by Tika given the example schema contains `<dynamicField name="ignored_*" type="ignored"/>`
`xpath=<XPath_expression>`	When extracting, only return Tika XHTML content that satisfies the XPath expression. See http://lucene.apache.org/tika/documentation.html for details on the format of Tika XHTML. See also TikaExtractOnlyExampleOutput.

6.2.5 Order of Operations

Here is the order in which the Solr Cell framework, using the Extraction Request Handler and Tika, processes its input.

1. Tika generates fields or passes them in as literals specified by `literal.fieldname=value`
2. If `lowernames==true`, Tika maps fields to lowercase.
3. Tika applies the mapping rules specified by `fmap.source=target` parameters.

4. If `uprefix` is specified, any unknown field names are prefixed with that value, else if `defaultField` is specified, any unknown fields are copied to the default field.

6.2.6 Configuring the Solr ExtractingRequestHandler

If you are not working in the supplied `example/solr` directory, you must copy all libraries from `example/solr/libs` into a `libs` directory within your own `solr` directory or to a directory you've specified in solrconfig.xml using the new `libs` directive. The ExtractingRequestHandler is not incorporated into the Solr WAR file, so you have to install it separately.

Here's an example of configuring the ExtractingRequestHandler in `solrconfig.xml`.

```
<requestHandler name="/update/extract"
class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
    <lst name="defaults">
      <str name="fmap.Last-Modified">last_modified</str>
      <str name="uprefix">ignored_</str>
    </lst>
    <!--Optional.  Specify a path to a tika configuration file.  See the
Tika docs for details.-->
    <str name="tika.config">/my/path/to/tika.config</str>
    <!-- Optional. Specify one or more date formats to parse.  See
DateUtil.DEFAULT_DATE_FORMATS for default date formats -->
    <lst name="date.formats">
      <str>yyyy-MM-dd</str>
    </lst>
  </requestHandler>
```

In the defaults section, we are mapping Tika's Last-Modified Metadata attribute to a field named `last_modified`. We are also telling it to ignore undeclared fields. These are all overridden parameters.

The `tika.config` entry points to a file containing a Tika configuration. You would only need this entry if you have customized your Tika configuration. The Tika configuration file contains information about parsers, MIME types, etc.

You may also need to adjust the `multipartUploadLimitInKB` attribute as follows if you are submitting very large documents.

```
  <requestDispatcher handleSelect="true" >
    <requestParsers enableRemoteStreaming="false"
multipartUploadLimitInKB="20480" />
    ....
```

Lastly, the `date.formats` allows you to specify various `java.text.SimpleDateFormats` date formats for working with transforming extracted input to a Date. Solr comes configured with the following date formats (see the DateUtil in Solr):

```
yyyy-MM-dd'T'HH:mm:ss'Z'
yyyy-MM-dd'T'HH:mm:ss
yyyy-MM-dd
yyyy-MM-dd hh:mm:ss
yyyy-MM-dd HH:mm:ss
EEE MMM d hh:mm:ss z yyyy
EEE, dd MMM yyyy HH:mm:ss zzz
EEEE, dd-MMM-yy HH:mm:ss zzz
EEE MMM d HH:mm:ss yyyy
```

6.2.6.1 MultiCore Configuration

For a multi-core configuration, specify `sharedLib='lib'` in `<solr />` in `example/solr/solr.xml` in order for Solr to find the JAR files in `example/solr/lib`.

For more information about Solr cores, see Chapter 8.

6.2.7 Metadata

As mentioned before, Tika produces metadata about the document. Metadata describes different aspects of a document, such as the author's name, the number of pages, the file size, and so on. The metadata produced depends on the type of document submitted. For instance, PDFs have different metadata than Word documents do.

In addition to Tika's metadata, Solr adds the following metadata (defined in ExtractingMetadataConstants):

Solr Metadata	Description
`stream_name`	The name of the ContentStream as uploaded to Solr. Depending

Solr Metadata	Description
	on how the file is uploaded, this may or may not be set
stream_source_info	Any source info about the stream. (See the section on Content Streams later in this chapter.)
stream_size	The size of the stream in bytes.
stream_content_type	The content type of the stream, if available.

NOTE: We recommend that you try using the `extractOnly` option to discover which values Solr is setting for these metadata elements.

6.2.8 Examples of Uploads Using the Extraction Request Handler

6.2.8.1 Capture and Mapping

The command below captures `<div>` tags separately, and then maps all the instances of that field to a dynamic field named `foo_t`.

```
curl "http://localhost:8983/solr/update/extract?
literal.id=doc2&captureAttr=true&defaultField=text&fmap.div=foo_t&capture=
div"  -F "tutorial=@tutorial.pdf"
```

6.2.8.2 Capture, Mapping, and Boosting

The command below captures `<div>` tags separately, maps the field to a dynamic field named `foo_t`, then boosts `foo_t` by 3.

```
curl "http://localhost:8983/solr/update/extract?
literal.id=doc3&captureAttr=true&defaultField=text&capture=div&fmap.div=fo
o_t&boost.foo_t=3" -F "tutorial=@tutorial.pdf"
```

6.2.8.3 Using Literals to Define Your Own Metadata

To add in your own metadata, pass in the `literal` parameter along with the file:

```
curl "http://localhost:8983/solr/update/extract?
literal.id=doc4&captureAttr=true&defaultField=text&capture=div&fmap.div=fo
o_t&boost.foo_t=3&literal.blah_s=Bah"  -F "tutorial=@tutorial.pdf"
```

6.2.8.4 XPath

The example below passes in an XPath expression to restrict the XHTML returned by Tika:

```
curl "http://localhost:8983/solr/update/extract?
literal.id=doc5&captureAttr=true&defaultField=text&capture=div&fmap.div=fo
o_t&boost.foo_t=3&literal.id=id&xpath=/xhtml:html/xhtml:body/xhtml:div/des
cendant:node()"  -F "tutorial=@tutorial.pdf"
```

6.2.8.5 Extracting Data without Indexing It

Solr allows you to extract data without indexing. You might want to do this if you're using Solr soley as an extraction server or if you're interested in testing Solr extraction.

The example below sets the `extractOnly=true` parameter to extract data without indexing it.

```
curl "http://localhost:8983/solr/update/extract?&extractOnly=true"
--data-binary @tutorial.html  -H 'Content-type:text/html'
```

The output includes XML generated by Tika (and further escaped by Solr's XML) using a different output format to make it more readable:

```
curl "http://localhost:8983/solr/update/extract?
&extractOnly=true&wt=ruby&indent=true"  --data-binary @tutorial.html  -H
'Content-type:text/html'
```

6.2.9 Sending Documents to Solr with a POST

The example below streams the file as the body of the POST, which does not, then, provide information to Solr about the name of the file.

```
curl "http://localhost:8983/solr/update/extract?
literal.id=doc5&defaultField=text" --data-binary @tutorial.html -H
'Content-type:text/html'
```

6.2.10 Sending Documents to Solr with Solr Cell and SolrJ

SolrJ is a Java client that you can use to add documents to the index, update the index, or query the index. (You'll find more information on SolrJ in Chapter 11.)

Here's an example of using Solr Cell and SolrJ to add documents to a Solr index.

First, let's use SolrJ to create a new SolrServer, then we'll construct a request containing a ContentStream (essentially a wrapper around a file) and sent it to Solr:

```
public class SolrCellRequestDemo {
  public static void main(String[] args) throws IOException,
SolrServerException {
    SolrServer server = new
CommonsHttpSolrServer("http://localhost:8983/solr");
    ContentStreamUpdateRequest req = new
ContentStreamUpdateRequest("/update/extract");
    req.addFile(new File("apache-solr/site/features.pdf"));
    req.setParam(ExtractingParams.EXTRACT_ONLY, "true");
    NamedList&lt;Object&gt; result = server.request(req);
    System.out.println("Result: " + result);
}
```

This operation streams the file `features.pdf` into the Solr index.

The sample code above calls the `extract` command, but you can easily substitute other commands that are supported by Solr Cell. The key class to use is the `ContentStreamUpdateRequest`, which makes sure the ContentStreams are set properly. SolrJ takes care of the rest.

Note that the `ContentStreamUpdateRequest` is not just specific to Solr Cell. You can send CSV to the CSV Update handler and to any other Request Handler that works with Content Streams for updates.

6.3 Uploading Data with Index Handlers

The example URLS given here reflect the handler configuration in the supplied `solrconfig.xml`. If the name associated with the handler is changed then the URLs will need to be modified. It is quite possible to access the same handler using more than one name, which can be useful if you wish to specify different sets of default options.

6.3.1 Using the XMLUpdateRequestHandler for XML-formatted Data

Until Solr 1.4, this was the standard way to upload data into Solr.

6.3.1.1 Configuration

The default configuration file has the update request handler configured by default.

```
<requestHandler name="/update" class="solr.XmlUpdateRequestHandler" />
```

6.3.1.2 Adding Documents

Documents are added to the index by sending an XML message to the update handler.

The XML schema recognized by the update handler is very straightforward:

- The `<add>` element introduces one more more documents to be added.
- The `<doc>` element introduces the fields making up a document.
- The `<field>` element presents the content for a specific field.

For example:

```
<add>
<doc>
   <field name="authors">Patrick Eagar</field>
   <field name="subject">Sports</field>
   <field name="dd">796.35</field>
   <field name="numpages">128</field>
   <field name="desc"></field>
   <field name="price">12.40</field>
   <field name="title" boost=”2.0”>Summer of the all-rounder: Test and
championship cricket in England 1982</field>
   <field name="isbn">0002166313</field>
```

```
   <field name="yearpub">1982</field>
   <field name="publisher">Collins</field>
</doc>
<doc boost="2.5">
...
</doc>
</add>
```

If the document schema defines a unique key, then an `/update` operation silently replaces a document in the index with the same unique key, unless the `<add>` element sets the `"allowDups"` attribute to `"true"`. If no unique key has been defined, indexing performance is somewhat faster, as no search has to be made for an existing document.

Each element has certain optional attributes which may be specified.

Command	Command Description	Optional Parameter	Parameter Description
`<add>`	Introduces one or more documents to be added to the index.	`commitWithin=`*`number`*	Add the document within the specified number of milliseconds
		`overwrite=true|false`	Default is "true". If true, newer documents will replace older ones with the ame uniqueKey.
`<doc>`	Introduces the definition of a specific document.	`boost=`*`float`*	Default is 1.0. Sets a boost value for the document.[11]
`<field>`	Defines a field within a document.	`boost=`*`float`*	Default is 1.0. Sets a boost value for the field.

NOTE: Other optional parameters for `<add>`, including `allowDups`, `overwritePending`, and `overwriteCommitted`, are now deprecated.

11 To learn more about boosting, see Chapter 7.

6.3.1.3 Commit and Optimize Operations

The <commit> operation writes all documents loaded since the last commit to one or more segment files on the disk. Before a commit has been issued, newly indexed content is not visible to searches. The commit operation opens a new searcher, and triggers any event listeners that have been configured.

Commits may be issued explicitly with a <commit/> message, and can also be triggered from <autocommit> parameters in solrconfig.xml.

The <optimize> operation requests Solr to merge internal data structures in order to improve search performance. For a large index, optimization will take some time to complete, but by merging many small segment files into a larger one, search performance will improve. If you are using Solr's replication mechanism (see Chapter 10) to distribute searches across many systems, be aware that after an optimize, a complete index will need to be transferred. In contrast, post-commit transfers are usually much smaller.

The <commit> and <optimize> elements accept these optional attributes:

Optional Attribute	Description
maxSegments	Default is 1. Optimizes the index to include no more than this number of segments.
waitFlush	Default is true. Blocks until index changes are flushed to disk.
waitSearcher	Default is true. Blocks until a new searcher is opened and registered as the main query searcher, making the changes visible.
expungeDeletes	Default is false. Merges segments and removes deleted documents.

Here are examples of <commit> and <optimize> using optional attributes:

```
<commit waitFlush="false" waitSearcher="false"/>
<commit waitFlush="false" waitSearcher="false" expungeDeletes="true"/>
<optimize waitFlush="false" waitSearcher="false"/>
```

6.3.1.4 Delete Operations

Documents can be deleted from the index in two ways. "Delete by ID" deletes the document with the specified ID, and can be used only if a `UniqueID` field has been defined in the schema. "Delete by Query" deletes all documents matching a specified query. A single delete message can contain multiple delete operations.

```
<delete>
      <id>0002166313</id>
      <id>0031745983</id>
      <query>subject:sport</query>
      <query>publisher:penguin</query>
</delete>
```

6.3.1.5 Rollback Operations

NOTE: This feature is new in Solr 1.4.

The rollback command rollbacks all add/deletes made to the index since the last commit. It neither calls any event listeners nor creates a new searcher. It's syntax is simple:

```
<rollback/>
```

6.3.1.6 Using curl to Perform Updates with the Update Request Handler.

You can use the `curl` utility[12] to perform any of the above commands, using its '`--data-binary`' option to append the XML message to the `curl` command, and generating a HTTP POST request, for example:

```
curl http://localhost:8983/update -H "Content-Type: text/xml" --data-binary '<add>
<doc> <field name="authors">Patrick Eagar</field> <field
name="subject">Sports</field> <field name="dd">796.35</field>
 <field name="isbn">0002166313</field><field name="yearpub">1982</field>
   <field name="publisher">Collins</field></doc> </add>'
```

For posting XML messages contained in a file, you can use the alternative form:

```
curl http://localhost:8983/update -H "Content-Type: text/xml" --data-
binary @myfile.xml
```

12 For an overview of curl, see page 148.

Short requests can also be sent using a HTTP GET command, URL-encoding the request, as in the following. Note the escaping of "<" and ">":

```
curl http://localhost:8983/update?stream.body=%3Ccommit/%3E
```

Responses from Solr take the form shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int name="QTime">127</int>
</lst>
</response>
```

The status field will be non-zero in case of failure. The servlet container will generate an appropriate HTML-formatted message in the case of an error at the HTTP layer.

6.3.1.7 A Simple, Cross-Platform Posting Tool

For demo purposes, the file `example/exampledocs/post.jar` includes a cross-platform Java tool for POST-ing XML documents. Open a window and run.

```
java -jar post.jar <list of files with  messages>
```

By default, this will contact the server at `localhost:8983`. The "`-help`" option outputs the following information on its usage:

```
SimplePostTool: version 1.2
This is a simple command line tool for POSTing raw XML to a Solr port.  XML data
can be read from files specified as command line args; as raw commandline arg
strings; or via STDIN.

Examples:
      java -Ddata=files -jar post.jar *.xml
      java -Ddata=args  -jar post.jar '<delete><id>42</id></delete>'
      java -Ddata=stdin -jar post.jar < hd.xml
Other options controlled by System Properties include the Solr URL to POST to, and
whether a commit should be executed.  These are the defaults for all System
Properties.
      -Ddata=files
      -Durl=http://localhost:8983/solr/update
      -Dcommit=yes
```

6.3.2 Using the CSVRequestHandler for CSV Content

6.3.2.1 Configuration

The default configuration file has the update request handler configured by default, although the "lazy load" flag is set.

```
<requestHandler name="/update/csv" class="solr.CSVRequestHandler"
   startup="lazy" />
```

6.3.2.2 Parameters

The CSV handler allows the specification of many parameters in the URL in the form:

```
f.parameter.optional_fieldname=value
```

The table below describes the parameters for the update handler.

Parameter	Usage	Global/ per field	Example
`separator`	Character used as field separator; default is ","	`G,(f: see split)`	`separator=%`
`trim`	If true, remove leading and trailing whitespace from values. Default=false.	`g,f`	`f.isbn.trim=true trim=false`
`header`	Set to true if first line of input contains field names. These will be used if the **fieldnames** parameter is absent.	`g`	
`fieldnames`	Comma separated list of field names to use when adding documents.	`g`	`fieldnames=isbn,price,title`
`skip`	Comma separated list of field names to skip	`g`	`skip=uninteresting,shoesize`
`skipLines`	Number of lines to discard in the input stream before the CSV data starts, including the header, if present. Default=0.	`g`	`skipLines=5`

Parameter	Usage	Global/ per field	Example
`encapsulator`	The character optionally used to surround values to preserve characters such as the CSV separator or whitespace. This standard CSV format handles the encapsulator itself appearing in an encapsulated value by doubling the encapsulator.	`g,(f:` see **`split`**`)`	`encapsulator="`
`escape`	The character used for escaping CSV separators or other reserved characters. If an escape is specified, the encapsulator is not used unless also explicitly specified since most formats use either encapsulation or escaping, not both	`g`	`escape=\`
`keepEmpty`	Keep and index zero length (empty) fields. Default=false.	`g,f`	`f.price.keepEmpty=true`
`map`	Map one value to another. Format is value:replacement (which can be empty.)	`g,f`	`map=left:right` `f.subject.map=history:bunk`
`split`	If true, split a field into multiple values by a separate parser.	`f`	
`overwrite`	If true (the default), check for and overwrite duplicate documents, based on the uniqueKey field declared in the Solr schema. If you know the documents you are indexing do not contain any duplicates then you may see a considerable speed up setting this to false.	`g`	
`commit`	Issues a commit after the data has been ingested	`g`	

6.3.3 Indexing Using SolrJ

Use of the the SolrJ client library is covered in Chapter 11.

6.4 Uploading Structure Data Store Data with the Data Import Handler

6.4.1 Overview

Many search applications store the content to be indexed in a structured data store, such as a relational database. The Data Import Handler (DIH) provides a mechanism for importing content from a data store and indexing it. In addition to relational databases, DIH can index content from HTTP based data sources such as RSS and ATOM feeds, e-mail repositories, and structured XML where an XPath processor is used to generate fields.

6.4.2 Concepts and Terminology

Descriptions of the Data Import Handler use several familiar terms, such as entity and processor, in specific ways, as explained in the table below..

Term	Definition
Datasource	As its name suggests, a datasource defines the location of the data of interest. For a database, it's a DSN. For an HTTP datasource, it's the base URL.
Entity	Conceptually, an entity is processed to generate a set of documents, containing multiple fields, which (after optionally being transformed in various ways) are sent to Solr for indexing. For a RDBMS data source, an entity is a view or table, which would be processed by one or more SQL statements to generate a set of rows (documents) with one or more columns (fields).
Processor	An entity processor does the work of extracting content from a data source, transforming it, and adding it to the index. Custom entity processors can be written to extend or replace the ones supplied.
Transformer	Each set of fields fetched by the entity may optionally be transformed. This process can modify the fields, create new fields, or generate multiple rows/documents form a single row. There are several built-in transformers in the DIH, which perform functions such as modifying dates and stripping HTML. It is possible to write custom transformers using the publicly

Term	Definition
	available interface.

6.4.3 Configuration

The Data Import Handler has to be registered in `solrconfig.xml`. For example:

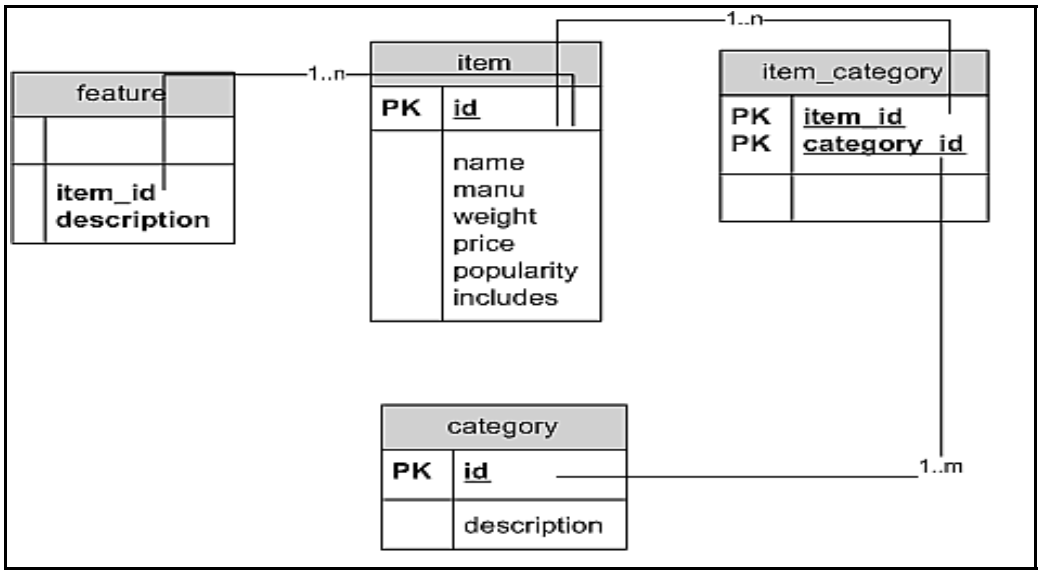
```
<requestHandler name="/dataimport"
class="org.apache.solr.handler.dataimport.DataImportHandler">
    <lst name="defaults">
      <str name="config">/path/to/my/DIHconfigfile.xml</str>
    </lst>
  </requestHandler>
```

You can have multiple DIH configuration files. Each file would require a separate definition in the `solrconfig.xml` file, specifying a path to the file.

The DIH configuration file contains specifications for the data source, how to fetch data, what data to fetch, and how to process it to generate the Solr documents to be posted to the index.

There is a sample DIH application distributed with LucidWorks in the directory `example/example-DIH`. This accesses a small hsqldb database. Details of how to run this example can be found in the `README.txt` file. Its DIH configuration can be found in the file `example/example-DIH/solr/db/conf/db-data-config.xml`.

An annotated configuration file, based on the sample, is shown below. It extracts fields from the four tables defining a simple product database, with this schema.




```
<dataConfig>
<!-- The first element is the dataSource, in this case an HSQLDB database. The
path to the JDBC driver and the JDBC URL and login credentials are all specified
here. Other permissible attributes include whether or not to autocommit to
Solr,the batchsize used in the JDBC connection, a 'readOnly' flag

    <dataSource driver="org.hsqldb.jdbcDriver" url="jdbc:hsqldb:./example-
DIH/hsqldb/ex" user="sa" password="mypass" />

<!-- a 'document' element follows, containing multiple 'entity' elements. Note
that 'entity' elements can be nested, and this allows the entity relationships in
the sample database to be mirrored here, so that we can generate a denormalized
Solr record which may include multiple features for one item, for instance -->

The possible attributes for the entity element are describe below.
Entity elements may contain one or more 'field' elements, which map the datasource
field names to Solr fields, and optionally specify per-field transformations -->

    <document name="products">
<!-- this entity is the 'root' entity. -->
      <entity name="item" pk="ID"
         query="select * from item"
         deltaQuery="select id from item where last_modified > '$
{dataimporter.last_index_time}'">
```

```
        <field column="ID" name="id" />

<!-- multiple Solr fields are generated from a single column in the table -->
        <field column="NAME" name="name" />
        <field column="NAME" name="nameSort" />
        <field column="NAME" name="alphaNameSort" />

<!-- This entity is nested and reflects the one-to-many relationship between an
item and its multiple features. Note the use of variables; ${item.ID} is the value
of the column 'ID' for the current item ('item' referring to the entity name)  -->

      <entity name="feature" pk="ITEM_ID"
          query="select DESCRIPTION from FEATURE where ITEM_ID='${item.ID}'"
          deltaQuery="select ITEM_ID from FEATURE where last_modified > '$
{dataimporter.last_index_time}'"
          parentDeltaQuery="select ID from item where ID=${feature.ITEM_ID}">
          <field name="features" column="DESCRIPTION" />
       </entity>
<!--  -->
      <entity name="item_category" pk="ITEM_ID, CATEGORY_ID"
         query="select CATEGORY_ID from item_category where ITEM_ID='${item.ID}'"
         deltaQuery="select ITEM_ID,CATEGORY_ID from item_category where
last_modified > '${dataimporter.last_index_time}'"
         parentDeltaQuery="select ID from item where ID=${item_category.ITEM_ID}">

          <entity name="category" pk="ID"
             query="select DESCRIPTION from category where ID = '$
{item_category.CATEGORY_ID}'"
             deltaQuery="select ID from category where last_modified > '$
{dataimporter.last_index_time}'"
             parentDeltaQuery="select ITEM_ID, CATEGORY_ID from item_category
where CATEGORY_ID=${category.ID}">
             field column="description" name="cat" />

          </entity>
       </entity>
    </entity>
</document>
```

6.4.4 Data Import Handler Commands

DIH commands are sent to Solr via an HTTP request. The following operations are supported.

Command	Description
`abort`	Aborts an ongoing operation . The URL is `http://<host>:<port>/solr/dataimport?command=abort` .
`delta-import`	For incremental imports and change detection. The command is of the form `http://<host>:<port>/solr/dataimport?command=delta-import`. It supports the same clean, commit, optimize and debug parameters as full-import command.
`full-import`	A Full Import operation can be started with a URL of the form `http://<host>:<port>/solr/dataimport?command=full-import` The command returns immediately. The operation will be started in a new thread and the *status* attribute in the response should be shown as *busy*. The operation may take some time depending on the size of dataset. Queries to Solr are not blocked during full-imports. When a `full-import` command is executed, it stores the start time of the operation in a file located at *conf/dataimport.properties* This stored timestamp is used when a delta-import operation is executed. For a list of parameters that can be passed to this command, see below.
`reload-config`	If the configuration file has been changed and you wish to reload it without restarting Solr. run the command `http://<host>:<port>/solr/dataimport?command=reload-config` .
`status`	The URL is `http://<host>:<port>/solr/dataimport?command=status` .It returns statistics on the number of documents created, deleted, queries run, rows fetched, status etc.

6.4.4.1 Parameters for the full-import Command

The `full-import` command accepts the following parameters:

Parameter	Description
`clean`	Default is true. Tells whether to clean up the index before the indexing is started.
`commit`	Default is true. Tells whether to commit after the operation.
`debug`	Default is false Runs the command in debug mode. It is used by the interactive development mode. Note that in debug mode, documents are never committed automatically. If you want to run debug mode and commit the results too, add "`commit=true`" as a request parameter.
`entity`	The name of an entity directly under the `<document>` tag in the configuration file. Use this to execute one or more entities selectively. Multiple "entity" parameters can be passed on to run multiple entities at once. If nothing is passed, all entities are executed.
`optimize`	Default is true. Tells Solr whether to optimize after the operation.

6.4.5 Data Sources

A dataSource specifies the origin of data and its type. Somewhat confusingly, some dataSources are configured within the associated entity processor. DataSources can also be specified in `solrconfig.xml`, which is useful when you have multiple environments (for example, development, QA, and production) differing only in their data sources.

You can create a custom dataSource by writing a class that extends `org.apache.solr.handler.dataimport.DataSource`

The mandatory attributes for a dataSource definition are its name and type. The name identifies the dataSource to an Entity element.

The types of dataSources available are described below.

6.4.5.1 ContentStreamDataSource

This the POST data as the data source. This can be used with any EntityProcessor that uses a DataSource<Reader>.

6.4.5.2 FieldReaderDataSource

This can be used where a database field contains XML which you wish to process using the XpathEntityProcessor. You would set up a configuration with both JDBC and FieldReader datasources, and two entities, as follows

```
 <dataSource name = "a1" driver="org.hsqldb.jdbcDriver" …   />
  <dataSource name="a2" type=FieldReaderDataSource" />
  <!-- processor for database →
  <entity name ="e1" dataSource="a1" processor="SQLEntityProcessor"  pk="docid"
       query="select * from t1 …..">
  <!-- nested XpathEntity; the field in the parent which is to be used for Xpath
is set in the 'datafield attribute inplace of the "url" attribute →
       <entity name="e2" dataSource="a2" processor="XPathEntityProcessor"
           dataField="e1.fieldToUseForXPath"
           <!-- Xpath configuration follows →
          …
      </entity>
  </entity>
```

6.4.5.3 FileDataSource

This can be used like an URL!DataSource, but is used to fetch content from files on disk. The only difference from URL!DataSource, when accessing disk files, is how a pathname is specified. The signature is as follows

```
public class FileDataSource extends DataSource<Reader>
```

This dataSource accepts these optional attributes.

Optional Attribute	Description
`basePath`	The base path relative to which the value is evaluated if it is not absolute.
`encoding`	If the files are to be read in an encoding that is not same as the platform encoding.

6.4.5.4 HTTPDataSource

NOTE: As of Solr 1.4, HTTPDataSource is deprecated in favor of URLDataSource, which is described below.

This dataSource can be used to fetch content from a URL (`http://` or `file://`). It's typically used with the `XpathEntityProcessor.` As the example below shows, this dataSource accepts the same attributes as the URLDataSource.

```
<!-- baseURL, encoding, connectionTimeout(ms) and readTimeout(ms) are all
optional -->
<dataSource name="x" type="HTTPDataSource" baseUrl="http://host:port/"
encoding="UTF-8" connectionTimeout="5000" readTimeout="10000"/>
```

6.4.5.5 JdbcDataSource

This is the default dataSource. It's used with the SQLEntityProcessor. See the example above for details on configuration.

6.4.5.6 URLDataSource

NOTE: As of Solr 1.4, we recommend that you use this dataSource rather than HTTPDataSource.

This dataSource is often used with X!PathEntityProcessor to fetch content from an underlying file:// or http:// location. The signature is as follows

```
public class URLDataSource extends DataSource<Reader>
```

Here's an example:

```
<dataSource name="a" type="URLDataSource" baseUrl="http://host:port/"
encoding="UTF-8" connectionTimeout="5000" readTimeout="10000"/>
```

The URLDataSource type accepts these optional parameters:

Optional Parameter	Description
`baseURL`	Specifies a new baseURL for pathnames. You can use this to specify host/port changes between Dev/QA/Prod environments. Using this attribute isolates the changes to be made to the `solrconfig.xml`
`connectionTimeout`	Specifies the length of time in milliseconds after which the connection should time out. The default value is 5000ms.
`encoding`	By default the encoding in the response header is used. You can use this property to override the default encoding.
`readTimeout`	Specifies the length of time in milliseconds after which a read operation should time out. The default value is 10000ms.

6.4.6 Entity Processors

Entity processors extract data, transform it, and add it to a Solr index. Examples of entities or data sources include views or tables in a data store.

Each processor has its own set of attributes, described in its own section below. In addition, there are non-specific attributes common to all entities which may be specified.

Attribute	Use
`datasource`	The name of a dataSource. Used if there are multiple datasources, specified, in which case each one must have a name.
`name`	Required. The unique name used to identify an entity.

Attribute	Use
pk	The primary key for the entity. It is optional, and required only when using delta-imports. It has no relation to the uniqueKey defined in schema.xml but they can both be the same. It is mandatory if you do delta-imports and then refers to the column name in ${dataimporter.delta.<column-name>} which is used as the primary key.
processor	Default is SQLEntityProcessor. Required only if the datasource is not RDBMS.
onError	Permissible values are (abort|skip|continue) . The default value is 'abort' .'skip' skips the current document. 'continue' ignores the error and processing continues.
preImportDeleteQuery	Before a full-import command, use this query this to cleanup the index instead of using '*:*'. This is honored only on an entity that is an immediate sub-child of <document>.
postImportDeleteQuery	Similar to the above, but executed after the import has completed.
rootEntity	By default the entities immediately under the <document> are root entities. If this attribute is set to false , the entity directly falling under that entity will be treated as the root entity (and so on). For every row returned by the root entity, a document is created in Solr.
transformer	Optional. One or more transformers to be applied on this entity.

6.4.6.1 The SQL Entity Processor

The SqlEntityProcessor is the default processor. The associated data source should be a JDBC URL.

The entity attributes specific to this processor are shown in the table below.

Attribute	Use
query	Required. The SQL query used to select rows.

Attribute	Use
deltaQuery	SQL query used if the operation is delta-import. This query selects the primary keys of the rows which will be parts of the delta-update. The pks will be available to the deltaImportQuery through the variable ${dataimporter.delta.<column-name>}.
parentDeltaQuery	SQL query used if the operation is delta-import.
deletedPkQuery	SQL query used if the operation is delta-import.
deltaImportQuery	SQL query used if the operation is delta-import. If this is not present , DIH tries to construct the import query by(after identifying the delta) modifying the 'query' (this is error prone). There is a namespace ${dataimporter.delta.<column-name>} which can be used in this query. e.g: select * from tbl where id=${dataimporter.delta.id}.

6.4.6.2 The XPathEntityProcessor

This processor is used when indexing XML formatted data. The data source is typically URLDataSource or FileDataSource. Xpath can also be used with the FileListEntityProcessor described below, to generate a document from each file.

The entity attributes unique to this processor are shown below.

Attribute	Use
Processor	Required. Must be set to “XpathEntityProcessor”.
url	Required. HTTP URL or file location.
stream	Optional: Set to true for a large file or download.
forEach	Required unless you define useSolrAddSchema. The Xpath expression which demarcates each record. This will be used to set up the processing loop.
xsl	Optional: Its value (a URL or filesystem path) is the name of a resource used as a preprocessor for applying the XSL transformation.
useSolrAddSchema	Set this to true if the content is in the form of the standard Solr update

	XML schema.
`flatten`	Optional: If set true, then text from under all the tags is extracted into one field.

Each field element in the entity can have the following attributes as well as the default ones.

Attribute	Use
`xpath`	Required. The XPath expression which will extract the content from the record for this field. Only a subset of Xpath syntax is supported.
`commonField`	Optional. If true, then when this field is encountered in a record it will be copied to future records when creating a Solr document.

Example:

```
<!-- slashdot RSS Feed --->
<dataConfig>
  <dataSource type="HttpDataSource" />
    <document>
      <entity name="slashdot"
        pk="link"
        url="http://rss.slashdot.org/Slashdot/slashdot"
        processor="XPathEntityProcessor"
 <!-- forEach sets up a processing loop ; here there are two expressions-->
        forEach="/RDF/channel | /RDF/item" transformer="DateFormatTransformer">

        <field column="source" xpath="/RDF/channel/title" commonField="true" />
        <field column="source-link" xpath="/RDF/channel/link" commonField="true"/>
        <field column="subject" xpath="/RDF/channel/subject" commonField="true" />
        <field column="title" xpath="/RDF/item/title" />
        <field column="link" xpath="/RDF/item/link" />
        <field column="description" xpath="/RDF/item/description" />
        <field column="creator" xpath="/RDF/item/creator" />
        <field column="item-subject" xpath="/RDF/item/subject" />
        <field column="date" xpath="/RDF/item/date"
            dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss" />
        <field column="slash-department" xpath="/RDF/item/department" />
        <field column="slash-section" xpath="/RDF/item/section" />
        <field column="slash-comments" xpath="/RDF/item/comments" />
      </entity>
  </document>
</dataConfig>
```

6.4.6.3 The FileList EntityProcessor

This processor is basically a wrapper, and is designed to generate a set of files satisfying conditions specified in the attributes which can then be passed to another processor, such as the XpathEntityProcessor. The entity information for this processor would be nested within the FileListEnitity entry. It generates four implicit fields : `fileAbsolutePath, fileSize, fileLastModified, fileName` which can be used in the nested processor. This processor does not use a datasource.

The attributes specific to this processor are described in the table below:

Attribute	Use
`fileName`	Required. A regular expression pattern to identify files to be included.
`basedir`	Required. The base directory (absolute path.)
`recursive`	Whether to search directories recursively. Default is 'false'.
`excludes`	A regular expression pattern to identify files which will be excluded.
`newerThan`	A date in the format `yyyy-MM-dd HH:mm:ss` or a date math expression `('NOW - 2YEARS')`.
`olderThan`	A date, using the same formats as newerThan.
`rootEntity`	This should be set to false. This ensures that each row (filepath) emitted by this processor is considered to be a document.
`dataSource`	Must be set to null.

The example below shows the combination of the FileListEntityProcessor with another processor which will generate a set of fields from each file found.

```
<dataConfig>
<dataSource type="FileDataSource"/>
<document>
   <!-- this outer processor generates a list of files satisfying the conditions
        specified in the attributes -->
  <entity name="f" processor="FileListEntityProcessor"
```

```
      fileName=".*xml"  newerThan="'NOW-30DAYS'"
      recursive="true"  rootEntity="false"
      dataSource="null" baseDir="/my/document/directory">
  <!-- this processor extracts content using Xpath from each file found -->
    <entity name="nested" processor="XPathEntityProcessor"
       forEach="/rootelement" url="${f.fileAbsolutePath}" >
       <field column="name" xpath="/rootelement/name"/>
       <field column="number" xpath="/rootelement/number"/>
    </entity>
  </entity>
</document>
</dataConfig>
```

6.4.6.4 LineEntityProcessor

New in Solr 1.4, this EntityProcessor reads all content from the data source on a line by line basis, a field called ;rawLine is returned for each line read. The content is not parsed in any way; however, you may add transformers to manipulate the data within the `rawLine` field, or to create other additional fields.

The lines read can be filtered by two regular expressions specified with the `acceptLineRegex` and `omitLineRegex` attributes. The table below describes the LineEntityProcessor's attributes:

Attribute	Description
`url`	A required attribute that specifies the location of the input file in a way that is compatible with the configured datasource. If this value is relative and you are using FileDataSource or URL! DataSource, it assumed to be relative to baseLoc.
`acceptLineRegex`	An optional attribute that if present discards any line which does not match the regExp.
`omitLineRegex`	An optional attribute that is applied after any acceptLineRegex and that discards any line which matches this regExp.

For example:

```
<entity name="jc"
        processor="LineEntityProcessor"
```

```
        acceptLineRegex="^.*\.xml$"
        omitLineRegex="/obsolete"
        url="file:///Volumes/ts/files.lis"
        rootEntity="false"
        dataSource="myURIreader1"
        transformer="RegexTransformer,DateFormatTransformer"
        >
   ...
```

While there are use cases where you might need to create a Solr document for each line read from a file, it is expected that in most cases that the lines read by this processor will consist of a pathname, which in turn will be consumed by another EntityProcessor, such as X!PathEntityProcessor.

6.4.6.5 PlainTextEntityProcessor

New in Solr 1.4, this EntityProcessor reads all content from the data source into an single implicit field called `plainText`. The content is not parsed in any way, however you may add transformers to manipulate the data within the `plainText` as needed, or to create other additional fields.

For example:

```
<entity processor="PlainTextEntityProcessor" name="x"
url="http://abc.com/a.txt" dataSource="data-source-name">
   <!-- copies the text to a field called 'text' in Solr-->
  <field column="plainText" name="text"/>
</entity>
```

Ensure that the `dataSource` is of type DataSource<Reader> (FileDataSource, URL!DataSource)

6.4.7 Transformers

Transformers manipulate the fields in a document returned by an entity. A transformer can create new fields or modify existing ones. You must tell the entity which transformers your import operation will be using, by adding an attribute containing a comma separated list to the <entity> element.

```
<entity name="abcde" transformer="org.apache.solr....,my.own.transformer,..." />
```

Specific transformation rules are then added to the attributes of a `<field>` element, as shown in the examples below. The transformers are applied in the order in which they are specified in the `transformer` attribute.

The Data Import Handler contains several built-in transformers. You can also write your own custom transformers, as described in the Solr Wiki (see http://wiki.apache.org/solr/DIHCustomTransformer). The ScriptTransformer (described below) offers an alternative method for writing your own transformers.

Solr 1.4 includes the following built-in transformers:

Transformer Name	Use
`ClobTransformer`	Used to create a String out of a Clob type in database .
`DateFormatTransformer`	Parse date/time instances.
`HTMLStripTransformer`	Strip HTML from a field.
`LogTransformer`	Used to log data to log files or a console.
`NumberFormatTransformer`	Uses the NumberFormat class in java to parse a string into a number.
`RegexTransformer`	Use regular expressions to manipulate fields.
`ScriptTransformer`	Write transformers in Javascript or any other scripting language supported by Java. Requires Java 6.
`TemplateTransformer`	Transform a field using a template.

These transformers are described below.

6.4.7.1 ClobTransformer

You can use the ClobTransformer, which is new in Solr 1.4, to create a string out of a CLOB in a database. A CLOB is a character large object: a collection of character data typically stored in a separate location that is referenced in the database.[13] Here's an example of invoking the ClobTransformer.

```
<entity name="e" transformer="ClobTransformer" ..>
<field column="hugeTextField" clob="true" />
...
</entity>
```

The ClobTransformer accepts these attributes:

13 See http://en.wikipedia.org/wiki/Character_large_object.

Attribute	Description
clob	Boolean value to signal if ClobTransformer should process this field or not. If this attribute is omitted, then the corresponding field is not transformed.
sourceColName	The source column to be used as input. If this is absent source and target are same

6.4.7.2 The DateFormatTransformer

This transformer converts dates from one format to another. This would be useful, for example, in a situation where you wanted to convert a field with a fully specified date/time into a less precise date format, for use in faceting.

DateFormatTransformer applies only on the fields with an attribute "dateTimeFormat". Other fields are not modified.

This transformer recognizes the following attributes:

Attribute	Description
dateTimeFormat	The format used for parsing this field. This must comply with the syntax of the Java SimpleDateFormat class.
sourceColName	The column on which the dateFormat is to be applied. If this is absent source and target are same.

Here's example code which would return the date rounded up to the month "2007-JUL":

```
<entity name="en" pk="id" transformer="DateTimeTransformer" …>
  …
   <field column="date" sourceColName="fulldate" dateTimeFormat="yyyy-MMM"/>
</entity>
```

6.4.7.3 The LogTransformer

You can use this transformer, which is new in Solr 1.4, to log data to the console or log files. For example:

```
<entity ...
transformer="LogTransformer"
logTemplate="The name is ${e.name}" logLevel="debug" >
....
</entity>
```

Unlike other transformers, the LogTransformer does not apply to any field, so the attributes are applied on the entity itself.

6.4.7.4 The NumberTransformer

Use this transformer to parse a number from a string, converting it into the specified format, and optionally using a different locale.

NumberFormatTransformer will be applied only to fields with an attribute "`formatStyle`".

This transformer recognizes the following attributes:

Attribute	Description
`formatStyle`	The format used for parsing this field. The value of the attribute must be one of (number|percent|integer|currency). This uses the semantics of the Java NumberFormat class.
`sourceColName`	The column on which the NumberFormat is to be applied. This is attribute is absent. The source column and the target column are the same.
`locale`	The locale to be used for parsing the strings. If this is absent, the system's default locale is used. It must be specified as language-country. For example, `en-US`.

For example:

```
<entity name="en" pk="id" transformer="NumberFormatTransformer" …>
  …
```

```
<!-- treat this field as UK pounds -->
   <field name="price_uk" column="price" formatStyle="currency" locale="en-UK" />
</entity>
```

6.4.7.5 The RegexTransformer

The `regex` transformer helps in extracting or manipulating values from fields (from the source) using Regular Expressions. The actual class name is `org.apache.solr.handler.dataimport.RegexTransformer`. But as it belongs to the default package the package-name can be omitted.

The table below describes the attributes recognized by the `regex` transformer.

Attribute	Description
`regex`	The regular expression that is used to match against the column or `sourceColName's` value(s). If `replaceWith` is absent, each regex *group* is taken as a value and a list of values is returned.
`sourceColName`	The column on which the regex is to be applied. If not present, then the source and target are identical.
`splitBy`	Used to split a string. It returns a list of values.
`groupNames`	A comma separated list of field column names, used where the `regex` contains groups and each group is to be saved to a different field. If some groups are not to be named leave a space between commas.
`replaceWith`	Used along with `regex` . It is equivalent to the method `new String(<sourceColVal>).replaceAll(<regex>, <replaceWith>).`

Here's an example of configuring the `regex` transformer:

```
<entity name="foo" transformer="RegexTransformer"
```

```
query="select full_name , emailids from foo"/>
... />
   <field column="full_name"/>
   <field column="firstName" regex="Mr(\w*)\b.*"
sourceColName="full_name"/>
   <field column="lastName" regex="Mr.*?\b(\w*)"
sourceColName="full_name"/>

   <!-- another way of doing the same -->
   <field column="fullName" regex="Mr(\w*)\b(.*)"
groupNames="firstName,lastName"/>
   <field column="mailId" splitBy="," sourceColName="emailids"/>
</entity>
```

In this example, `regex` and `sourceColName` are custom attributes used by the transformer. The transformer reads the field `full_name` from the resultset and transforms it to two new target fields, `firstName` and `lastName`. Even though the query returned only one column, `full_name`, in the resultset, the Solr document gets two extra fields `firstName` and `lastName` which are "derived" fields. These new fields are only created if the `regexp` matches.

The `emailids` field in the table can be a comma-separated value. It ends up producing one or more email IDs ,and we expect the `mailId` to be a multivalued field in Solr.

Note that this transformer can either be used to split a string into tokens based on a `splitBy` pattern, or to perform a string substitution as per `replaceWith`, or it can assign groups within a pattern to a list of `groupNames`. It decides what it is to do based upon the above attributes `splitBy`, `replaceWith` and `groupNames` which are looked for in order. This first one found is acted upon and other unrelated attributes are ignored.

6.4.7.6 The ScriptTransformer

The script transformer allows arbitrary transformer functions to be written in any scripting language supported by Java, such as Javascript, JRuby, Jython, Groovy, or BeanShell. Javascript is integrated into Java 6; You'll need to integrate other languages yourself.

Each function you write must accept a row variable (which corresponds to a Java `Map<String,Object>`, thus permitting `get,put,remove` operations). Thus you can modify the value of an existing field or add new fields. The return value of the function is the returned object.

The script is inserted into the DIH configuration file file at the top level and is called once for each row.

Here is a simple example.

```
<dataconfig>
<!-- simple script to generate a new row, converting a temperature from
Fahrenheit to Centigrade -->
  <script><![CDATA[
     function f2c(row) {
       var tempf, tempc;
       tempf = row.get('temp_f');
       if (tempf != null) {
         tempc = (tempf - 32.0)*5.0/9.0
          row.put('temp_c', temp_c);
         }
       return row;
     }
   ]]></script>
   <document>
 <!-- the function is specified as an entity attribute -->
     <entity name="e1" pk="id" transformer="script:f2c" query="select *
from X">
     ....
     </entity>
   </document>
</dataConfig>
```

6.4.7.7 The TemplateTransformer

You can use the template transformer to construct or modify a field value, perhaps using the value of other fields. You can insert extra text into the template.

```
<entity name="en" pk="id" transformer="TemplateTransformer" …>
  …
<!-- generate a full address from fields containing the component parts -->
   <field column="full_address" template="$en.{street},$en{city},$en{zip}" />
</entity>
```

6.4.8 Special Commands for the Data Import Handler

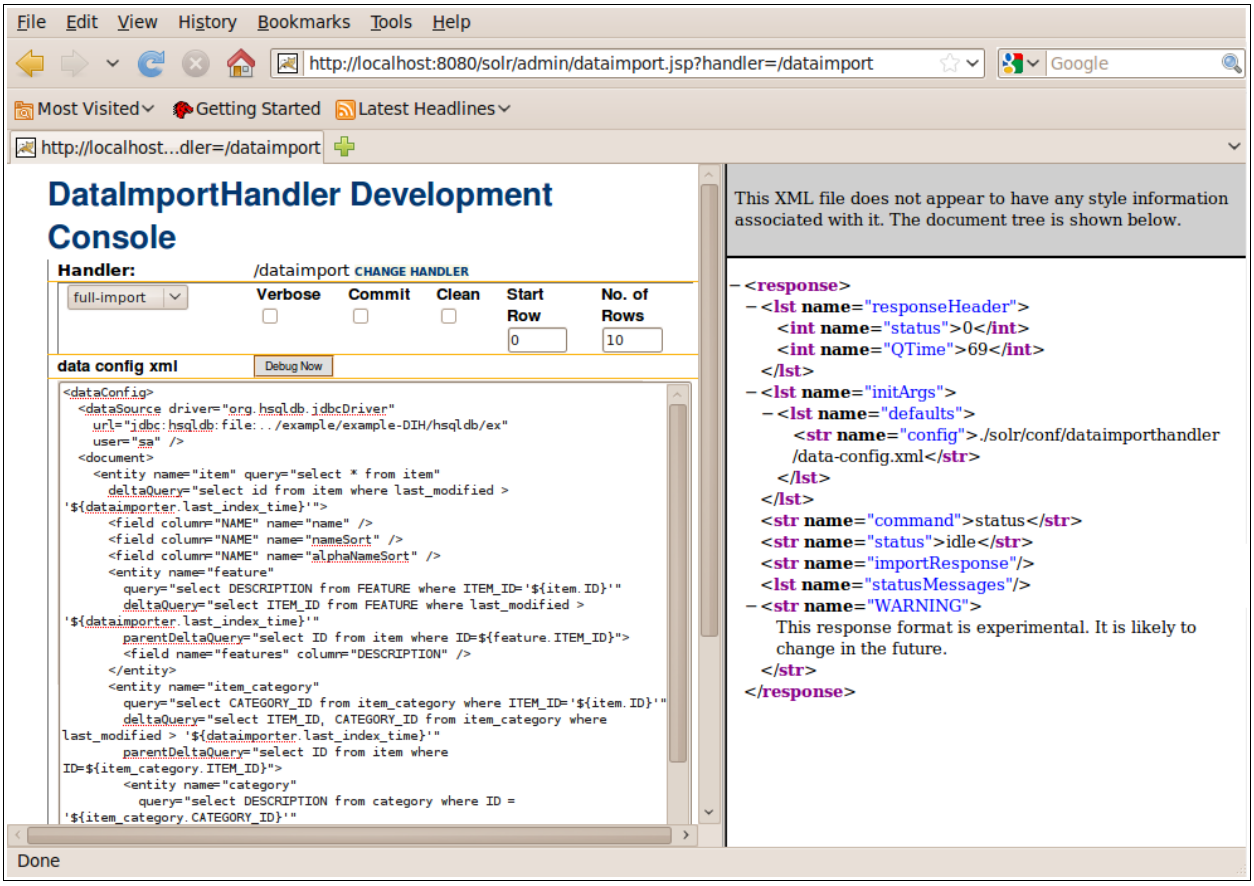
You can pass special commands to the DIH by adding any of the variables listed below to any row returned by any component:

Variable	Description
`$skipDoc`	Skip the current document; i.e., do not add it to Solr. The value can be the String `true|false`.
`$skipRow`	Skip the current row. The document will be added with rows from other entities. The value can be the String `true|false`
`$docBoost`	Boost the current document. The boost value can be a number or the toString conversion of a number.
`$deleteDocById`	Delete a document from Solr with this ID. The value has to be the uniqueKey value of the document.
`$deleteDocByQuery`	Delete documents from Solr using this query. The value must be a Solr Query.

6.4.9 The Data Import Handler Development Console

The Data Import Handler includes a browser-based console to help with development. You can access the console at this address: `http://host:port/solr/admin/dataimport.jsp`.

The screenshot below shows the DIH Development Console.

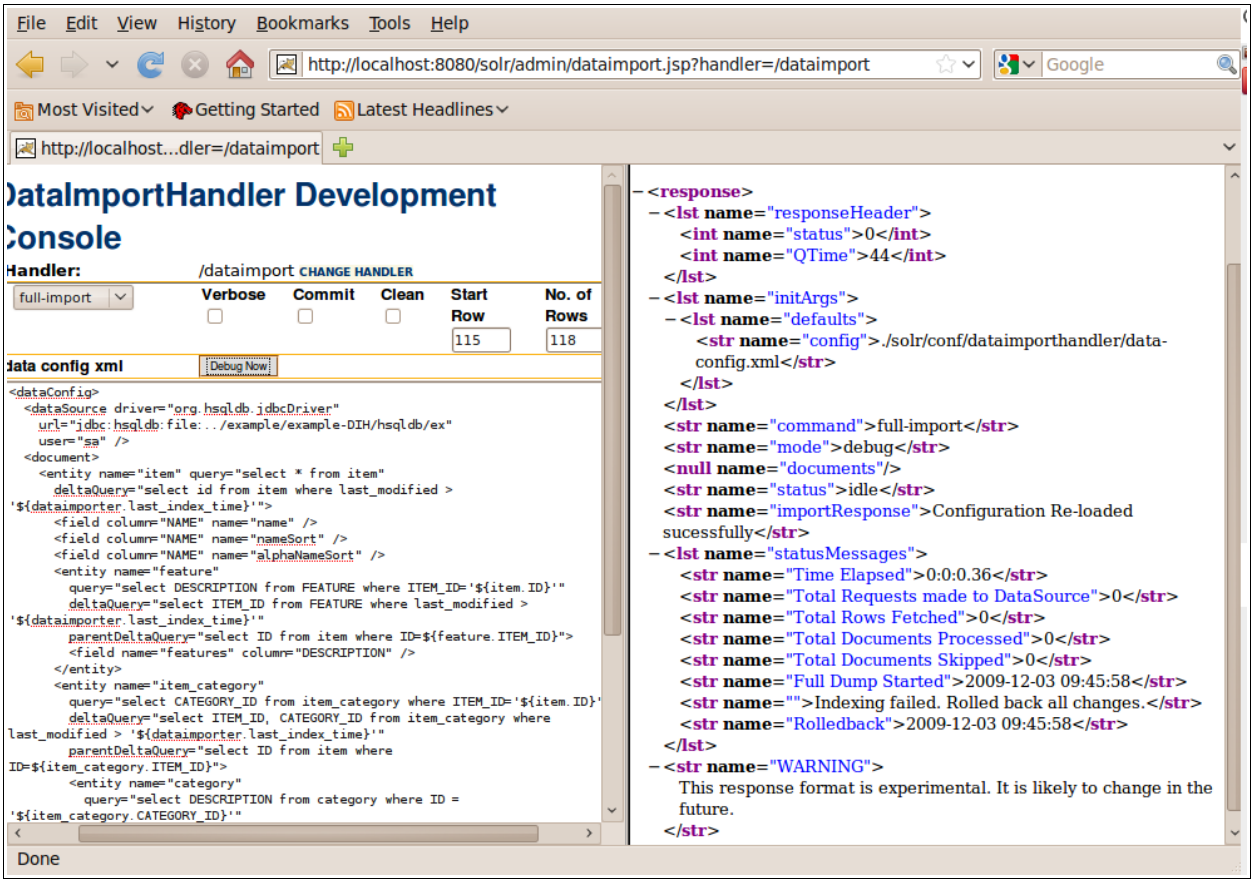


The Data Import Handler Console.

The console features two panels: the left-hand panel holds input (a `dataconfig.xml` file in the `conf/` directory), and the right-hand panel shows output.

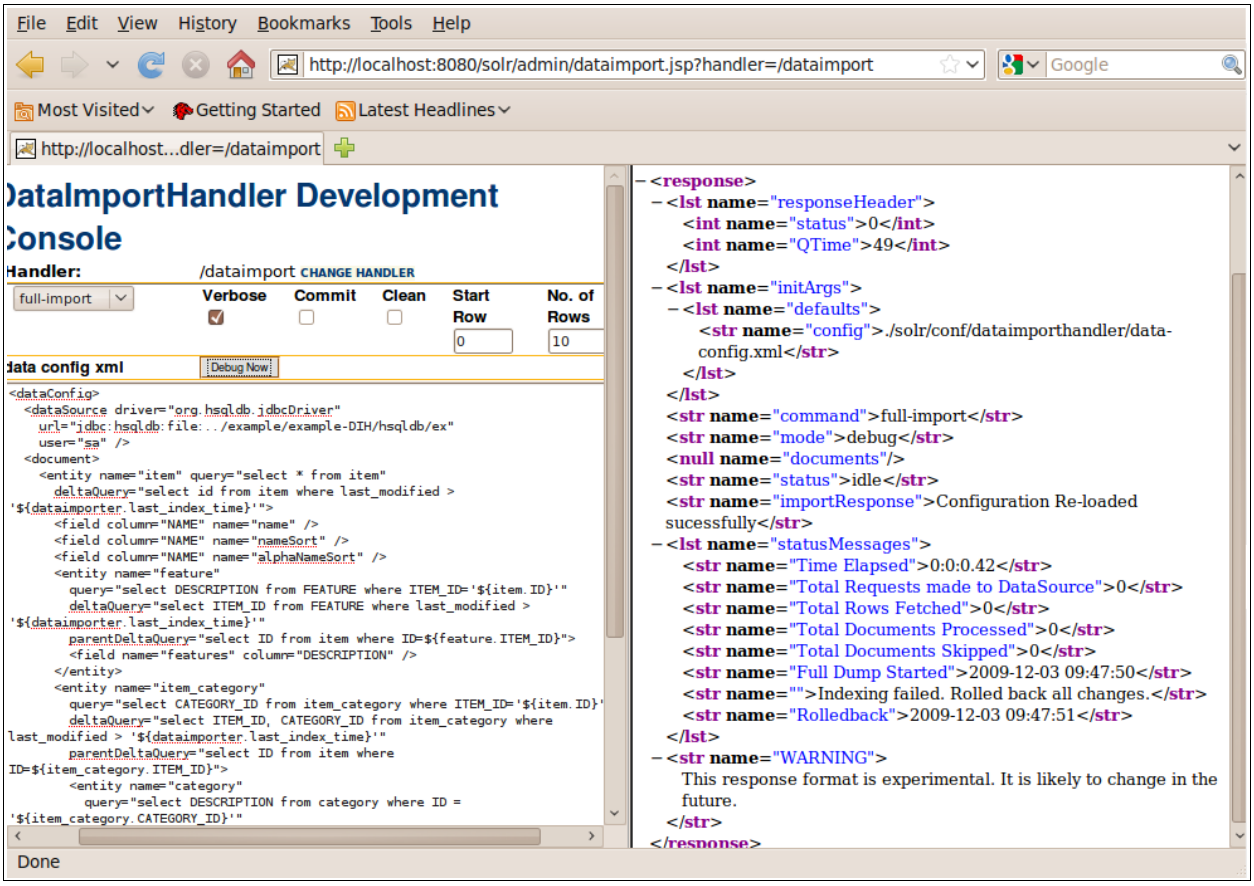
When you click the *Debug Now* button, the console runs the configuration and shows the documents created.

You can configure the start and rows parameters to debug a specific range of documents—for example, documents 115 to 118, as shown in the figure below.



Limiting Output to a Specific Set of Rows

Choose the "verbose" option, as shown in the figure below, to see details about the intermediate steps in a response: the original data the query emitted, the data that went into the transformer, and the data that the transformer then produced.



Verbose mode shows details about intermediate steps.

If an exception occurred during the run, the console's right-hand panel shows the stacktrace.

NOTE: Fields produced by the entities or transformers may not be visible in documents if the fields are either not present in the `schema.xml` of there is an explicit `<field>` declaration.

6.5 Content Streams

6.5.1 Overview

When SolrRequestHandlers are accessed using path based URLs, the SolrQueryRequest object containing the parameters of the request may also contain a list of ContentStreams containing bulk data for the request.

(The name SolrQueryRequest is a bit misleading: it is involved in all requests, regardless of whether it is a query request or an update request.)

6.5.2 Stream Sources

Currently RequestHandlers can get content streams in a variety of ways:

- For multipart file uploads, each file is passed as a stream.
- For POST requests where the content-type is not `application/x-www-form-urlencoded`, the raw POST body is passed as a stream.
- The contents of parameter `stream.body` is passed as a stream.
- If remote streaming is enabled, the contents of each `stream.url` and `stream.file` parameters are fetched and passed as a stream.

If the `contentType` is `application/x-www-form-urlencoded`, the full POST body is parsed as parameters and included in the Solr parameters..

By default, `curl` sends a `contentType="application/x-www-form-urlencoded"` header. If you need to test a SolrContentHeader content stream, you will need to set the content type with the "-H" flag. For example:

```
curl $URL -H 'Content-type:text/xml; charset=utf-8' --data-binary @$f
```

6.5.3 RemoteStreaming

Remote streaming allows you to send the contents of a URL as a stream to a given SolrRequestHandler. You could use remote streaming to send a remote or local file to an update plugin. For security reasons, remote streaming is disabled in the `solrconfig.xml` included in the `example` directory.

NOTE: If you enable streaming, be aware that this allows *anyone* to send a request to any URL or local file. If dump is enabled, it will essentially let anyone to view any file on your system.

```
  <!--Make sure your system has authentication before enabling remote
streaming!-->
  <requestParsers enableRemoteStreaming="true"
multipartUploadLimitInKB="2048" />
```

6.5.4 Debugging Requests

The example `solrconfig.xml` includes a "dump" RequestHandler:

```
  <requestHandler name="/debug/dump" class="solr.DumpRequestHandler" />
```

This handler simply outputs the contents of the SolrQueryRequest using the specified writer type `wt`. This is a useful tool to help understand what streams are available to to the RequestHandlers.

6.6 Summary

Solr 1.4 offers users a variety of ways of uploading and transforming data for use in a Solr index.

- The new Solr Cell framework makes it easier than ever to upload binary files such as PDFs and Microsoft Office files into Solr.
- Update handlers are also available for uploading content in XML or CSV formats.
- You can upload data from a data store using the Data Import Handler. Data store data can be transformed as needed for use within Solr.
- Bulk data can be uploaded in streams.

Once you have uploaded content into Solr, you're ready to consider how Solr should respond to queries. This is the subject of our next chapter.

7 Searching

7.1 Overview of Searching in Solr 1.4

Solr offers a rich, flexible set of features for search. To understand the extent of this flexibility, it's helpful to begin with an overview of the steps and components involved in a Solr search.

When a user runs a search in Solr, the search query is processed by a **request handler**. A request handler is a Solr plug-in that defines the logic to be used when Solr processes a request. Solr supports a variety of request handlers. Some are designed for processing search queries, while others manage tasks such as index replication.

Search applications select a particular request handler by default. In addition, applications can be configured to allow users to override the default selection in preference of a different request handler.

To process a search query, a request handler calls a **query parser**, which interprets the terms and parameters of a query. Different query parsers support different syntax. In Solr 1.4, the default query parser is the DisMax query parser. Solr also includes an earlier “standard” query parser. The “standard” query parser's syntax allows for greater precision in searches, but the DisMax query parser is much more tolerant of errors. The DisMax query parser is designed to provide an experience similar to that of popular search engines such as Google, which rarely display syntax errors to users.

In addition, there are common query parameters that are accepted by all query parsers.

Input to a query parser can include:

- search strings—that is, *terms* to search for in the index
- *parameters for fine-tuning the query* by increasing the importance of particular strings or fields, by applying Boolean logic among the search terms, or by excluding content from the search results
- *parameters for controlling the presentation of the query response*, such as specifying the order in which results are to be presented or limiting the response to particular fields of the search application's schema.

Search parameters may also specify a **query filter**. As part of a search response, a query filter runs a query against the entire index and caches the results. Because Solr allocates a separate cache for filter queries, the strategic use of filter queries can improve search performance. (Despite their similar names, query filters are not related to analysis filters. Query filters perform queries at search time against data already in the index, while analysis filters, such as Tokenizers, parse content for indexing, following specified rules.)

A search query can request that certain terms be highlighted in the search response; that is, the selected terms will be displayed in colored boxes so that they "jump out" on the screen of search results. **Highlighting** can make it easier to find relevant passages in long documents returned in a search.[14] Solr includes a rich set of search parameters for controlling how terms are highlighted. (See page 234 for details.)

Search responses can also be configured to include **snippets** (document excerpts) featuring highlighted text. Popular search engines such as Google and Yahoo! return snippets in their search results: 3-4 lines of text offering a description of a search result.

To help users zero in on the content they're looking for, Solr supports two special ways of grouping search results to aid further exploration: faceting and clustering.

Faceting is the arrangement of search results into categories (which are based on indexed terms). Within each category, Solr reports on the number of hits for relevant term, which is called a facet constraint.

14 Solr 1.3 does not support multi-term highlighting, but Solr 1.4 will.

Faceting makes it easy for users to explore search results on sites such as movie sites and product review sites, where there are many categories and many items within a category.

The image below shows an example of faceting from the CNET Web site, which was the first site to use Solr.

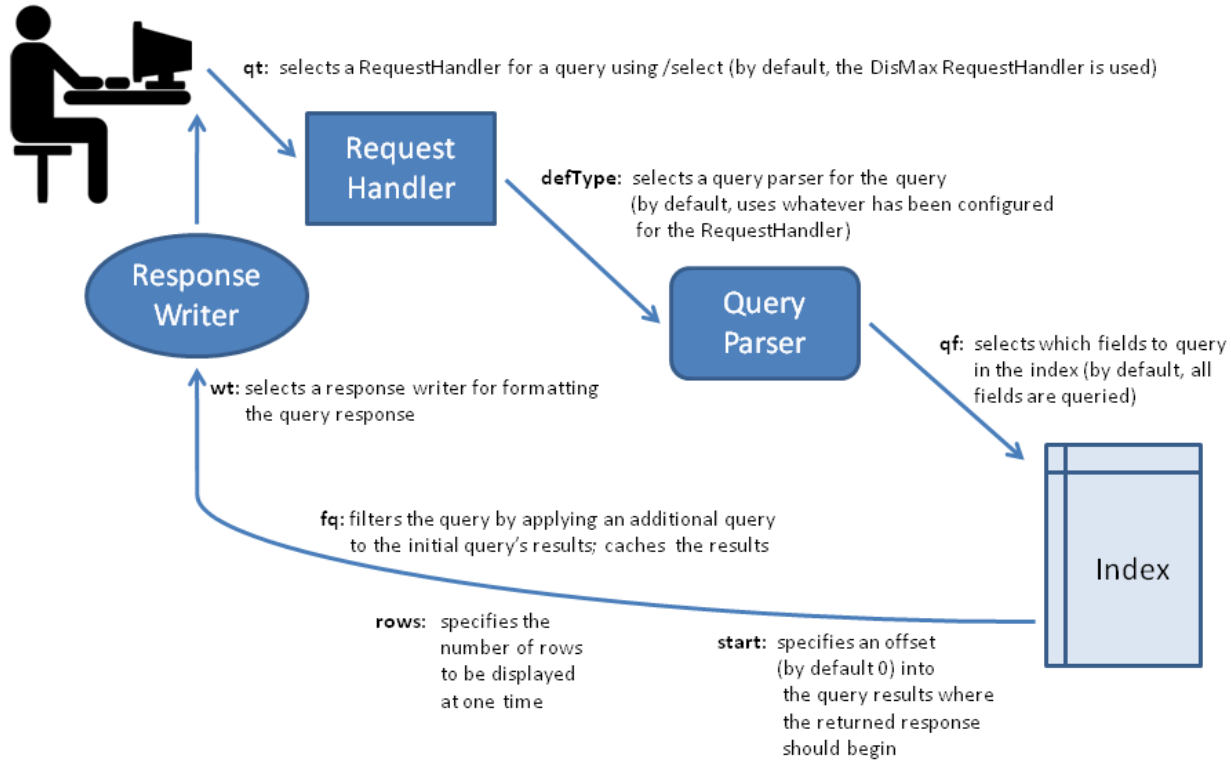
Faceting makes use of fields defined when the search applications were indexed. In the example above, these fields include categories of information that are useful for describing digital cameras: manufacturer, resolution, and zoom range.

Clustering groups search results by similarities discovered when a search is executed, rather than when content is indexed. The results of clustering often lack the neat hierarchical organization found in faceted search results, but clustering can be useful nonetheless. It can reveal unexpected commonalities among search results, and it can help users rule out content that isn't pertinent to what they're really searching for.

Solr also supports a feature called MoreLikeThis, which enables users to submit new queries that focus on particular terms returned in an earlier query. MoreLikeThis queries can make use of faceting or clustering to provide additional aid to users.

A Solr component called a **response writer** manages the final presentation of the query response. Solr includes a variety of response writers, including an XML Response Writer and a JSON Response Writer.

The diagram below summarizes some key elements of the search process.



7.2 Relevance

Relevance is the degree to which a query response satisfies a user who is searching for information.

The relevance of a query response depends on the context in which the query was performed. A single search application may be used in different contexts by users with different needs and expectations. For example, a search engine of climate data might be used by a university researcher studying long-term climate trends, a farmer interested in calculating the likely date of the last frost of spring, a civil engineer interested in rainfall patterns and the frequency of floods, and a college student planning a vacation to a

region and wondering what to pack. Because the motivations of these users vary, the relevance of any particular response to a query will vary as well.

How comprehensive should query responses be? Like relevance in general, the answer to this question depends on the context of a search. The cost of *not* finding a particular document in response to a query is high in some contexts, such as a legal e-discovery search in response to a subpoena, and quite low in others, such as a search for a cake recipe on a Web site with dozens or hundreds of cake recipes. When configuring Solr, you should weigh comprehensiveness against other factors such as timeliness and ease-of-use.

The e-discovery and recipe examples demonstrate the importance of two concepts related to relevance:

- **Precision** is the percentage of documents in the returned results that are relevant.
- **Recall** is the percentage of relevant results returned out of all relevant results in the system. Obtaining perfect recall is trivial: simply return every document in the collection for every query.

Returning to the examples above, it's important for an e-discovery search application to have 100% recall returning all the documents that are relevant to a subpoena. It's far less important that a recipe application offer this degree of precision, however. In some cases, returning too many results in casual contexts could overwhelm users. In some contexts, returning fewer results that have a higher likelihood of relevance may be the best approach.

Using the concepts of precision and recall, it's possible to quantify relevance across users and queries for a collection of documents. A perfect system would have 100% precision and 100% recall for every user and every query. In other words, it would retrieve all the relevant documents and nothing else. In practical terms, when talking about precision and recall in real systems, it is common to focus on precision and recall at a certain number of results, the most common (and useful) being ten results.

Through faceting, query filters, and other search components, a Solr application can be configured with the flexibility to help users fine-tune their searches in order to return the most relevant results for users. That is, Solr can be configured to balance precision and recall to meet the needs of a particular user community.

The configuration of a Solr application should take into account:

- the needs of the application's various users (which can include ease of use and speed of response, in addition to strictly informational needs)
- the categories that are meaningful to these users in their various contexts (e.g., dates, product categories, or regions)
- any inherent relevance of documents (e.g., it might make sense to ensure that an official product description or FAQ is always returned near the top of the search results)
- whether or not the age of documents matters significantly (in some contexts, the most recent documents might always be the most important)

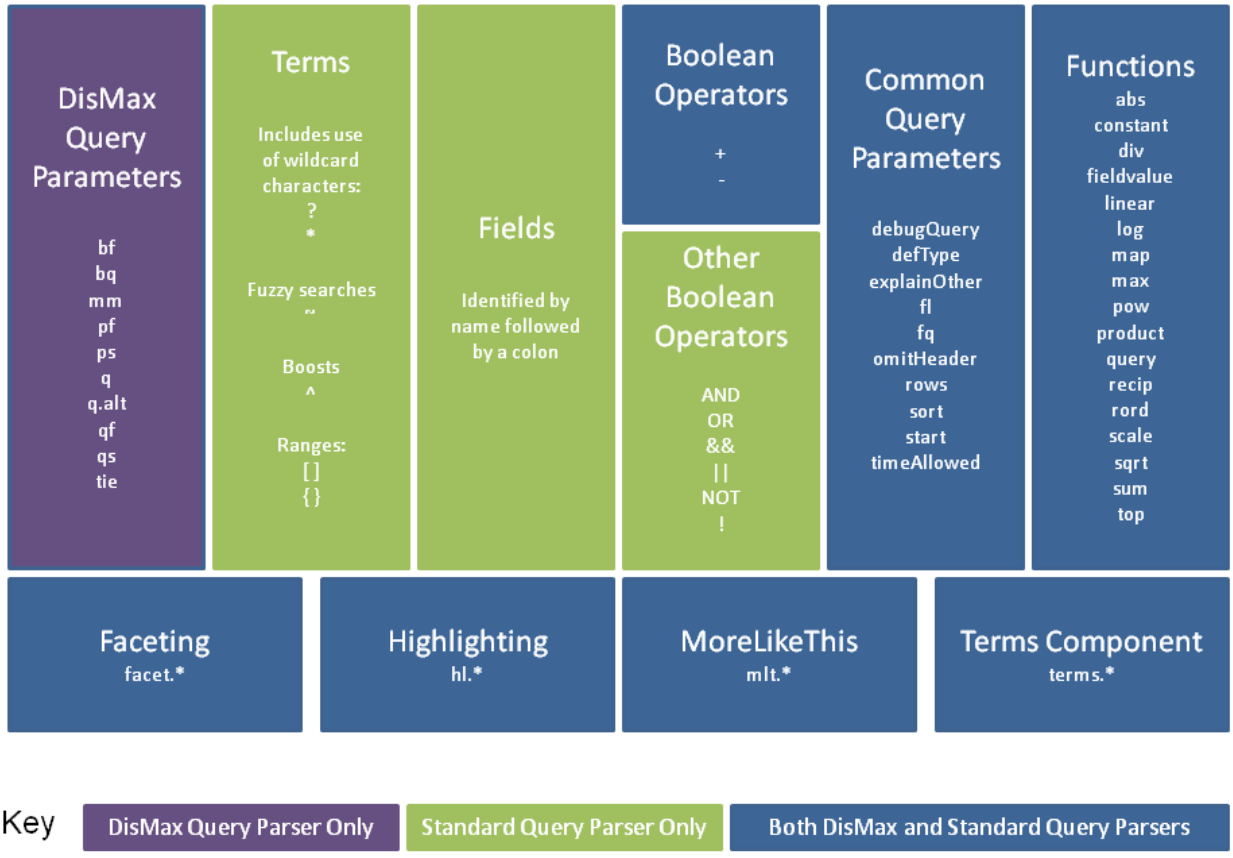
Keeping all these factors in mind, it's often helpful in the planning stages of a Solr deployment to sketch out the types of responses you think the search application should return for sample queries. Once the application is up and running, you can employ a series of testing methodologies, such as focus groups, in-house testing, TREC tests (http://trec.nist.gov) and A/B testing to fine tune the configuration of the application to best meet the needs of its users.

For more information about relevance, see Grant Ingersoll's tech article, “Debugging Relevance Issues in Search,” which is available on the Lucid Imagination Web site.

7.3 Query Syntax and Parsing

Solr supports several query parsers, offering search application designers great flexibility in controlling how queries are parsed. This section explains how to specify the query parser to be used. It also describes the syntax and features supported by the query parsers included with Solr.

The next figure summarizes the parameters supported by the two principal Solr query parsers: the default DisMax query parser and the “standard” Lucene query parser.



7.4 The DisMax Query Parser

NOTE: Solr 1.3 included a DisMax Request Handler, which is now deprecated. To call the DisMax query parser, we recommend that you call the StandardRequestHandler and use the parameter `defType=dismax` to select the DisMax query parser.

The DisMax query parser is designed to process simple phrases (without complex syntax) entered by users and to search for individual terms across several fields using different weighting (boosts) based on the significance of each field. Additional options enable users to influence the score based on rules specific to each use case (independent of user input).

In general, the DisMax query parser's interface is more like that of Google than the interface of the "standard" Solr request handler. This similarity makes DisMax the appropriate query parser for many consumer applications. It accepts a simple syntax, and it rarely produces error messages.

The DisMax query parser supports an extremely simplified subset of the Lucene QueryParser syntax. As in Lucene, quotes can be used to group phrases, and +/- can be used to denote mandatory and optional clauses. All other Lucene query parser special characters (except AND and OR) are escaped to simplify the user experience. The DisMax query parser takes responsibility for building a good query from the user's input using Boolean clauses containing DisMax queries across fields and boosts specified by the user. It also lets the Solr administrator provide additional boosting queries, boosting functions, and filtering queries to artificially affect the outcome of all searches. These options can all be specified as default parameters for the handler in the `solrconfig.xml` file or overridden in the Solr query URL.

7.4.1 DisMax Defined

Interested in the technical concept behind the DisMax name? DisMax stands for Maximum Disjunction. Here's a definition of a Maximum Disjunction or "DisMax" query:

> *A query that generates the union of documents produced by its subqueries, and that scores each document with the maximum score for that document as produced by any subquery, plus a tie breaking increment for any additional matching subqueries.*

Whether or not you remember this explanation, do remember that the DisMax request handler was primarily designed to be easy to use and to accept almost any input without returning an error.

7.4.2 DisMax Parameters

In addition to the common request parameter, highlighting parameters, and simple facet parameters, the DisMax query parser supports the parameters described below. Like the standard query parser, the DisMax query parser allows default parameter values to be specified in `solrconfig.xml`, or overridden by query-time values in the request.

Parameter	Description
q	Defines the raw input strings for the query.
q.alt	Calls the standard query parser and defines query input strings, when the q parameter is not used.
qf	Query Fields: specifies the fields in the index on which to perform the query.
mm	Minimum “Should” Match: specifies a minimum number of fields that must match in a query.
pf	Phrase Fields: boosts the score of documents in cases where all of the terms in the q parameter appear in close proximity.
ps	Phrase Slop: specifies the number of positions two terms can be apart in order to match the specified phrase.
tie	Tie Breaker: specifies a float value (which should be something much less than 1) to use as tiebreaker in DisMax queries.
bq	Boost Query: specifies a factor by which a term or phrase should be “boosted” in importance when considering a match.
bf	Boost Functions: specifies functions to be applied to boosts. (See 227 for details about function queries.)

The sections below explain these parameters in detail.

7.4.2.1 The q Parameter

The q parameter defines the main "query” constituting the essence of the search. The parameter supports raw input strings provided by users with no special escaping. The + and - characters are treated as "mandatory" and "prohibited" modifiers for terms. Text wrapped in balanced quote characters (for example, "San Jose") is treated as a phrase. Any query containing an odd number of quote characters is evaluated as if there were no quote characters at all.

NOTE: The q parameter does not support wildcard characters such as *.

7.4.2.2 The q.alt Parameter

If specified, the `q.alt` parameter defines a query (which by default will be parsed using standard query parsing syntax) when the main `q` parameter is not specified or is blank. The `q.alt` parameter comes in handy when you need something like a query to match all documents (don't forget `&rows=0` for that one!) in order to get collection-wise faceting counts.

7.4.2.3 The qf (Query Fields) Parameter

The `qf` parameter introduces a list of fields, each of which is assigned a boost factor to increase or decrease that particular field's importance in the query. For example, the query below:

```
qf="fieldOne^2.3 fieldTwo fieldThree^0.4"
```

assigns `fieldOne` a boost of 2.3, leaves `fieldTwo` with the default boost (because no boost factor is specified), and `fieldThree` a boost of 0.4. These boost factors make matches in `fieldOne` much more significant than matches in `fieldTwo`, which in turn are much more significant than matches in `fieldThree`.

7.4.2.4 The mm (Minimum Should Match) Parameter

When processing queries, Lucene/Solr recognizes three types of clauses: mandatory, prohibited, and "optional" (also known as "should" clauses). By default, all words or phrases specified in the `q` parameter are treated as "optional" clauses unless they are preceded by a "+" or a "-". When dealing with these "optional" clauses, the `mm` parameter makes it possible to say that a certain minimum number of those clauses must match. The DisMax query parser offers great flexibility in how the minimum number can be specified.

The table below explains the various ways that `mm` values can be specified.

Syntax	Example	Description
Positive integer	`3`	Defines the minimum number of clauses that must match, regardless of how many clauses there are in total.
Negative integer	`-2`	Sets the minimum number of matching clauses to the total number of optional clauses, minus this value.
Percentage	`75%`	Sets the minimum number of matching clauses to this percentage of the total number of optional clauses. The number computed from the percentage is rounded down and used as the minimum.
Negative percentage	`-25%`	Indicates that this percent of the total number of optional clauses can be missing. The number computed from the percentage is rounded down, before being subtracted from the total to determine the minimum number.
An expression beginning with a positive integer followed by a > or < sign and another value	`3<90%`	Defines a conditional expression indicating that if the number of optional clauses is equal to (or less than) the integer, they are all required, but if it's greater than the integer, the specification applies. In this example: if there are 1 to 3 clauses they are all required, but for 4 or more clauses only 90% are required.
Multiple conditional expressions involving > or < signs	`2<-25% 9<-3`	Defines multiple conditions, each one being valid only for numbers greater than the one before it. In the example at left, if there are 1 or 2 clauses, then both are required. If there are 3-9 clauses all but 25% are required. If there are more then 9 clauses, all but three are required.

When specifying `mm` values, keep in mind the following:

- When dealing with percentages, negative values can be used to get different behavior in edge cases. 75% and -25% mean the same thing when dealing with 4 clauses, but when dealing with 5 clauses 75% means 3 are required, but -25% means 4 are required.

- If the calculations based on the parameter arguments determine that no optional clauses are needed, the usual rules about Boolean queries still apply at search time. (That is, a Boolean query containing no required clauses must still match at least one optional clause).
- No matter what number the calculation arrives at, Solr will never use a value greater than the number of optional clauses, or a value less then 1. (In other words, no matter how low or how high the calculated result, the minimum number of required matches will never be less then 1 or greater than the number of clauses.)

The default value of `mm` is 100% (meaning that all clauses must match).

7.4.2.5 The pf (Phrase Fields) Parameter

Once the list of matching documents has been identified using the `fq` and `qf` parameters, the `pf` parameter can be used to "boost" the score of documents in cases where all of the terms in the `q` parameter appear in close proximity.

The format is the same as that used by the `qf` parameter: a list of fields and "boosts" to associate with each of them when making phrase queries out of the entire `q` parameter.

7.4.2.6 The ps (Phrase Slop) Parameter

The `ps` parameter specifies the amount of “phrase slop” to apply to queries specified with the `pf` parameter. Phrase slop is the number of positions one token needs to be moved in relation to another token in order to match a phrase specified in a query.

7.4.2.7 The qs (Query Phrase Slop) Parameter

The `qs` parameter specifies the amount of slop permitted on phrase queries explicitly included in the user's query string with the `qf` parameter. As explained above, slop refers to the number of positions one token needs to be moved in relation to another token in order to match a phrase specified in a query.

7.4.2.8 The tie (Tie Breaker) Parameter

The `tie` parameter specifies a float value (which should be something much less than 1) to use as tiebreaker in DisMax queries.

When a term from the user's input is tested against multiple fields, more than one field may match. If so, each field will generate a different score based on how common that word is in that field (for each document relative to all other documents). The `tie` parameter lets you control how much the final score of the query will be influenced by the scores of the lower scoring fields compared to the highest scoring field.

A value of "0.0" makes the query a pure "disjunction max query": that is, only the maximum scoring subquery contributes to the final score. A value of "1.0" makes the query a pure "disjunction sum query" where it doesn't matter what the maximum scoring sub query is, because the final score will be the sum of the subquery scores. Typically a low value, such as 0.1, is useful.

7.4.2.9 The bq (Boost Query) Parameter

The `bq` parameter specifies a raw query string (expressed in Solr query syntax) that will be included with the user's query to influence the score. For example, if you wanted to add a relevancy boost for recent documents:

```
q=cheese
bq=date[NOW/DAY-1YEAR TO NOW/DAY]
```

You can specify multiple `bq` parameters. If you want your query to be parsed as separate clauses with separate boosts, use multiple `bq` parameters.

7.4.2.10 The bf (Boost Functions) Parameter

The `bf` parameter specifies functions (with optional boosts) that will be included in the user's query to influence the score. Any function supported natively by Solr can be used, along with a boost value. For example:

```
recip(rord(myfield),1,2,3)^1.5
```

Specifying functions with the `bf` parameter is just shorthand for using the `_val_:"...function..."` syntax in a `bq` parameter.

For example, if you want to show the most recent documents first, use

```
recip(rord(creationDate),1,1000,1000)
```

7.4.3 Examples of Queries Submitted to the DisMax Query Parser

```
Normal results for the word "video" using the StandardRequestHandler with
the default search field...
  http://localhost:8983/solr/select/?q=video&fl=name+score&qt=standard

The "dismax" handler is configured to search across the text, features,
name, sku, id, manu, and cat fields all with varying boosts designed to
ensure that "better" matches appear first, specifically: documents which
match on the name and cat fields get higher scores...
  http://localhost:8983/solr/select/?q=video&qt=dismax

...note that this instance is also configured with a default field list,
which can be overridden in the URL...
  http://localhost:8983/solr/select/?q=video&qt=dismax&fl=*,score

You can also override which fields are searched on and how much boost
each field gets...
  http://localhost:8983/solr/select/?q=video&qt=dismax&qf=features^20.0+text^0.3

You can boost results that have a field that matches a specific value...
  http://localhost:8983/solr/select/?
q=video&qt=dismax&bq=cat:electronics^5.0

Another instance of the handler is registered using the qt "instock" and
has slightly different configuration options, notably: a filter for (you
guessed it) inStock:true)...
  http://localhost:8983/solr/select/?q=video&qt=dismax&fl=name,score,inStock
  http://localhost:8983/solr/select/?q=video&qt=instock&fl=name,score,inStock

One of the other really cool features in this handler is robust
support for specifying the "BooleanQuery.minimumNumberShouldMatch" you
want to be used based on how many terms are in your user's query.
These allows flexibility for typos and partial matches.  For the
dismax handler, 1 and 2 word queries require that all of the optional
clauses match, but for 3-5 word queries one missing word is allowed...
  http://localhost:8983/solr/select/?q=belkin+ipod&qt=dismax
  http://localhost:8983/solr/select/?q=belkin+ipod+gibberish&qt=dismax
  http://localhost:8983/solr/select/?q=belkin+ipod+apple&qt=dismax

Just like the StandardRequestHandler, it supports the debugQuery
option to viewing the parsed query, and the score explanations for each
doc...
  http://localhost:8983/solr/select/?
q=belkin+ipod+gibberish&qt=dismax&debugQuery=true
```

```
  http://localhost:8983/solr/select/?
q=video+card&qt=dismax&debugQuery=true
```

7.5 The Standard Query Parser

This section describes the parameters accepted by the standard query parser.

Before Solr1.3, the Standard Request Handler called the standard query parser as the default query parser. In Solr 1.3, the Standard Request Handler calls the DisMax query parser as the default query parser. You can configure Solr to call the standard query parser instead, if you like.

The advantage of the standard query parser is that it enables users to specify very precise queries. The disadvantage is that it's less tolerant of syntax errors than the DisMax query parser. The DisMax query parser is designed to throw as few errors as possible.

7.5.1 Standard Query Parser Parameters

In addition to the Common Query Parameters, Faceting Parameters, Highlighting Parameters, and MoreLikeThis Parameters, the standard query parser supports the parameters described in the table below.

Parameter	Description
`q`	Defines a query using standard query syntax. This parameter is mandatory.
`q.op`	Specifies the default operator for query expressions, overriding the default operator specified in the `schema.xml` file. Possible values are "AND" or "OR".
`df`	Specifies a default field, overriding the definition of a default field in the `schema.xml` file.

Default parameter values are specified in `solrconfig.xml`, or overridden by query-time values in the request.

7.5.2 The Standard Query Parser's Response

By default, the response from the standard query parser contains one `<result>` block, which is unnamed. If the `debugQuery` parameter is used, then an additional `<lst>` block will be returned, using the name "debug". This will contain some useful debugging info, including the original query string, the parsed query string, and explain info for each document in the `<result>` block. If the `explainOther` parameter is also used, then additional explain info will be provided for all the documents matching that query.

7.5.2.1 Sample Responses

This section presents examples of responses from the standard query parser.

The URL below submits a simple query and requests the XML Response Writer to use indentation to make the XML response more readable.

```
http://yourhost.tld:9999/solr/select?q=id:SP2514N&version=2.1&indent=1
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<responseHeader><status>0</status><QTime>1</QTime></responseHeader>

<result numFound="1" start="0">
 <doc>
  <arr name="cat"><str>electronics</str><str>hard drive</str></arr>
  <arr name="features"><str>7200RPM, 8MB cache, IDE Ultra ATA-
133</str><str>NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing
(FDB) motor</str></arr>

  <str name="id">SP2514N</str>
  <bool name="inStock">true</bool>
  <str name="manu">Samsung Electronics Co. Ltd.</str>
  <str name="name">Samsung SpinPoint P120 SP2514N - hard drive - 250 GB -
ATA-133</str>
  <int name="popularity">6</int>
  <float name="price">92.0</float>

  <str name="sku">SP2514N</str>
 </doc>
</result>
```

```
</response>
```

Here's an example of a query with a limited field list.

```
http://yourhost.tld:9999/solr/select?q=id:SP2514N&version=2.1&indent=1&fl=id+name
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<responseHeader><status>0</status><QTime>2</QTime></responseHeader>

<result numFound="1" start="0">
 <doc>
  <str name="id">SP2514N</str>
  <str name="name">Samsung SpinPoint P120 SP2514N - hard drive - 250 GB -
ATA-133</str>
 </doc>

</result>
</response>
```

7.5.3 Specifying Terms for the Standard Query Parser

A query to the standard query parser is broken up into terms and operators. There are two types of terms: single terms and phrases.

- A single term is a single word such as “test” or “hello.”
- A phrase is a group of words surrounded by double quotes such as “hello dolly.”

Multiple terms can be combined together with Boolean operators to form more complex queries (as described below).

> **NOTE:** It is important that the analyzer used for queries parses terms and phrases in a way that is consistent with the way the analyzer used for indexing parses terms and phrases; otherwise, searches may produce unexpected results.

7.5.3.1 Term Modifiers

Solr supports a variety of term modifiers that add flexibility or precision, as needed, to searches. These modifiers include wildcard characters, characters for making a search "fuzzy" or more general, and so on. The sections below describe these modifiers in detail.

7.5.3.2 Wildcard Searches

Solr's standard query parser supports single and multiple character wildcard searches within single terms.

NOTE: Wildcard characters can be applied to single terms, but not to search phrases.

Wildcard Search Type	Special Character	Example
Single character (matches a single character)	?	The search string: `te?t` would match both test and text.

Multiple characters (matches zero or more sequential characters)	*	The wildcard search:
		`tes*`
		would match `test`, `testing`, and `tester`.
		You can also use wildcard characters in the middle of a term. For example:
		`te*t`
		would match `test` and `text`.
		`*est`
		would match `pest` and `test`.

NOTE: As of Solr 1.4, you can use a * or ? symbol as the first character of a search with the standard query parser.

7.5.3.3 Fuzzy Searches

Solr's standard query parser supports fuzzy searches based on the Levenshtein Distance or Edit Distance algorithm. Fuzzy searches discover terms that are similar to a specified term without necessarily being an exact match. To perform a fuzzy search, use the tilde ~ symbol at the end of a single-word term. For example, to search for a term similar in spelling to "roam," use the fuzzy search:

```
roam~
```

This search will match terms like foam and roams. It will also match the word “roam” itself.

An optional, additional parameter specifies the degree of similarity required for a match in a fuzzy search. The value must be between 0 and 1. When set closer to 1, the optional parameter causes only terms with a higher similarity to be matched. For example, the search below requires a high degree of similarity to the term “roam” in order for Solr to return a match:

```
roam~0.8
```

If this numerical parameter is omitted, Lucene performs the search as though the parameter were set to 0.5.

NOTE: The sample query above is not very scalable. Upon parsing this query will check the quasi-edit distance for every term in the index. As a result, this query is practical for only very small indexes.

NOTE: In many cases, stemming (reducing terms to a common stem) can produce similar effects to fuzzy searches and wildcard searches.

7.5.3.4 Proximity Searches

A proximity search looks for terms that are within a specific distance from one another.

To perform a proximity search, add the tilde character ~ and a numeric value to the end of a search phrase. For example, to search for a “apache” and “jakarta” within 10 words of each other in a document, use the search:

```
"jakarta apache"~10
```

NOTE: The distance referred to here is the number of term movements needed to match the specified phrase. In the example above, if “apache” and “jakarta” were 10 spaces apart in a field, but “apache” appeared before “jakarta”, more than 10 term movements would be required to move the terms together and position “apache” to the right of “jakarta” with a space in between.

7.5.3.5 Range Searches

A range search specifies a range of values for a field (a range with an upper bound and a lower bound). The query matches documents whose values for the specified field or fields fall within the range. Range queries can be inclusive or exclusive of the upper and lower bounds. Sorting is done lexicographically, except on numeric fields. For example, the range query below matches all documents whose `mod_date` field has a value between 20020101 and 20030101, inclusive.

```
mod_date:[20020101 TO 20030101]
```

Range queries are not limited to date fields or even numerical fields. You could also use range queries with non-date fields:

```
title:{Aida TO Carmen}
```

This will find all documents whose titles are between Aida and Carmen, but not including Aida and Carmen.

The brackets around a query determine its inclusiveness.

- Square brackets `[ ]` denote an inclusive range query that matches values including the upper and lower bound.
- Curly brackets `{ }` denote an exclusive range query that matches values between the upper and lower bounds, but excluding the upper and lower bounds themselves.

7.5.3.6 Boosting a Term with ^

Lucene/Solr provides the relevance level of matching documents based on the terms found. To boost a term use the caret symbol ^ with a boost factor (a number) at the end of the term you are searching. The higher the boost factor, the more relevant the term will be.

Boosting allows you to control the relevance of a document by boosting its term. For example, if you are searching for

```
jakarta apache
```

and you want the term “jakarta” to be more relevant, you can boost it by adding the ^ symbol along with the boost factor immediately after the term. For example, you could type:

```
jakarta^4 apache
```

This will make documents with the term jakarta appear more relevant. You can also boost Phrase Terms as in the example:

```
"jakarta apache"^4 "Apache Lucene"
```

By default, the boost factor is 1. Although the boost factor must be positive, it can be less than 1 (for example, it could be 0.2).

7.5.4 Specifying Fields in a Query to the Standard Query Parser

Data indexed in Solr is organized in fields, which are defined in the Solr `schema.xml` file. Searches can take advantage of fields to add precision to queries. For example, you can search for a term only in a specific field, such as a title field.

The `schema.xml` file defines one field as a default field. If you do not specify a field in a query, Solr searches only the default field. Alternatively, you can specify a different field or a combination of fields in a query.

To specify a field, type the field name followed by a colon ":" and then the term you are searching for within the field.

For example, suppose an index contains two fields, `title` and `text`, and that `text` is the default field. If you want to find a document called “The Right Way” which contains the text “don't go this way,” you could include either of the following terms in your search query:

```
title:"The Right Way" AND text:go
```

```
title:"Do it right" AND go
```

Since `text` is the default field, the field indicator is not required; hence the second query above omits it.

Note: The field is only valid for the term that it directly precedes, so the query
`title:Do it right`
will find only "Do" in the title field. It will find "it" and "right" in the default field (in this case the text field).

7.5.5 Boolean Operators Supported by the Standard Query Parser

Boolean operators allow you to apply Boolean logic to queries, requiring the presence or absence of specific terms or conditions in fields in order to match documents. The table below summarizes the Boolean operators supported by the standard query parser.

Boolean Operator	Alternative Symbol	Description
AND	&&	Requires both terms on either side of the Boolean operator to be present for a match.
NOT	!	Requires that the following term not be present.
OR	||	Requires that either term (or both terms) be present for a match.
	+	Requires that the following term be present.
	-	Prohibits the following term (that is, matches on fields or documents that do not include that term). The – operator is functional similar to the Boolean operator !. Because it's used by popular search engines such as Google, it may be more familiar to some user communities.

Boolean operators allow terms to be combined through logic operators. Lucene supports AND, "+", OR, NOT and "-" as Boolean operators.

NOTE: When specifying Boolean operators with keywords such as AND or NOT, the keywords must appear in all uppercase.

NOTE: The standard query parser supports all the Boolean operators listed in the table above. The DisMax query parser supports only + and -.

The OR operator is the default conjunction operator. This means that if there is no Boolean operator between two terms, the OR operator is used. The OR operator links two terms and finds a matching document if either of the terms exist in a document. This is equivalent to a union using sets. The symbol || can be used in place of the word OR.

NOTE: In the `schema.xml` file, you can specify which symbols can take the place of Boolean operators such as OR.

To search for documents that contain either "jakarta apache" or just "jakarta," use the query:

```
"jakarta apache" jakarta
```

or

```
"jakarta apache" OR jakarta
```

7.5.5.1 The Boolean Operator +

The + symbol (also known as the "required" operator) requires that the term after the + symbol exist somewhere in a field in at least one document in order for the query to return a match.

For example, to search for documents that must contain "jakarta" and that may or may not contain "lucene," use the following query:

```
+jakarta lucene
```

NOTE: This operator is supported by both the standard query parser and the DisMax query parser.

7.5.5.2 The Boolean Operator AND (&&)

The `AND` operator matches documents where both terms exist anywhere in the text of a single document. This is equivalent to an intersection using sets. The symbol `&&` can be used in place of the word `AND`.

To search for documents that contain "jakarta apache" and "Apache Lucene," use either of the following queries:

```
"jakarta apache" AND "Apache Lucene"

"jakarta apache" && "Apache Lucene"
```

7.5.5.3 The Boolean Operator NOT (!)

The `NOT` operator excludes documents that contain the term after `NOT`. This is equivalent to a difference using sets. The symbol ! can be used in place of the word NOT.

The following queries search for documents that contain the phrase "jakarta apache" but do not contain the phrase "Apache Lucene":

```
"jakarta apache" NOT "Apache Lucene"

"jakarta apache" ! "Apache Lucene"
```

NOTE: The NOT operator cannot be used with just one term. For example, the following search will return no results:

```
NOT "jakarta apache"
```

7.5.5.4 The Boolean Operator -

The - symbol or "prohibit" operator excludes documents that contain the term after the - symbol.

For example, to search for documents that contain "jakarta apache" but not "Apache Lucene," use the following query:

```
"jakarta apache" -"Apache Lucene"
```

7.5.6 Special Topic: Grouping Terms to Form Subqueries

Lucene/Solr supports using parentheses to group clauses to form subqueries. This can be very useful if you want to control the Boolean logic for a query.

The query below searches for either "jakarta" or "apache" and "website":

```
(jakarta OR apache) AND website
```

This adds precision to the query, requiring that the term “website” exist, along with either term “jakarta” and “apache.”

7.5.6.1 Grouping Clauses within a Field

To apply two or more Boolean operators to a single field in a search, group the Boolean clauses within parentheses.

For example, the query below searches for a title field that contains both the word “return” and the phrase “pink panther”:

```
title:(+return +"pink panther")
```

7.5.7 Escaping Special Characters

Solr gives the following characters special meaning when they appear in a query.

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \
```

To make Solr interpret any of these characters literally, rather as a special character, precede the character with a backslash character \. For example, to search for (1+1):2 without having Solr interpret the plus sign and parentheses as special characters for formulating a subquery with two terms, escape the characters by preceding each one with a backslash:

```
\(1\+1\)\:2
```

7.5.8 Differences between Lucene Query Parser and the Solr Standard Query Parser

Solr's standard query parser differs from the Lucene Query Parser in the following ways:

- A * may be used for either or both endpoints to specify an open-ended range query.
  - `field:[* TO 100]` finds all field values less than or equal to 100

  - `field:[100 TO *]` finds all field values greater than or equal to 100
  - `field:[* TO *]` matches all documents with the field
- Pure negative queries (all clauses prohibited) are allowed (only as a top-level clause).
  - `-inStock:false` finds all field values where `inStock` is not false
  - `-field:[* TO *]` finds all documents without a value for field
- A hook into FunctionQuery syntax. You'll need to use quotes to encapsulate the function if it includes parentheses, as shown in the second example below.
  - Example: `_val_:myfield`
  - Example: `_val_:"recip(rord(myfield),1,2,3)"`
- Nested query support for any type of query parser. Quotes will often be necessary to encapsulate the nested query if it contains reserved characters.
  - Example: `_query_:"{!dismax qf=myfield}how now brown cow"`

The standard Solr query parser also differs from earlier (pre-2.9) versions of the Lucene query parser in this way:

- Range queries `[a TO z]`, prefix queries `a*`, and wildcard queries `a*b` are constant-scoring (all matching documents get an equal score). The scoring factors TF, IDF, index boost, and coord are not used. There is no limitation on the number of terms that match (as there was in past versions of Lucene).

7.5.8.1 Specifying Dates and Times

If you use the Solr "DateField" type, any queries on those fields (typically range queries) should use the TrieDate Field. In previous releases, you would use the complete ISO 8601 date syntax that "DateField" supports, or the Lucene/Solr DateMathParser's syntax to get relative dates.

Here are some examples of valid parameters using syntax appropriate for the DateField type:

- `timestamp:[*TO NOW]`
- `createdate:[1976-03-06T23:59:59.999Z TO *]`

- `createdate:[1995-12-31T23:59:59.999Z TO 2007-03-06T00:00:00Z]`
- `pubdate:[NOW-1YEAR/DAY TO NOW/DAY+1DAY]`
- `createdate:[1976-03-06T23:59:59.999Z TO 1976-03-06T23:59:59.999Z+1YEAR]`
- `createdate:[1976-03-06T23:59:59.999Z/YEAR TO 1976-03-06T23:59:59.999Z]`

7.6 Common Query Parameters

The table below summarizes Solr's common query parameters, which are supported by both the Standard Request Handler and the (now deprecated) DisMax Requested Handler.

Lucid Imagination strongly recommends that any future SolrRequestHandlers support these parameters, as well.

Parameter	Description
`defType`	Selects the query parser to be used to process the query.
`sort`	Sorts the response to a query in either ascending or descending order based on the response's score or another specified characteristic.
`start`	Specifies an offset (by default, 0) into the responses at which Solr should begin displaying content.
`rows`	Controls how many rows of responses are displayed at a time (default value: 10)
`fq`	Applies a filter query to the search results.
`fl`	Limits the query's responses to a listed set of fields.
`debugQuery`	Causes Solr to include additional debugging information in the response, including "explain" information for each of the documents returned. Note that this parameter takes effect if it is present, regardless of its setting.
`explainOther`	Allows clients to specify a Lucene query to identify a set of documents. If non-blank, the explain info of each document which matches this query, relative to the main query (specified by the `q` parameter) will be returned along with the rest of the debugging information.
`timeAllowed`	Defines the time allowed for the query to be processed. If the time elapses before the query response is complete, partial information may be returned.
`omitHeader`	Excludes the header from the returned results, if set to true. The header contains information about the request, such as the time the request took to complete. The default is false.
`wt`	Specifies the Response Writer to be used to format the query response.

The following sections describe these parameters in detail.

7.6.1 The defType Parameter

The `defType` parameter selects the query parser that Solr should use to process the request. For example:

```
defType=dismax
```

In Solr 1.3 and later, the query parser is set to `dismax` by default.

7.6.2 The sort Parameter

The `sort` parameter arranges search results in either ascending (`asc`) or descending (`desc`) order. The parameter can be used with either numerical or alphabetical content.

Solr can sort query responses according to document scores or the value of any indexed field with a single value (that is, any field whose attributes in `schema.xml` include `multiValued="false"` and `indexed="true"`), provided that:

- the field is non-tokenized (that is, the field has no analyzer and its contents have been been parsed into tokens, which would make the sorting inconsistent), or
- the field uses an analyzer (such as the KeywordTokenizer) that produces only a single term.

If you want to be able to sort on a field whose contents you want to tokenize to facilitate searching, use the `<copyField>` directive in the `schema.xml` file to clone the field. Then search on the field and sort on its clone.

The table explains how Solr responds to various settings of the sort parameter.

Example of a sort Parameter	Result
	If the sort parameter is omitted, sorting is performed as though the parameter were set to `score desc`.
`score desc`	Sorts in descending order from the highest score to the lowest score.
`price asc`	Sorts in ascending order of the `price` field
`inStock desc, price asc`	Sorts by the contents of the `inStock` field in descending order, then within those results sorts in ascending order by the contents of the `price` field.

Regarding the `sort` parameter's arguments:

- A sort ordering must include a field name (or `score` as a pseudo field), followed by whitespace (escaped as + or `%20` in URL strings), followed by a sort direction (`asc` or `desc`).
- Multiple sort orderings can be separated by a comma, using this syntax:
  `sort=<field name>+<direction>[,<field name>+<direction>]...`

7.6.3 The start Parameter

When specified, the `start` parameter specifies an offset into a query's result set and instructs Solr to begin displaying results from this offset.

The default value is "`0`". In other words, by default, Solr returns results without an offset, beginning where the results themselves begin.

Setting the `start` parameter to some other number, such as 3, causes Solr to skip over the preceding records and start at the document identified by the offset.

You can use the `start` parameter this way for paging. For example, if the `rows` parameter is set to 10, you could display three successive pages of results by setting `start` to 0, then re-issuing the same query and setting `start` to 10, then issuing the query again and setting `start` to 20.

7.6.4 The rows Parameter

You can use the `rows` parameter to paginate results from a query. The parameter specifies the maximum number of documents from the complete result set that Solr should return to the client at one time.

The default value is 10. That is, by default, Solr returns 10 documents at a time in response to a query.

7.6.5 The fq (Filter Query) Parameter

The `fq` parameter defines a query that can be used to restrict the superset of documents that can be returned, without influencing score. It can be very useful for speeding up complex queries, since the queries specified with `fq` are cached independently of the main query. When a later query uses the same filter, there's a cache hit, and filter results are returned quickly from the cache.

When using the `fq` parameter, keep in mind the following:

- The `fq` parameter can be specified multiple times in a query. Documents will only be included in the result if they are in the intersection of the document sets resulting from each instance of the parameter. In the example below, only documents which have a popularity greater then 10 and have a section of 0 will match.

```
   fq=popularity:[10 TO *]
 & fq=section:0
```

- Filter queries can involve complicated Boolean queries. The above example could also be written as a single `fq` with two mandatory clauses like so:

```
   fq=+popularity:[10 TO *] +section:0
```

- The document sets from each filter query are cached independently. Thus, concerning the previous examples: use a single `fq` containing two mandatory clauses if those clauses appear together often, and use two separate `fq` parameters if they are relatively independent. (To learn about tuning cache sizes and making sure a filter cache actually exists, see Chapter 8.)

- As with all parameters: special characters in an URL need to be properly escaped and encoded as hex values. [15]

15 Online tools are available to help you with URL-encoding. For example: http://netzreport.googlepages.com/online_tool_for_url_en_decoding.html

7.6.6 The fl (Field List) Parameter

The `fl` parameter limits the information included in a query response to a specified list of fields. The fields need to have been indexed as stored for this parameter to work correctly.

The field list can be specified as a space-separated or comma-separated list of field names. The string "`score`" can be used to indicate that the score of each document for the particular query should be returned as a field. The wildcard character "`*`" selects all the stored fields in a document.

Field List	Result
`id name price`	Return only the id, name, and price fields.
`id,name,price`	Return only the id, name, and price fields.
`id name, price`	Return only the id, name, and price fields.
`id score`	Return the id field and the score.
`*`	Return all the fields in each document. This is the default value of the `fl` parameter.
`* score`	Return all the fields in each document, along with each field's score.

As noted in the table above, the default value is "`*`".

7.6.7 The debugQuery Parameter

If the `debugQuery` parameter is present (regardless of its value), then additional debugging information will be included in the response, including "explain" info for each of the documents returned. (The “explain” info tells you why your query matched and indicates which parts of the query contributed to the overall score.) This debugging info is meant for human consumption. Its XML format could change in future Solr releases.

The default behavior is not to include debugging information.

7.6.8 The explainOther Parameter

The `explainOther` parameter specifies a Lucene query in order to identify a set of documents. If this parameter is included and is set to a non-blank value, the query will return debugging information, along with the "explain info" of each document that matches the Lucene query, relative to the main query (which is specified by the `q` parameter). For example:

```
q=supervillians&debugQuery=on&explainOther=id:juggernaut
```

The query above allows you to examine the scoring explain info of the top matching documents, compare it to the explain info for documents matching `id:juggernaut`, and determine why the rankings are not as you expect.

The default value of this parameter is blank, which causes no extra "explain info" to be returned.

7.6.9 The omitHeader Parameter

This parameter may be set to either `true` or `false`.

If set to `true`, this parameter excludes the header from the returned results. The header contains information about the request, such as the time it took to complete. The default value for this parameter is `false`.

7.6.10 The wt Parameter

The `wt` parameter selects the Response Writer that Solr should use to format the query's response. For detailed descriptions of Response Writers, see page 263.

7.7 Local Parameters in Queries

Local parameters are arguments in a Solr request that are specific to a query parameter. Local parameters provide a way to add meta-data to certain argument types such as query strings. (In Solr documentation, local parameters are sometimes referred to as LocalParams.)

Local parameters are specified as prefixes to arguments. Take the following query argument, for example:

```
q=solr rocks
```

We can prefix this query string with local parameters to provide more information to the Standard Query Parser. For example, we can change the default operator type to "AND" and the default field to "title":

```
q={!q.op=AND df=title}solr rocks
```

These local parameters would change the query to require a match on both “solr” and “rocks” while searching the “title” field by default.

7.7.1 Basic Syntax of Local Parameters

To specify a local parameter, insert the following before the argument to be modified:

- Began with `{!`
- Insert any number of `key=value` pairs separated by white space
- End with `}` and immediately follow with the query argument

You may specify only one local parameters prefix per argument. Values in the key-value pairs may be quoted via single or double quotes, and backslash escaping works within quoted strings.

7.7.2 Query Type Short Form

If a local parameter value appears without a name, it is given the implicit name of "type". This allows short-form representation for the type of query parser to use when parsing a query string. Thus

```
q={!dismax qf=myfield}solr rocks
```

is equivalent to:

```
q={!type=dismax qf=myfield}solr rocks
```

7.7.3 Specifying the Parameter Value with the 'v' Key

A special key of `v` within local parameters is an alternate way to specify the value of that parameter.

```
q={!dismax qf=myfield}solr rocks
```

is equivalent to

```
q={!type=dismax qf=myfield v='solr rocks'}
```

7.7.4 Parameter Dereferencing

Parameter dereferencing or indirection lets you use the value of another argument rather than specifying it directly. This can be used to simplify queries, decouple user input from query parameters, or decouple front-end GUI parameters from defaults set in `solrconfig.xml`.

```
q={!dismax qf=myfield}solr rocks
```

is equivalent to:

```
q={!type=dismax qf=myfield v=$qq}&qq=solr rocks
```

7.8 Function Queries

Function Query parameters enable you to generate a relevancy score using the actual value of one or more numeric fields. Function queries are supported by both the DisMax query parser and the standard query parser.

The table below summarizes the functions available for function queries.

Function	Description	Syntax Examples
`abs`	Returns the absolute value of the specified value or function.	`abs(x)` `abs(-5)`
`constant`	Specifies a floating point constant.	`1.5` `_val_:1.5`
`div`	Divides one value or function by another. `div(x,y)` divides x by y.	`div(1,y)` `div(sum(x,100),max(y,1))`

Function	Description	Syntax Examples
`fieldvalue`	Returns the numeric field value of an indexed (not multi-valued) field with a maximum of one value per document. The syntax is simply the field name by itself. 0 is returned for documents without a value in the field.	`myFloatField` `_val_:myFloatField`
`linear`	Implements `m*x+c` where `m` and `c` are constants and `x` is an arbitrary function. This is equivalent to `sum(product(m,x),c)`, but slightly more efficient as it is implemented as a single function.	`linear(x,m,c)` `linear(x,2,4)` returns `2*x+4`
`log`	Returns the log base 10 of the specified function.	`log(x)` `log(sum(x,100))`

`map`	Maps any values of the function `x` that fall within min and max inclusive to the specified target. The arguments `min,max,target` are constants. The function outputs the field's value if it does not fall between `min` and `max`.	`map(x,min,max,target)`
		`map(x,0,0,1)`
		changes any values of 0 to 1. This can be useful in handling default 0 values.
		
		`map(x,min,max,target,altarg)`
		
		`map(x,0,0,1,0)`
		changes any values of 0 to 1 and if the value is not zero it can be set to the value of the 5th argument instead of defaulting to the field's value.

`max`	Returns the max of another function and a constant, which are specified as arguments: `max(x,c)` The `max` function is useful for "bottoming out" another function at some constant.	`max(myfield,0)`

Function	Description	Syntax Examples

`ms`	Returns milliseconds of difference between its arguments. Dates are relative to the Unix or POSIX time epoch, midnight, January 1, 1970 UTC.
	Arguments may be numerically indexed date fields such as TrieDate (the default in 1.4), or date math based on a constant date or `NOW`.
		`ms()`
		Equivalent to `ms(NOW)`, number of milliseconds since the epoch.
		
		`ms(a)`
		Returns the number of milliseconds since the epoch that the argument represents. Examples:
		`ms(NOW/DAY)`
		`ms(2000-01-01T00:00:00Z)`
		`ms(mydatefield)`
		
		`ms(a,b)`
		Returns the number of milliseconds that `b` occurs before `a` (i.e. `a - b`). Note that this offers higher precision than `sub(a,b)` because the arguments are not converted to floating point numbers before subtraction. Examples:
		`ms(NOW,mydatefield)`
		`ms(mydatefield,2000-01-01T00:00:00Z)`
		`ms(datefield1,datefield2)`

Function	Description	Syntax Examples

`ord`	Returns the ordinal of the indexed field value within the indexed list of terms for that field in Lucene index order (lexicographically ordered by unicode value), starting at 1. In other words, for a given field, all values are ordered lexicographically; this function then returns the offset of a particular value in that ordering. The field must have a maximum of one value per document (not multi-valued). 0 is returned for documents without a value in the field.
	WARNING: `ord()` depends on the position in an index and can thus change when other documents are inserted or deleted.
	See `rord`.
		`ord(`*`myIndexedField`*`)`
		`val_:"ord(myIndexedField)"`
		
		Example: If there were only three values ("apple","banana","pear") for a particular field, then:
		`ord("apple")=1`
		`ord("banana")=2`
		`ord("pear")=3`

`pow`	Raises the specified base to the specified power. `pow(x,y)` raises `x` to the power of `y`.	`pow(x,y)`
		`pow(x,log(y))`
		`pow(x,0.5)` is the same as `sqrt`

`product`	Returns the product of multiple values or functions, which are specified in a comma-separated list.	`product(x,y,...)`
		`product(x,2)`
		`product(x,y)`

Function	Description	Syntax Examples

`query`	Returns the score for the given subquery, or the default value for documents not matching the query.
	Any type of subquery is supported through either parameter dereferencing `$otherparam` or direct specification of the query string in the Local Parameters through the `v` key.
		`query(subquery, default)`
		
		`q=product(popularity, query({!dismax v='solr rocks'}))`
		returns the product of the popularity and the score of the DisMax query.
		
		`q=product(popularity, query($qq))&qq={!dismax}solr rocks`
		is equivalent to the previous query, using parameter dereferencing.
		
		`q=product(popularity, query($qq,0.1))&qq={!dismax}solr rocks`
		specifies a default score of 0.1 for documents that don't match the DisMax query.

`recip`	Performs a reciprocal function with `recip(myfield,m,a,b)` implementing `a/(m*x+b)`. `m,a,b` are constants, and `x` is any arbitrarily complex function. When a and b are equal, and x>=0, this function has a maximum value of 1 that drops as `x` increases. Increasing the value of `a` and `b` together results in a movement of the entire function to a flatter part of the curve. These properties can make this an ideal function for boosting more recent documents when `x` is `rord(datefield)`.	`recip(myfield,m,a,b)` `recip(rord(creationDate),1,1000,1000)`

Function	Description	Syntax Examples

`rord`	Returns the reverse ordering of that returned by `ord`.	`rord(`*`myDateField`*`)`
		`val_:"rord(myDateField)"`
		Example: `rord(myDateField)` is a metric for how old a document is. The youngest document will return 1. The oldest document will return the total number of documents.

`scale`	Scales values of the function x such that they fall between the specified `minTarget` and `maxTarget` inclusive.
	NOTE: The current implementation traverses all of the function values to obtain the min and max, so it can pick the correct scale.
	NOTE: The current implementation cannot distinguish when documents have been deleted or documents that have no value. It uses 0.0 values for these cases. This means that if values are normally all greater than 0.0, one can still end up with 0.0 as the min value to map from. In these cases, an appropriate `map()` function could be used as a workaround to change 0.0 to a value in the real range, as shown here:
	`scale(map(x,0,0,5),1,2)`
		`scale(x,minTarget,maxTarget)`
		`scale(x,1,2)`
		scales the values of `x` such that all values will be between 1 and 2 inclusive.

`sqrt`	Returns the square root of the specified value or function.	`sqrt(x)`
		`sqrt(100)`
		`sqrt(sum(x,100))`

`sub`	Returns `x-y` from `sub(x,y)`.	`sub(myfield,myfield2)` `sub(100,sqrt(myfield))`

Function	Description	Syntax Examples

`sum`	Returns the sum of multiple values or functions, which are specified in a comma-separated list.	`sum(x,y,...)`
		`sum(x,1)`
		`sum(x,y)`
		`sum(sqrt(x),log(y),z,0.5)`

`top`	Causes the function query argument to derive its values from the top-level IndexReader containing all parts of an index. For example, the ordinal of a value in a single segment will be different from the ordinal of that same value in the complete index. The `ord()` and `rord()` functions implicitly use `top()`, and hence `ord(foo)` is equivalent to `top(ord(foo))`.	

7.8.1 Using FunctionQuery

There are two principal ways of including function queries in a Solr query:

- Introduce a function query with the `_val_` keyword. For example:

```
_val_:mynumericfield
_val_:"recip(rord(myfield),1,2,3)"
```

- Use a parameter that has an explicit type of FunctionQuery, such as the DisMax query parser's `bf` (boost function) parameter.

  Note that the `bf` parameter actually takes a list of function queries separated by white space and each with an optional boost. Make sure you eliminate any internal white space in single function queries when using `bf`. For example:

```
q=dismax&bf="ord(popularity)^0.5 recip(rord(price),1,1000,1000)^0.3"
```

Functions must be expressed as function calls (e.g. `sum(a,b)` instead of simply `a+b`).

7.8.2 Example of Function Queries Using the top Function

To give you a better understanding of how function queries can be used in Solr, suppose an index stores the dimensions in meters x,y,z of some hypothetical boxes with arbitrary names stored in field `boxname`. Suppose we want to search for box matching name `findbox` but ranked according to volumes of boxes. The query parameters would be:

```
q=boxname:findbox _val_:"product(product(x,y),z)
```

This query will rank the results based on volumes. In order to get the computed volume, you will need to add the parameter:

```
&fl=*, score
```

where `score` will contain the resultant volume.

Suppose that you also have a field storing the weight of the box as 'weight'. To sort by the density of the box and return the value of the density in `score`, you would submit the following query:

```
http://localhost:8983/solr/select/?q=boxname:findbox
_val_:"div(weight,product(product(x,y),z))"&fl=boxname x y z weight
score
```

7.9 Highlighting

Solr provides a collection of highlighting utilities which can be called by various Request Handlers to include "highlighted" matches in field values. These highlighting utilities may be used with either the DisMax query parser or the standard query parser.

NOTE: Only text that has been both indexed and stored may be highlighted.

> **NOTE:** Some parameters may be overridden on a per-field basis with the following syntax:
> `f.<fieldName>.<originalParam>=<value>`
> For example:
> `f.contents.hl.snippets=2`

The table below describes Solr's parameters for highlighting.

Parameter	Description
hl	When set to "true", enables highlighted snippets to be generated in the query response. If set to "false" or to a blank or missing value, disables highlighting. The default value is blank, which disables highlighting.
hl.fl	Specifies a list of fields to highlight. Accepts a comma- or space-delimited list of fields for which Solr should generate highlighted snippets. If left blank, highlights the defaultSearchField (or the field specified the df parameter if used) for the StandardRequestHandler. For the DisMaxRequestHandler, the qf fields are used as defaults. A '*' can be used to match field globs, e.g. 'text_*' or even '*' to highlight on all fields where highlighting is possible. When using '*', consider adding hl.requireFieldMatch=true. The default value is blank.
hl.snippets	Specifies maximum number of highlighted snippets to generate per field. Note: it is possible for any number of snippets from zero to this value to be generated. This parameter accepts per-field overrides. The default value is "1".
hl.fragsize	Specifies the size, in characters, of fragments to consider for highlighting. "0" indicates that the whole field value should be used (no fragmenting). This parameter accepts per-field overrides. The default value is "100".
hl.mergeContinuous	Instructs Solr to collapse contiguous fragments into a single fragment. "true" indicates contiguous fragments will be collapsed into single fragment. This parameter accepts per-field overrides. The default value is "false", which is also the backward-compatible setting.

Parameter	Description
hl.requireFieldMatch	If set to true, highlights terms only if they appear in the specified field. Normally, terms are highlighted in all requested fields regardless of which field matched the query. The default value is "false".
hl.maxAnalyzedChars	Specifies the number of characters into a document that Solr should look for suitable snippets. The default value is "51200".
hl.alternateField	Specifies a field to be used as a backup default summary if Solr cannot generate a snippet (because no terms match). This parameter accepts per-field overrides. By default, Solr does not select a field for a backup summary.
hl.maxAlternateFieldLength	Specifies the maximum number of characters of the field to return. Any value less than or equal to 0 means the field's length is unlimited. The default value is unlimited. Requires the use of the hl.alternateField parameter.
hl.formatter	Selects a formatter for the highlighted output. Currently the only legal value is "simple", which surrounds a highlighted term with a customizable pre- and post-text snippet. This parameter accepts per-field overrides. The default value is "simple".
hl.simple.pre hl.simple.post	Specifies the text that should appear before and after a highlighted term when using the simple formatter. This parameter accepts per-field overrides. The default values are "" and "".

Parameter	Description
hl.fragmenter	Specifies a text snippet generator for highlighted text. The standard fragmenter is gap (which is so called because it creates fixed-sized fragments with gaps for multi-valued fields). Another option is regex, which tries to create fragments that resemble a specified regular expression. The hl.fragmenter parameter accepts per-field overrides. The default value is gap.
hl.usePhraseHighlighter	If set to "true," instructs Solr to use the Lucene SpanScorer class to highlight phrase terms only when they appear within the query phrase in the document. The default is "true."
hl.highlightMultiTerm	If set to "true," instructs Solr to highlight phrase terms that appear in multi-term queries. The default is "true."
hl.regex.slop	Specifies the factor by which the regex fragmenter can stray from the ideal fragment size (given by hl.fragsize) to accommodate a regular expression. For instance, a slop of 0.2 with fragsize of 100 should yield fragments between 80 and 120 characters in length. It is usually good to provide a slightly smaller fragsize when using the regex fragmenter. The default value is 0.6.
hl.regex.pattern	Specifies the regular expression for fragmenting. This could be used to extract sentences.
hl.regex.maxAnalyzedChars	Instructs Solr to analyze only this many characters from a field when using the regex fragmenter (after which, the fragmenter produces fixed-sized fragments). Applying a complicated regex to a huge field is computationally "expensive." The default value is "10000".

7.10 MoreLikeThis

The MoreLikeThis component enables users to query for results similar to the specified terms.

MoreLikeThis constructs a Lucene query based on terms in a document. For best results, use stored TermVectors in the `schema.xml` for fields specified for similarity. For example:

```
<field name="cat" ... termVectors="true" />
```

If termVectors are not stored, MoreLikeThis will generate terms from stored fields.

7.10.1 Common Parameters for MoreLikeThis

The table below summarizes the MoreLikeThis parameters supported by Lucene/Solr.

Parameter	Description
`mlt.fl`	Specifies the fields to use for similarity. NOTE: if possible, these should have a stored TermVector.
`mlt.mintf`	Specifies the Minimum Term Frequency—the frequency below which terms will be ignored in the source doc.
`mlt.mindf`	Specifies the Minimum Document Frequency—the frequency at which words will be ignored which do not occur in at least this many docs.
`mlt.minwl`	Sets the minimum word length below which words will be ignored.
`mlt.maxwl`	Sets the maximum word length above which words will be ignored.
`mlt.maxqt`	Sets the maximum number of query terms that will be included in any generated query.
`mlt.maxntp`	Sets the maximum number of tokens to parse in each example document field that is not stored with TermVector support.
`mlt.boost`	[true/false] set if the query will be boosted by the interesting term relevance.
`mlt.qf`	Query fields and their boosts using the same format as that used

Parameter	Description
	by the DisMaxRequestHandler. These fields must also be specified in `mlt.fl`.

7.10.2 Parameters for the StandardRequestHandler

This method returns similar documents for each document in the response set.

Parameter	Description
`mlt`	If set to true, activates the MoreLikeThis component and enables Solr to return MoreLikeThis results.
`mlt.count`	Specifies the number of similar documents to be returned for each result. The default value is 5.

7.10.3 Parameters for the MoreLikeThis Request Handler

The table below summarizes parameters accessible through the MoreLikeThisHandler, which was introduced in Solr 1.3. It supports faceting, paging, and filtering using common query parameters.

Parameter	Description
`mlt.match.include`	Specifies whether or not the response should include the matched document. If set to false, the response will look like a normal select response.
`mlt.match.offset`	Specifies an offset into the main query search results to locate the document on which the MoreLikeThis query should operate. By default, the query operates on the first result for the `q` parameter.
`mlt.interestingTerms`	Controls how the MoreLikeThis component presents the "interesting" terms (the top TF/IDF terms) for the query. Supports three settings. The setting `list` lists the terms. The setting `none` lists no terms. The setting `details` lists the terms along with the boost value used for each term. Unless `mlt.boost=true` all terms will have boost=1.0.

7.11 Faceting

As described in Section 7.1, faceting is the arrangement of search results into categories based on indexed terms. Searchers are presented with the indexed terms, along with numerical counts of how many matching documents were found were each term. Faceting makes it easy for users to explore search results, narrowing in on exactly the results they're looking for.

The table below summarizes the general parameters for controlling faceting.

Parameter	Description
`facet`	If set to true, enables faceting.
`facet.query`	Specifies a Lucene query to generate a facet count.

These parameters are described in the sections below.

7.11.1 facet

If set to “true,” this parameter enables facet counts in the query response. If set to “false” to a blank or missing value, this parameter disables faceting. None of the other parameters listed below will have any effect unless this parameter is set to “true.”

The default value is blank.

7.11.2 facet.query : Arbitrary Query Faceting

This parameter allows you to specify an arbitrary query in the Lucene default syntax to generate a facet count. By default, Solr's faceting feature automatically determines the unique terms for a field and returns a count for each of those terms. Using `facet.query`, you can override this default behavior and select exactly which terms or expressions you would like to see counted. In a typical implementation of faceting, you will specify a number of `facet.query` parameters. This parameter can be particularly useful for numeric-range-based facets or prefix-based facets.

You can set the `facet.query` parameter multiple times to indicate that multiple queries should be used as separate facet constraints.

To use facet queries in a syntax other than the default syntax, prefix the facet query with the name of the query notation. For example, to use the hypothetical `myfunc` query parser, you could set the `facet.query` parameter like so:

```
facet.query={!myfunc}name~fred
```

7.11.3 Field-Value Faceting Parameters

Several parameters can be used to trigger faceting based on the indexed terms in a field.

When using this parameter, it is important to remember that “term” is a very specific concept in Lucene: it relates to the literal field/value pairs that are indexed after any analysis occurs. For text fields that include stemming, lowercasing, or word splitting, the resulting terms may not be what you expect. If you want Solr to perform both analysis (for searching) and faceting on the full literal strings, use the `copyField`

directive in the `schema.xml` file to create two versions of the field: one Text and one String. Make sure both are `indexed="true"`. (For more information about the `copyField` directive, see Chapter 4.)

The table below summarizes Solr's field value faceting parameters.

Parameter	Description
`facet.field`	Identifies a field to be treated as a facet.
`facet.prefix`	Limits the terms used for faceting to those that begin with the specified prefix.
`facet.sort`	Controls how faceted results are sorted.
`facet.limit`	Controls how many constraints should be returned for each facet.
`facet.offset`	Specifies an offset into the facet results at which to begin displaying facets.
`facet.mincount`	Specifies the minimum counts required for a facet field to be included in the response.
`facet.missing`	Controls whether Solr should compute a count of all matching results which have no value for the field, in addition to the Term-based constraints of a facet field.
`facet.method`	Selects the algorithm or method Solr should use when faceting a field.
`facet.enum.cache.minDF`	Specifies the minimum document frequency (the number of documents matching a term) for which the filterCache should be used when determining the constraint count for that term.

These parameters are described in the sections below.

7.11.3.1 The facet.field Parameter

The `facet.field` parameter identifies a field that should be treated as a facet. It iterates over each Term in the field and generate a facet count using that Term as the constraint. This parameter can be specified multiple times in a query to select multiple facet fields.

NOTE: If you do not set this parameter to at least one field in the schema, none of the other parameters described in this section will have any effect.

7.11.3.2 The facet.prefix Parameter

The `facet.prefix` parameter limits the terms on which to facet to those starting with the given string prefix.

This parameter can be specified on a per-field basis.

7.11.3.3 The facet.sort Parameter

This parameter determines the ordering of the facet field constraints.

NOTE: The `true/false` values for this parameter, which were supported in earlier Solr releases, are deprecated in Solr 1.4.

facet.sort Setting	Results
`count`	Sort the constraints by count (highest count first).
`index`	Return the constraints sorted in their index order (lexicographic by indexed term). For terms in the ASCII range, this will be alphabetically sorted.

The default is `count` if `facet.limit` is greater than 0, otherwise, the default is `index`.

This parameter can be specified on a per-field basis.

7.11.3.4 The facet.limit Parameter

This parameter specifies the maximum number of constraint counts that should be returned for the facet fields. A negative value means that Solr will return unlimited number of constraint counts.

The default value is 100.

This parameter can be specified on a per-field basis to apply a distinct limit to each field.

7.11.3.5 The facet.offset Parameter

The `facet.offset` parameter indicates an offset into the list of constraints to allow paging.

The default value is 0.

This parameter can be specified on a per-field basis.

7.11.3.6 The facet.mincount Parameter

The `facet.mincount` parameter specifies the minimum counts required for a facet field to be included in the response. (If a field's counts are below the minimum, the field's facet is not returned.)

The default value is 0.

This parameter can be specified on a per-field basis.

7.11.3.7 The facet.missing Parameter

If set to true, this parameter indicates that, in addition to the Term-based constraints of a facet field, a count of all matching results which have no value for the field should be computed.

The default value is false.

This parameter can be specified on a per-field basis.

7.11.3.8 The facet.method Parameter

The `facet.method` parameter selects the type of algorithm or method Solr should use when faceting a field.

facet.method Setting	Results
`enum`	Enumerates all terms in a field, calculating the set intersection of documents that match the term with documents that match the query. This method is recommended for faceting multi-valued fields that have only a few distinct values. The average number of values per document does not matter. For example, faceting on a field with U.S. States e.g. `Alabama, Alaska, … Wyoming` would lead to fifty cached filters which would be used over and over again. The filterCache should be large enough to hold all the cached filters.
`fc`	Calculates facet counts by iterating over documents that match the query and summing the terms that appear in each document. This is currently implemented using an UnInvertedField cache if the field either is multi-valued or is tokenized (according to FieldType.isTokened()). Each document is looked up in the cache to see what terms/values it contains, and a tally is incremented for each value. This method is excellent for situations where the number of indexed values for the field is high, but the number of values per document is low. For multi-valued fields, a hybrid approach is used that uses term filters from the filterCache for terms that match many documents. (The letters `fc` standard for field cache.)

The default value is `fc` (except for BoolField) since it tends to use less memory and is faster when a field has many unique terms in the index.

This parameter can be specified on a per-field basis.

7.11.3.9 The facet.enum.cache.minDf Parameter

This parameter indicates the minimum document frequency (the number of documents matching a term) for which the filterCache should be used when determining the constraint count for that term. This is only used with the `facet.method=enum` method of faceting.

A value greater than zero decreases the filterCache's memory usage, but increases the time required for the query to be processed. If you are faceting on a field with a very large number of terms, and you wish to decrease memory usage, try setting this parameter to a value between 25 and 50, and run a few tests. Then, optimize the parameter setting as necessary.

The default value is 0, causing the filterCache to be used for all terms in the field.

This parameter can be specified on a per-field basis.

7.11.4 Date Faceting Parameters

Several parameters can be used to trigger faceting based on date ranges computed using simple expressions. When using date faceting, the following parameters are mandatory:

- `facet.date`
- `facet.date.start`
- `facet.date.end`
- `facet.date.gap`

The table below summarizes Solr's date faceting parameters.

Parameter	Description
`facet.date`	Specifies which fields should be treated as date facets.
`facet.date.start`	Species the starting date in a date range.
`facet.date.end`	Specifies the ending date in a date range.
`facet.date.gap`	Specifies the interval to be used to define the date range.
`facet.date.hardend`	Controls how Solr handles date ranges that do not divide evenly.
`facet.date.other`	Controls how Solr calculates other counts related to date faceting.

These parameters are described in detail in the following sections.

7.11.4.1 The facet.date Parameter

This parameter specifies the names of fields (of type DateField) which should be treated as date facets.

This parameter can be specified multiple times to indicate multiple date facet fields.

7.11.4.2 The facet.date.start Parameter

This parameter specifies the lower bound for the first date range for all Date Faceting on this field. This should be a single date expression which may use the DateMathParser syntax.

This parameter can be specified on a per-field basis.

7.11.4.3 The facet.date.end Parameter

The minimum upper bound for the last date range for all Date Faceting on this field (see the description of the `facet.date.hardend` parameter for an explanation of why the actual end value of the range may be greater). This should be a single date expression which may use the DateMathParser syntax.

This parameter can be specified on a per-field basis.

7.11.4.4 The facet.date.gap Parameter

This parameter specifies the size of each date range expressed as an interval to be added to the lower bound using the DateMathParser syntax. For example:

```
facet.date.gap=%2B1DAY (+1DAY)
```

This parameter can be specified on a per-field basis.

7.11.4.5 The facet.date.hardend Parameter

This parameter is a Boolean parameter instructing Solr what to do in the event that `facet.date.gap` does not divide evenly between `facet.date.start` and `facet.date.end`. If this parameter is true, the last date range constraint will have an upper bound of `facet.date.end`. If false, the last date

range will have the smallest possible upper bound greater then `facet.date.end`, such that the range is exactly `facet.date.gap` wide.

The default is false.

This parameter can be specified on a per-field basis.

7.11.4.6 The facet.date.other Parameter

This parameter indicates that, in addition to the counts for each date range constraint between `facet.date.start` and `facet.date.end`, counts should also be computed for:

facet.date.other Setting	Result
`before`	Computes counts for all records with field values lower then lower bound of the first range.
`after`	Computes counts for all records with field values greater then the upper bound of the last range.
`between`	Computes counts all records with field values between the start and end bounds of all ranges.
`none`	Computes counts for none of these records.
`all`	Computes counts for all of these records; a shortcut for `before`, `after`, and `between`.

This parameter can be specified on a per-field basis.

In addition to the `all` option, this parameter can be specified multiple times to indicate multiple choices, but `none` will override all other options.

7.11.5 LocalParams for Faceting

The LocalParams syntax provides a method of adding meta-data to other parameter values, much like XML attributes.

7.11.5.1 Tagging and Excluding Filters

You can tag specific filters and exclude those filters when faceting. This is useful when doing multi-select faceting.

Consider the following example query with faceting:

```
q=mainquery&fq=status:public&fq=doctype:pdf&facet=on&facet.field=doctype
```

Because everything is already constrained by the filter `doctype:pdf`, the `facet.field=doctype` facet command is currently redundant and will return 0 counts for everything except `doctype:pdf`.

To implement a multi-select facet for doctype, a GUI may want to still display the other doctype values and their associated counts, as if the `doctype:pdf` constraint had not yet been applied. For example:

```
=== Document Type ===
  [ ] Word (42)
  [x] PDF  (96)
  [ ] Excel(11)
  [ ] HTML (63)
```

To return counts for doctype values that are currently not selected, tag filters that directly constrain doctype, and exclude those filters when faceting on doctype.

```
q=mainquery&fq=status:public&fq={!tag=dt}doctype:pdf&facet=on&facet.field={!
ex=dt}doctype
```

Filter exclusion is supported for all types of facets. Both the `tag` and `ex` local parameters may specify multiple values by separating them with commas.

7.11.5.2 key: Changing the Output Key

To change the output key for a faceting command, specify a new name with the `key` local parameter. For example:

```
facet.field={!ex=dt key=mylabel}doctype
```

The parameter setting above causes the results to be returned under the key "mylabel" rather than "doctype" in the response. This can be helpful when faceting on the same field multiple times with different exclusions.

7.12 Spell Checking

The SpellCheck component accepts the parameters described in the table below.

Parameter	Description
`spellcheck`	Turns on or off SpellCheck suggestions for the request. If true, then spelling suggestions will be generated.
`spellcheck.q` `q`	Selects the query to be spellchecked.
`spellcheck.build`	Instructs Solr to build a dictionary for use in spellchecking.
`spellcheck.collate`	Causes Solr to build a new query based on the best suggestion for each term in the submitted query.
`spellcheck.count`	Specifies the maximum number of spelling suggestions to be returned.
`spellcheck.dictionary`	Specifies the dictionary that should be used for spellchecking.
`spellcheck.extendedResults`	Causes Solr to return additional information about spellcheck results, such as the frequency of each original term in the index (origFreq) as well as the frequency of each suggestion in the index (frequency). Note that this result format differs from the non-extended one as the returned suggestion for a word is actually an array of lists, where each list holds the suggested term and its frequency.
`spellcheck.onlyMorePopular`	Limits spellcheck responses to queries that are more popular than the original query.
`spellcheck.reload`	Reloads the spellchecker.

These parameters are described in detail in the sections below.

7.12.1 The spellcheck Parameter

This parameter turns on or off SpellCheck suggestions for the request. If true, then spelling suggestions will be generated.

7.12.2 The q OR spellcheck.q Parameter

Specifies the query to spellcheck. If `spellcheck.q` is defined, then it is used; otherwise the original input query is used. The `spellcheck.q` parameter is intended to be the original query, minus any extra markup like field names, boosts, etc. If the `q` parameter is specified, then the SpellingQueryConverter class is used to parse it into tokens; otherwise the WhitespaceTokenizer is used. The choice of which one to use is up to the application. Essentially, if you have a spelling "ready" version in your application, then it is probably better to use `spellcheck.q`. Otherwise, if you just want Solr to do the job, use the `q` parameter.

NOTE: The SpellingQueryConverter class does not deal properly with non-ASCII characters. In this case, you have either to use `spellcheck.q`, or implement your own QueryConverter.

7.12.3 The spellcheck.build Parameter

If set to true, this parameter creates the dictionary that the SolrSpellChecker will use for spell-checking. In a typical search application, you will need to build the dictionary before using the SolrSpellChecker. However, it's not always necessary to build a dictionary first. For example, you can configure the spellchecker to use a dictionary that already exists.

7.12.4 The spellcheck.reload Parameter

If set to true, this parameter reloads the spellchecker. The results depend on the implementation of `SolrSpellChecker.reload()`. In a typical implementation, reloading the spellchecker means reloading the dictionary.

7.12.5 The spellcheck.count Parameter

This parameter specifies the maximum number of suggestions that the spellchecker should return for a term. If this parameter isn't set, the value defaults to 1. If the parameter is set but not assigned a number, the value defaults to 5. If the parameter is set to a positive integer, that number becomes the maximum number of suggestions returned by the spellchecker.

7.12.6 The spellcheck.onlyMorePopular Parameter

This parameter causes Solr to return suggestions that result in more hits for the query than the existing query.

7.12.7 The spellcheck.extendedResults Parameter

This parameter causes to Solr to include additional information about the suggestion, such as the frequency in the index.

7.12.8 The spellcheck.collate Parameter

This parameter directs Solr to take the best suggestion for each token (if it exists) and construct a new query from the suggestions. For example, if the input query was “jawa class lording” and the best suggestion for “jawa” was “java” and “lording” was “loading”, then the resulting collation would be “java class loading”.

NOTE: This only returns a query to be used. It does not actually run the suggested query.

7.12.9 The spellcheck.dictionary Parameter

This parameter causes Solr to use the dictionary named in the parameter's argument. The default setting is "default". This parameter can be used to invoke a specific spellchecker on a per request basis.

7.12.10 Example

This example shows the results of a simple query that defines a query using the `spellcheck.q` parameter. The query also includes a `spellcheck.build=true` parameter, which is needs to be

called only once in order to build the index. `spellcheck.build` should not be specified with for each request.

```
http://localhost:8983/solr/spellCheckCompRH?q=*:*&spellcheck.q=hell
%20ultrashar&spellcheck=true&spellcheck.build=true
```

Results:

```
<lst name="spellcheck">
        <lst name="suggestions">
                <lst name="hell">
                        <int name="numFound">1</int>
                        <int name="startOffset">0</int>
                        <int name="endOffset">4</int>
                        <arr name="suggestion">
                                <str>dell</str>
                        </arr>
                </lst>
                <lst name="ultrashar">
                        <int name="numFound">1</int>
                        <int name="startOffset">5</int>
                        <int name="endOffset">14</int>
                        <arr name="suggestion">
                                <str>ultrasharp</str>
                        </arr>
                </lst>
        </lst>
</lst>
```

7.13 The Terms Component

7.13.1 Overview

The Terms Component provides access to the indexed terms in a field and the number of documents that match each term. This can be useful for building an auto-suggest feature or any other feature that operates at the term level instead of the search or document level. Retrieving terms in index order is very fast since the implementation directly uses Lucene's TermEnum to iterate over the term dictionary.

In a sense, this component provides fast field-faceting over the whole index, not restricted by the base query or any filters. The document frequencies returned are the number of documents that match the term, including any documents that have been marked for deletion but not yet removed from the index.

To use the TermsComponent, users can pass in a variety of options to control what terms are returned. The supported parameters are available in the class:

```
http://lucene.apache.org/solr/api/org/apache/solr/common/params/TermsParams.html
```

These parameters are:

Parameter	Description	Syntax
`terms`	If set to true, terms on the Terms Component. By default, the Terms Component is turned off.	`terms={true|false}`
`terms.fl`	Specifies the field from which to retrieve terms.	`terms.fl=`*`field`*
`terms.lower`	Specifies the term at which to start. If not specified, the empty string is used, causing Solr to start at the beginning of the field.	`terms.lower=`*`term`*
`terms.lower.incl`	If set to true, includes the lower-bound term in the result set. By default, this parameter is set to true.	`terms.lower.incl={true|false}`
`terms.mincount`	Specifies the minimum document frequency to return in order for a term to be included in a query response. Results are inclusive of the mincount (i.e., >= mincount). This parameter is optional.	`terms.mincount=`*`integer`*

Parameter	Description	Syntax
`terms.maxcount`	Specifies the maximum document frequency a term must have in order to be included in a query response. The default setting is -1, which sets no upper bound. Results are inclusive of the maxcount (i.e., <= maxcount). This parameter is optional.	`terms.maxcount=`*`integer`*
`terms.prefix`	Restricts matches to terms that begin with the specified string.	`terms.prefix={string}`
`terms.limit`	Specifies the maximum number of terms to return. The default is 10. If the limit is set to a number less than 0, then no maximum limit is enforced.	`terms.limit=`*`integer`*
`terms.upper`	Specifies the term to stop at. Any application using the Terms component must set either `terms.limit` or `terms.upper`.	`terms.upper=`*`upper_term`*
`terms.upper.incl`	If set to true, includes the upper bound term in the result set. The default is false.	`terms.upper.incl={true|false}`
`terms.raw`	If set to true, returns the raw characters of the indexed term, regardless of whether it is human-readable. For instance, the indexed form of numeric numbers is not human-readable. The default is false.	`terms.raw={true|false}`

The output is a list of the terms and their document frequency values.

7.13.2 Examples

The following examples use the sample Solr configuration located in the `<Solr>/example` directory.

The query below requests the first ten terms in the `name` field.

```
http://localhost:8983/solr/terms?terms.fl=name
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>

<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">1</int>
</lst>
<lst name="terms">
 <lst name="name">
  <int name="0">5</int>

  <int name="1">15</int>
  <int name="11">5</int>
  <int name="120">5</int>
  <int name="133">5</int>
  <int name="184">15</int>
  <int name="19">5</int>
  <int name="1900">5</int>
  <int name="2">15</int>
  <int name="20">5</int>
 </lst>
</lst>
</response>
```

The query below requests the first ten terms in the `name` field, beginning with the first term that begins with the letter a.

```
http://localhost:8983/solr/terms?terms.fl=name&terms.lower=a
```

Results:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">2</int>
</lst>
<lst name="terms">
 <lst name="name">
  <int name="a">8</int>
```

```
  <int name="adata">5</int>
  <int name="all">5</int>
  <int name="allinon">5</int>
  <int name="amber">1</int>
  <int name="appl">5</int>
  <int name="asus">5</int>
  <int name="ata">5</int>
  <int name="ati">5</int>
  <int name="b">5</int>
 </lst>
</lst>
</response>
```

7.13.3 Using the Terms Component for an Auto-Suggest Feature

Internet search engines such as Google now feature auto-suggest features, in which the search engine offers and then modifies a list of suggested search terms based on the characters that the user has typed so far. You can use the Terms component in Solr to build a similar feature for your own search application. Simply submit a query specifying whatever characters the user has typed so far as a prefix. For example, if the user has typed “at”, the search engine's interface would submit the following query:

```
http://localhost:8983/solr/terms?terms.fl=name&terms.prefix=at
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>

<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">120</int>
</lst>
<lst name="terms">
 <lst name="name">
  <int name="ata">5</int>
  <int name="ati">5</int>
 </lst>
</lst>
</response>
```

You can use the parameter `omitHeader=true` to omit the response header from the query response, like so:

```
http://localhost:8983/solr/terms?
terms.fl=name&terms.prefix=at&indent=true&wt=json&omitHeader=true
```

Result:

```
{
 "terms":[
  "name",[
        "ata",1,
        "ati",1]]}
```

7.14 The TermVector Component

The Term Vector Component (TVC) is a SearchComponent designed to return information about documents that is stored when setting the termVector attribute on a field:

```
<field name="features" type="text" indexed="true" stored="true"
multiValued="true" termVectors="true" termPositions="true"
termOffsets="true"/>
```

For each document, the TVC can return, the term vector, the term frequency, inverse document frequency, position and offset information. As with most components, there are a number of options that are outlined in the samples below. All examples are based on using the Solr example.

7.14.1 Enabling the TVC

7.14.1.1 Changes required in solrconfig.xml

To enable the TermVectorComponent, you need to configure a searchComponent element in your `solrconfig.xml` file, like so::

```
<searchComponent name="tvComponent"
class="org.apache.solr.handler.component.TermVectorComponent"/>
```

A RequestHandler configuration using this component could look like this:

```
<requestHandler name="tvrh"
class="org.apache.solr.handler.component.SearchHandler">
        <lst name="defaults">
                <bool name="tv">true</bool>
        </lst>
        <arr name="last-components">
```

```
                <str>tvComponent</str>
        </arr>
</requestHandler>
```

7.14.1.2 Invoking the TermVector Component

The example below shows an invocation of this component:

```
http://localhost:8983/solr/select/?q=*
%3A*&version=2.2&start=0&rows=10&indent=on&qt=tvrh&tv=true
```

In the example, the component is associated with a request handler named `tvrh`, but you can associate it with any RequestHandler. To turn on the component for a request, add the `tv=true` parameter (or add it to your RequestHandler defaults configuration).

Example output: See TermVectorComponentExampleEnabled.

7.14.2 Optional Parameters

The example below shows optional parameters for this component:

```
http://localhost:8983/solr/select/?q=*
%3A*&version=2.2&start=0&rows=10&indent=on&qt=tvrh&tv=true&tv.tf=true&tv.d
f=true&tv.positions&tv.offsets=true
```

Boolean Parameters	Description
`tv.all`	A shortcut that invokes all the parameters listed below.
`tv.df`	Returns the Document Frequency (DF) of the term in the collection. This can be computationally expensive.
`tv.offsets`	Returns offset information for each term in the document.
`tv.positions`	Returns position information.
`tv.tf`	Returns document term frequency info per term in the document.
`tv.tf_idf`	Calculates TF*IDF for each term. Requires the parameters `tv.tf` and `tv.df` to be "true". This can be computationally

Boolean Parameters	Description
	expensive. (The results are not shown in example output)

To learn more about TermVector component output, see the Wiki page: TermVectorComponentExampleOptions.

For schema requirements, see the Wiki page: FieldOptionsByUseCase.

The TermVector component also accepts these optional parameters:

Parameters	Description
`tv.docIds`	Returns term vectors for the specified list of Lucene document IDs (not the Solr Unique Key).
`tv.fl`	Returns term vectors for the specified list of fields. If not specified, the `fl` parameter is used.

7.14.3 SolrJ and the TermVector Component

Neither the SolrQuery class nor the QueryResponse class offer specific method calls to set TermVectorComponent parameters or get the "termVectors" output. However, there is a patch for it: SOLR-949.

7.15 The Stats Component

Introduced in Solr 1.4, the Stats component returns simple statistics for numeric fields within the DocSet.

7.15.1 Stats Component Parameters

The Stats Component accepts the following parameters:

Parameter	Description
stats	If true, then invokes the Stats component.
stats.field	Specifies a field for which statistics should be generated. This parameter may be invoked multiple times in a query in order to request statistics on multiple fields. (See the example below.)
stats.facet	Returns sub-results for values within the specified facet.

7.15.2 Example

The query below:

```
http://localhost:8983/solr/select?
q=*:*&stats=true&stats.field=price&stats.field=popularity&rows=0&indent=tr
ue
```

Would produce the following results:

```
<lst name="stats">
 <lst name="stats_fields">
  <lst name="price">
    <double name="min">0.0</double>
    <double name="max">2199.0</double>
    <double name="sum">5251.2699999999995</double>
    <long name="count">15</long>
    <long name="missing">11</long>
    <double name="sumOfSquares">6038619.160300001</double>
    <double name="mean">350.08466666666664</double>
    <double name="stddev">547.737557906113</double>
  </lst>
  <lst name="popularity">
    <double name="min">0.0</double>
    <double name="max">10.0</double>
    <double name="sum">90.0</double>
    <long name="count">26</long>
    <long name="missing">0</long>
    <double name="sumOfSquares">628.0</double>
    <double name="mean">3.461538461538461 7</double>
    <double name="stddev">3.5578731762756157</double>
  </lst>
 </lst>
</lst>
```

Here are the same results with faceting requested for the field `inStock`, using the parameter `&stats.facet=inStock`.

```
<lst name="stats">
 <lst name="stats_fields">
  <lst name="price">
  <double name="min">0.0</double>
  <double name="max">2199.0</double>
  <double name="sum">5251.2699999999995</double>
  <long name="count">15</long>
  <long name="missing">11</long>
  <double name="sumOfSquares">6038619.160300001</double>
  <double name="mean">350.0846666666664</double>
  <double name="stddev">547.737557906113</double>
  <lst name="facets">
   <lst name="inStock">
    <lst name="false">
      <double name="min">11.5</double>
      <double name="max">649.99</double>
      <double name="sum">1161.39</double>
      <long name="count">4</long>
      <long name="missing">0</long>
      <double name="sumOfSquares">653369.2551</double>
      <double name="mean">290.3475</double>
      <double name="stddev">324.63444676281654</double>
    </lst>
    <lst name="true">
      <double name="min">0.0</double>
      <double name="max">2199.0</double>
      <double name="sum">4089.879999999999</double>
      <long name="count">11</long>
      <long name="missing">0</long>
      <double name="sumOfSquares">5385249.905200001</double>
      <double name="mean">371.8072727272727</double>
      <double name="stddev">621.6592938755265</double>
    </lst>
   </lst>
  </lst>
 </lst>
</lst>
```

7.15.3 The Stats Component and Faceting

The facet field can be selectively applied. That is if you want stats on field "A" and "B", you can facet a on "X" and B on "Y" using the parameters:

```
&stats.field=A&f.A.stats.facet=X&stats.field=B&f.B.stats.facet=Y
```

NOTE: All facet results are returned, be careful what fields you ask for!

Multi-valued fields and facets may be slow.

Multi-value fields rely on UnInvertedField.java for implementation. This is like a FieldCache, so be aware of your memory footprint.

7.15.4 Statistics Returned

The table below describes the statistics returned by the Stats component.

Name	Description
`min`	The minimum value in the field.
`max`	The maximum value in the field.
`sum`	The sum of all values in the field.
`count`	The number of non-null values in the field.
`missing`	The number of null values in the field.
`sumOfSquares`	Sum of all values squared (useful for stddev).
`mean`	The average `(v1 + v2 …. + vN)/N`
`stddev`	Standard deviation, measuring how widely spread the values in the data set are.

7.16 Response Writers

A Response Writer generates the formatted response of a search. Solr supports a variety of Response Writers to ensure that query responses can be parsed by the appropriate language or application.

The `wt` parameter selects the Response Writer to be used. The table below lists the most common settings for the `wt` parameter.

wt Parameter Setting	Response Writer Selected
`json`	JSONResponseWriter
`php`	PHPResponseWriter
`phps`	PHPSerializedResponseWriter
`python`	PythonResponseWriter
`ruby`	RubyResponseWriter
`xml`	XMLResponseWriter
`xslt`	XSLTResponseWriter

7.16.1 The Standard XML Response Writer

The XML Response Writer is the most general purpose and reusable Response Writer currently included with Solr. It is the format used in most discussions and documentation about the response of Solr queries.

Note that the XsltResponseWriter can be used to convert the XML produced by this writer to other vocabularies or text-based formats.

The behavior of the XML Response Writer can be driven by the following query parameters.

7.16.1.1 The version Parameter

The version parameter determines the XML protocol used in the response. Clients are strongly encouraged to *always* specify the protocol version, so as to ensure that the format of the response they receive does not change unexpectedly if/when the Solr server is upgraded.

XML Version	Notes
2.0	An `<arr>` tag was used for multiValued fields only if there was more then one value.
2.1	An `<arr>` tag is used for multiValued fields even if there is only one value.
2.2	The format of the responseHeader changed to use the same `<lst>` structure as the rest of the response.

The default value is the latest supported.

7.16.1.2 The stylesheet Parameter

The `stylesheet` parameter can be used to direct Solr to include a `<?xml-stylesheet type="text/xsl" href="..."?>` declaration in the XML response it returns.

The default behavior is not to return any stylesheet declaration at all.

NOTE: Use of the `stylesheet` parameter is discouraged, as there is currently no way to specify external stylesheets, and no stylesheets are provided in the Solr distributions. This is a legacy parameter, which may be developed further in a future release.

7.16.1.3 The indent Parameter

If the `indent` parameter is used, and has a non-blank value, then Solr will make some attempts at indenting its XML response to make it more readable by humans.

The default behavior is not to indent.

7.16.2 The XSLT Response Writer

The XSLT Response Writer applies an XML stylesheet to output. It can be used for tasks such as formatting results for an RSS feed.

7.16.2.1 Parameters

The XSLT Response Writer accepts one parameter: the tr parameter, which identifies the XML transformation to use. The transformation must be found in the Solr `conf/xslt` directory.

The Content-Type of the response is set according to the `<xsl:output>` statement in the XSLT transform, for example:

```
<xsl:output media-type="text/html"/>
```

7.16.2.2 Configuration

The example below, from the default `solrconfig.xml` file, shows how the XSLT Response Writer is configured.

```
<!--
  Changes to XSLT transforms are taken into account
  every xsltCacheLifetimeSeconds at most.
-->
<queryResponseWriter
  name="xslt"
  class="org.apache.solr.request.XSLTResponseWriter"
>
  <int name="xsltCacheLifetimeSeconds">5</int>
</queryResponseWriter>
```

A value of 5 for `xsltCacheLifetimeSeconds` is good for development, to see XSLT changes quickly. For production you probably want a much higher value.

7.16.3 JsonResponseWriter

A very commonly used Response Writer is the JsonResponseWriter, which formats output in JavaScript Object Notation (JSON), a lightweight data interchange format specified in specified in RFC 4627. Setting the `wt` parameter to `json` invokes this Response Writer.

7.16.4 PythonResponseWriter

Solr has an optional Python response format that extends its JSON output in the following ways to allow the response to be safely evaluated by Python's interpreter:

- true and false changed to True and False
- Python unicode strings are used where needed
- ASCII output (with unicode escapes) is used for less error-prone interoperability
- newlines are escaped
- null changed to None

7.16.5 PHPResponseWriter and PHPSerializedResponseWriter

Solr has a PHP response format that outputs an array (as PHP code) which can be evaluated. Setting the `wt` parameter to `php` invokes the PHP Response Writer.

Example usage:

```
$code = file_get_contents('http://localhost:8983/solr/select?
q=iPod&wt=php');
eval("$result = " . $code . ";");
print_r($result);
```

Solr also includes a PHPSerializedResponseWriter that formats output in a serialized array. Setting the `wt` parameter to `phps` invokes the PHP Serialized Response Writer.

Example usage:

```
$serializedResult = file_get_contents('http://localhost:8983/solr/select?
q=iPod&wt=phps');
$result = unserialize($serializedResult);
print_r($result);
```

Before you use either the PHP or Serialized PHP Response Writer, you may first need to un-comment these two lines in `solrconfig.xml`:

```
<queryResponseWriter name="php"
class="org.apache.solr.request.PHPResponseWriter"/>
<queryResponseWriter name="phps"
class="org.apache.solr.request.PHPSerializedResponseWriter"/>
```

7.16.6 RubyResponseWriter

Solr has an optional Ruby response format that extends its JSON output in the following ways to allow the response to be safely evaluated by Ruby's interpreter:

- Ruby's single quoted strings are used to prevent possible string exploits.
- \ and ' are the only two characters escaped.
- Unicode escapes are not used. Data is written as raw UTF-8.
- nil used for null.
- => is used as the key/value separator in maps.

Here is a simple example of how one may query Solr using the Ruby response format:

```
require 'net/http'

h = Net::HTTP.new('localhost', 8983)
hresp, data = h.get('/solr/select?q=iPod&wt=ruby', nil)
rsp = eval(data)

puts 'number of matches = ' + rsp['response']['numFound'].to_s
#print out the name field for each returned document
rsp['response']['docs'].each { |doc| puts 'name field = ' + doc['name'] }
```

7.16.7 BinaryResponseWriter

Solr also includes a Response Writer that outputs binary format for use with a Java client. See Chapter 11 for more details.

7.17 Summary

Solr offers a flexible, highly configurable architecture for search. Requests, including search queries, are passed to requests handlers. To process a query, a request handler calls a query parser. The default query parser is the DisMax query parser. DisMax supports a simple interface like that used on popular public search engines such as Google. Another query parser, the standard query parser, offers more parameters for narrowly specifying search parameters. Both the DisMax query parser and the standard query parser support special components that provide features such as highlighting, faceting, and auto-suggesting terms.

After the request handler parses and executes a query, it sends the response to a response writer for formatting. Solr provides a variety of response writers, including the XML Response Writer, the JSON Response Writer, and the Binary Response Writer.

Solr provides search applications a great deal of control over the specification of queries and the formatting and presentation of responses. By taking advantage of Solr's many search features, application designers can ensure that search responses are as relevant as possible to users.

This page is intentionally left blank.

8 The Well Configured Solr Instance

This chapter tells you how to fine-tune your Solr instance for optimum performance. The chapter covers these topics:

- configuring the `solrconfig.xml` file, one of the most important files for configuring key
- running multiple SolrCores within a single Solr instance
- configuring Lucene IndexWriters
- configuring the HTTP Request Dispatcher
- configuring the JVM

Some of these options control Solr, while others affect the underlying Lucene engine.

> **NOTE:** The focus of this chapter is on configuring a single Solr instance. To scale a Solr implementation, either through sharding or replication, please see Chapter 10.

8.1 Configuring solrconfig.xml

The solrconfig.xml file is the configuration file with the most parameters affecting Solr itself. The file comprises a series of XML statements that set configuration values. In solrconfig.xml, you configure important features such as:

- request handlers

- listeners (processes that "listen" for particular query-related events; listeners can be used to trigger the execution of special code, such as invoking some common queries to warm-up caches)
- the Request Dispatcher for managing HTTP communications
- the Admin Web interface
- parameters related to replication and duplication (these parameters are covered in detail in Chapter 10)

The `solrconfig.xml` file is found in the `solr/conf/` directory.

8.1.1 Specifying a Location for Index Data with the dataDir Parameter

By default, Solr stores its index data in a directory called `/data` under the Solr home. If you would like to specify a different directory for storing index data, use the `<dataDir>` parameter in the solrconfig.xml file. You can specify another directory either with a full pathname or a pathname relative to the current working directory of the servlet container. For example:

```
<dataDir>/var/data/solr/</dataDir>
```

If you are using replication to replicate the Solr index (as described in Chapter 10), then the `<dataDir>` directory should correspond to the index directory used in the replication configuration.

8.1.2 Configuring the Lucene IndexWriter(s)

```
<indexDefaults>
    ...
</indexDefaults>
```

The settings in this section are specified in the `<indexDefaults>` element in `solrconfig.xml` and control the behavior of Lucene index writers.

8.1.2.1 UseCompoundFile

```
<useCompoundFile>false</useCompoundFile>
```

Setting `<useCompoundFile>` to "`true`" combines the various files on disk that make up an index into a single file. On systems where the number of open files allowed per process is limited, setting this to "`true`" may avoid hitting that limit. (The open files limit might also be tunable for your OS with the Linux/Unix `ulimit` command, or something similar for other operating systems.)

Updating a compound index may incur a minor performance hit for various reasons, depending on the runtime environment. For example, filesystem buffers are typically associated with open file descriptors, which may limit the total cache space available to each index.

This setting may also affect how much data needs to be transferred during index replication operations.

This setting is "`false`" in the `solrconfig.xml` file for the example application. Since Lucene 1.4, the default in the code is "`true`" ,if not explicitly specified.

8.1.2.2 mergeFactor

```
<mergeFactor>10</mergeFactor>
```

The `mergeFactor` 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 that segment fills up (see `maxBufferedDocs` and `ramBufferSizeMB` in the next section), a new segment is created and subsequent updates are placed there.

If creating a new segment would cause the number of lowest-level segments to exceed the "`mergeFactor`" value, then all those segments are merged together to form a single large segment. Thus, if the merge factor is ten, each merge results in the creation of a single segment that is roughly ten times larger than each of its ten constituents. When there are "`mergeFactor`" settings for these larger segments, then they in turn are merged into an even larger single segment. This process can continue indefinitely.

Choosing the best merge factor is generally a trade-off of indexing speed vs. searching speed. Having fewer segments in the index generally accelerates searches, because there are fewer places to look. It also can also result in fewer physical files on disk. But to keep the number of segments low, merges will occur more often, which can add load to the system and slow down updates to the index.

Conversely, keeping more segments can accelerate indexing, because merges happen less often – making an update is less likely to trigger a merge. But searches become more computationally expensive and will likely be slower, because search terms must be looked up in more index segments. Faster index updates also means shorter commit turnaround times, which means more timely search results.

The default value in the example `solrconfig.xml` is `10`, which is a reasonable starting point.

8.1.2.3 Other Indexing Settings

```
<maxBufferedDocs>1000</maxBufferedDocs>
<ramBufferSizeMB>32</ramBufferSizeMB>
<maxMergeDocs>2147483647</maxMergeDocs>
<maxFieldLength>10000</maxFieldLength>
```

These settings can affect how or when updates are made to an index.

Setting	Description
`maxBufferedDocs`	Once this many document updates have been buffered in memory, they are flushed to disk and added to the current index segment. If the segment fills up, a new one may be created, or a merge may be started. The default Solr configuration leaves this value undefined.
`ramBufferSizeMB`	Once accumulated document updates exceed this much memory space (specified in megabytes), then the pending updates are flushed. This can also create new segments or trigger a merge. Using this setting is generally preferable to `maxBufferedDocs`. If both `maxBufferedDocs` and `ramBufferSizeMB` are set in `solrconfig.xml`, then a flush will occur when either limit is reached.
`maxMergeDocs`	This sets the maximum number of documents for a single segment. If this limit is reached, the segment is closed and a new segment is created. A merge, as governed by "`mergeFactor`" may also occur.

Setting	Description
maxFieldLength	This determines the maximum number of terms that will be stored for a field. If field analysis generates more than the number of indexable tokens specified by this parameter, the excess tokens are discarded. Raising this limit too high can degrade performance because long term lists require more resources and take longer to traverse. Choose this value according to the needs of your application.

8.1.3 Controlling the Behavior of the Update Handler

```
<updateHandler class="solr.DirectUpdateHandler2">
    ...
</updateHandler>
```

The settings in this section are configured in the `<updateHandler>` element in `solrconfig.xml` and may affect the performance of index updates. These settings affect how updates are done internally. `<updateHandler>` configurations do not affect the higher level configuration of `RequestHandlers` that process client update requests.

8.1.3.1 autoCommit

```
<autoCommit>
    <maxDocs>10000</maxDocs>
    <maxTime>1000</maxTime>
</autoCommit>
```

These settings control how often pending updates will be automatically pushed to the index.

Setting	Description
maxDocs	The number of updates that have occurred since the last commit.
maxTime	The number of milliseconds since the oldest uncommitted update.

If either of these limits are reached, then Solr automatically performs a commit operation. If the `<autoCommit>` tag is missing, then only explicit commits will update the index. The decision whether to use auto-commit or not depends on the needs of your application.

Determining the best auto-commit settings is a tradeoff between performance and accuracy. Settings that cause frequent updates will improve the accuracy of searches because new content will be searchable more quickly, but performance may suffer because of the frequent updates. Less frequent updates may improve performance but it will take longer for updates to show up in queries.

8.1.4 maxPendingDeletes

```
<maxPendingDeletes>100000</maxPendingDeletes>
```

This value sets a limit on the number of deletions that Solr will buffer during document deletion. This can affect how much memory is used during indexing.

8.1.5 Query Settings in solrconfig.xml

The settings in this section affect the way that LucidWorks for Solr will process and respond to queries. These settings are all configured in child elements of the `<query>` element in `solrconfig.xml`.

```
<query>
    ...
</query>
```

8.1.5.1 Caching

Solr caches are associated with a specific instance of an Index Searcher—a specific view of an index that doesn't change during the lifetime of that searcher. As long as that Index Searcher is being used, any items in its cache will be valid and available for reuse. Caching in Solr differs from caching in many other applications in that cached Solr objects do not expire after a time interval; instead, they remain valid for the lifetime of the Index Searcher.

When a new searcher is opened, the current searcher continues servicing requests while the new one auto-warms its cache. The new searcher uses the current searcher's cache to pre-populate its own. When the

new searcher is ready, it is registered as the current searcher and begins handling all new search requests. The old searcher will be closed once it has finished servicing all its requests.

In Solr 1.4, there are two cache implementations: `solr.search.LRUCache` and `solr.search.FastLRUCache`.

The acronym LRU stands for Least Recently Used. When an LRU cache fills up, the entry with the oldest last-accessed timestamp is evicted to make room for the new entry. The net effect is that entries that are accessed frequently tend to stay in the cache, while those that are not accessed frequently tend to drop out and will be re-fetched from the index if needed again.

The FastLRUCache, which was introduced in Solr 1.4, is designed to be lock-free, so it is well suited for caches which are hit several times in a request.

The Statistics page in the LucidWorks for Solr Admin Web interface will display information about the performance of all the active caches. This information can help you fine-tune the sizes of the various caches appropriately for your particular application. When a Searcher terminates, a summary of its cache usage is also written to the log.

There are three predefined types of caches you can configure.

8.1.5.2 filterCache

```
<filterCache class="solr.LRUCache" size="512"
    initialSize="512" autowarmCount="128"/>
```

This cache is used by `SolrIndexSearcher` for filters (DocSets) for unordered sets of all documents that match a query. The numeric attributes control the number of entries in the cache.

Solr uses the filterCache to cache results of queries that use the `fq` search parameter. Subsequent queries using the same parameter setting result in cache hits and rapid returns of results. See Chapter 7 for a detailed discussion of the `fq` parameter.

Solr also makes this cache for faceting when the configuration parameter `facet.method` is set to `fc`. For a discussion of faceting, see section 7.6 in Chapter 7.

8.1.5.3 queryResultCache

```
<queryResultCache class="solr.LRUCache" size="512"
    initialSize="512" autowarmCount="128"/>
```

This cache holds the results of previous searches – ordered lists of document ids (DocList) based on a query, a sort, and the range of documents requested.

8.1.5.4 documentCache

```
<documentCache class="solr.LRUCache" size="512"
    initialSize="512" autowarmCount="0"/>
```

This cache holds Lucene Document objects (the stored fields for each document). Since Lucene internal document ids are transient, this cache will not be autowarmed.

8.1.5.5 User Defined Caches

```
<cache name="myUserCache" class="solr.LRUCache" size="4096"
    initialSize="1024" autowarmCount="1024"
    regenerator="org.mycompany.mypackage.MyRegenerator" />
```

You can also define named caches for your own application code to use. You can locate and use your cache object by name by calling the `SolrIndexSearcher` methods `getCache()`, `cacheLookup()` and `cacheInsert()`. If you want auto-warming of your cache, include a "`regenerator`" attribute with the fully qualified name of a class that implements `solr.search.CacheRegenerator`.

8.1.6 maxBooleanClauses

```
<maxBooleanClauses>1024</maxBooleanClauses>
```

This sets the maximum number of clauses allowed in a boolean query. This can affect range or prefix queries that expand to a query with a large number of boolean terms. If this limit is exceeded, an exception is thrown.

8.1.7 enableLazyFieldLoading

```
<enableLazyFieldLoading>true</enableLazyFieldLoading>
```

If this parameter is set to `true`, then fields that are not directly requested will be loaded lazily as needed. This can boost performance if the most common queries only need a small subset of fields, especially if infrequently accessed fields are large in size.

8.1.8 useColdSearcher

```
<useColdSearcher>false</useColdSearcher>
```

This setting controls whether search requests for which there is not a currently registered searcher should wait for a new searcher to warm up (`false`) or proceed immediately (`true`). When set to "false", requests will block until the searcher has warmed its caches.

8.1.9 maxWarmingSearchers

```
<maxWarmingSearchers>2</maxWarmingSearchers>
```

This parameter sets the maximum number of searchers that may be warming up in the background at any given time. Exceeding this limit will raise an error. For read-only slaves, a value of two is reasonable. Masters should probably be set a little higher.

8.1.10 HTTP RequestDispatcher Settings

The `<requestDispatcher>` element of `solrconfig.xml` controls the way the Solr servlet's `RequestDispatcher` implementation responds to HTTP requests.

8.1.10.1 handleSelect Attribute

The first configurable item is the `handleSelect` attribute on the `<requestDispatcher>` element itself:

```
<requestDispatcher handleSelect="true" >
    ...
</requestDispatcher>
```

This attribute can be set to one of two values, either "`true`" or "`false`". A value of "true" (the default) indicates that error handling should consistent for `/select` and `/update` URLs. The value "`false`" indicates that error formatting should be compatible with Solr 1.1.

8.1.10.2 requestParsers Element

```
<requestDispatcher handleSelect="true">
    <requestParsers
        enableRemoteStreaming="false" multipartUploadLimitInKB="2048"/>
</requestDispatcher>
```

The `<requestParsers>` sub-element controls a couple of values related to parsing requests. This is an empty XML element that doesn't have have any content, only attributes. The attribute "`enableRemoteStreaming`" controls whether remote streaming of content is allowed. If set to "`false`" (the default), streaming will not be allowed. Setting it to "`true`" lets you specify the location of content to be streamed using `stream.file` or `stream.url` parameters.

If you enable remote streaming, be sure that you have authentication enabled. Otherwise, someone could potentially gain access to your content by accessing arbitrary URLs. It's also a good idea to place Solr behind a firewall to prevent it being accessed from untrusted clients.

The attribute "`multipartUploadLimitInKB`" sets an upper limit on the size of a document that may be submitted in a multi-part HTTP POST request. The value specified is multiplied by 1024 to determine the size in bytes.

8.1.10.3 httpCaching Element

```
<httpCaching never304="false" lastModFrom="openTime" etagSeed="Solr">
    <cacheControl>max-age=30, public</cacheControl>
</httpCaching>
```

The `<httpCaching>` element controls HTTP cache control headers. Do not confuse these settings with Solr's internal cache configuration. This element controls caching of HTTP responses as defined by the W3C HTTP specifications.

This element allows for three attributes and one sub-element. The attributes of the `<httpCaching>` element control whether a 304 response to a GET request is allowed, and if so, what sort of response it should be. When an HTTP client application issues a GET, it may optionally specify that a 304 response is acceptable if the resource has not been modified since the last time it was fetched.

Parameter	Description
`never304`	If present with the value `"true"`, then a GET request will never respond with a 304 code, even if the requested resource has not been modified. When this attribute is set to true, the following two attributes are ignored. Setting this to true is handy for development, as the 304 response can be confusing when tinkering with Solr responses through a web browser or other client that supports cache headers.
`lastModFrom`	This attribute may be set to either "`openTime`" (the default) or "`dirLastMod`". The value "`openTime`" indicates that last modification times, as compared to the `If-Modified-Since` header sent by the client, should be calculated relative to the time the Searcher started. Use "`dirLastMod`" if you want times to exactly correspond to when the index was last updated on disk.
`etagSeed`	This value of this attribute is sent as the value of the "`ETag`" header. Changing this value can be helpful to force clients to re-fetch content even when the indexes have not changed—for example, when you've made some changes to the configuration.

The cacheControl Element

In addition to these attributes, `<httpCaching>` accepts one child element: `<cacheControl>`. The content of this element will be sent as the value of the `Cache-Control` header on HTTP responses. This header is used to modify the default caching behavior of the requesting client. The possible values for the `Cache-Control` header are defined by the HTTP 1.1 specification in Section 14.9[16].

Setting the `max-age` field controls how long a client may re-use a cached response before requesting it again from the server. This time interval should be set according to how often you update your index and whether or not it is acceptable for your application to use content that is somewhat out of date. Setting `must-revalidate` will tell the client to validate with the server that its cached copy is still good before re-using it. This will ensure that the most timely result is used, while avoiding a second fetch of the content if it isn't needed, at the cost of a request to the server to do the check.

16 http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9

8.2 Using Multiple SolrCores

It's possible to segment Solr into multiple virtual instances, or *cores*, each with its own configuration and indices. Cores may be dedicated to a single application or to very different ones, but all are administered through a common administration interface. You can create new SolrCores on the fly, shutdown cores, even replace one running core with another, all without ever stopping or restarting your servlet container.

SolrCores are configured by placing a file named `solr.xml` in your `solr.home` directory. A typical `solr.xml` looks like this:

```
<solr persistent="true" sharedLib="lib">
    <cores adminPath="/admin/cores">
        <core name="core0" instanceDir="core0dir"/>
        <core name="core1" instanceDir="core1dir"/>
    </cores>
</solr>
```

This sets up two SolrCores, named "`core0`" and "`core1`" and names the directories (relative to the Solr installation path) which will store the configuration and data subdirectories.

NOTE: As of Solr 1.4, it's possible to run Solr without configuring any cores.

8.2.1 The <solr> Element

There are two attributes that you can specify on `<solr>`, which is the root element of `solr.xml`.

Attribute	Description
`persistent`	Indicates that changes made through the API or admin UI should be saved back to this `solr.xml` If not "`true`", any runtime changes will be lost on the next Solr restart. The servlet container running Solr must have sufficient permissions to replace `solr.xml` (file delete and create), or errors will result. Any comments in `solr.xml` are not preserved when the file is updated.

Attribute	Description
sharedLib	Specifies the path to a common library directory that will be shared across all cores. Any JAR files in this directory will be added to the search path for Solr plugins. This path is relative to the top-level container's solr.home.

NOTE: If you set the persistent attribute to true, be sure that the Web server has permission to replace the file. If the permissions are set incorrectly, the server will generate 500 errors and throw IOExceptions. Also, note that any comments in the solr.xml file will be lost when the file is overwritten.

8.2.2 The <cores> Element

The <cores> element, which contains definitions for each SolrCore, is a child of <solr> and accepts three attributes of its own.

Attribute	Description
adminPath	This is the relative URL path to access the SolrCore administration pages. For example, a value of "/admin/cores" means that you can access the CoreAdminHandler with a URL that looks like this: http://localhost:8983/solr/admin/cores. If this attribute is not present, then SolrCore administration will not be possible.
shareSchema	This attribute, when set to "true," ensures that the multiple cores pointing to the same schema.xml will be referring to the same IndexSchema Object. Sharing the IndexSchema Object makes loading the core faster. If you use this feature, make sure that no core-specific property is used in your schema.xml.

Attribute	Description
adminHandler	If used, this attribute should be set to the FQN(Fully qualified name) of a class that inherits from CoreAdminHandler. For example, `adminHandler="com.myorg.MyAdminHandler"` would configure the custom admin handler (MyAdminHandler) to handle admin requests. If this attribute isn't set, Solr uses the default admin handler, `org.apache.solr.handler.admin.CoreAdminHandler`.

For a use case of the adminHandler attribute, suppose we wanted to get statistics from different cores in a Solr instance. First, we could define a new action called 'mystat' that could be accessed from the client as below.

http://localhost:8983/solr/admin/cores?action=MYSTAT

Then, we would define the implementation of the MYSTAT action like so:

```
import org.apache.solr.handler.admin.CoreAdminHandler ;

class MyAdminHandler extends CoreAdminHandler {

  /**
   * @return true, if the changes need to be persisted by the
CoreContainer. (use only if solr.xml would be changed because of this
action. )
   *         false, otherwise.   (Use this if unsure or having a read-only
access to the CoreContainer like collecting statistics)
   *
   */
  protected boolean handleCustomAction(SolrQueryRequest req,
SolrQueryResponse rsp) {
     CoreContainer container = super.getCoreContainer();
     SolrCore mycore1 = container.getCore("core1");
     SolrCore mycore2 = container.getCore("core2");
     SolrParams params = req.getParams();
     String a = params.get( CoreAdminParams.ACTION );
     if (a.toLowerCase().equals("mystat"))  {
         // TODO: populate 'rsp' as necessary.
      }
  }
}
```

There are other methods in CoreAdminHandler that could be used to override default actions, but for most of the common cases they would not be necessary.

8.2.3 The <core> Element

There is one `<core>` element for each SolrCore you define. They are children of the `<cores>` element and each one accepts six attributes.

Attribute	Description
`name`	The name of the SolrCore. You'll use this name to reference the SolrCore when running commands with the CoreAdminHandler.
`instanceDir`	This relative path defines `solr.home` for the core.
`config`	The configuration file name for a given core. The default is '`solrconfig.xml`'.
`schema`	The schema file name for a given core. The default is '`schema.xml`'
`dataDir`	This relative path defines `solr.home` for the core.
`properties`	The name of the properties file for this core. The value can be an absolute pathname or a path relative to the value of `instanceDir`.

8.2.4 Properties in solr.xml

You can define properties in `solr.xml` that you may then reference in `solrconfig.xml` and `schema.xml`. Properties are name/value pairs. The scope of a property depends on which element it occurs within.

```
<solr persistent="true" sharedLib="lib">
    <property name="productname" value="Acme Online"/>
    <cores adminPath="/admin/cores">
        <core name="core0" instanceDir="core0">
            <property name="dataDir" value="/data/core0"/>
        </core>
```

```
        <core name="core1" instanceDir="core1"/>
    </cores>
</solr>
```

If a property is declared under `<solr>` but outside a `<core>` element, then it will have container scope and will be visible to all cores. In the example above, "`productname`" is such a property.

If a property declaration occurs within a `<core>` element, then its scope is limited to that core and it will not be visible to other cores. A property at core scope will override one of the same name declared at container scope.

In addition to any properties you declare at core scope, there are several properties that Solr defines automatically for each core. These properties are described in the table below:

Property	Description
`solr.core.name`	The core's name, as defined by the "`name`" attribute.
`solr.core.instanceDir`	The core's instance directory under which that its `conf/` and `data/` directories are located, derived from the core's "`instanceDir`" attribute.
`solr.core.dataDir`	The core's data directory, `${solr.core.instanceDir}/data` by default.
`solr.core.configName`	The name of the core's configuration file, `solrconfig.xml` by default.
`solr.core.schemaName`	The name of the core's schema file, `schema.xml` by default.

Any of the above properties can be referenced by name in `schema.xml` or `solrconfig.xml`.

When defining properties, you can assign a property a default value that will be used if another value isn't specified.. For example:

```
// Without a default value, result will be empty if property not defined
${productname}
// With a default value
${productname:SearchCo MegaIndex}
```

8.2.5 CoreAdminHandler

The `CoreAdminHandler` is a special `SolrRequestHandler` that is used to manage SolrCores. Unlike normal `SolrRequestHandlers`, the `CoreAdminHandler` is not attached to a single core. Instead, it manages all the cores running in a single Solr instance. Only one `CoreAdminHandler` exists for each top-level Solr instance.

To use the `CoreAdminHandler`, make sure that the "`adminPath`" attribute is defined on the `<cores>` element; otherwise you will not be able to make HTTP requests to perform SolrCore administration.

The `CoreAdminHandler` supports seven different actions that may be invoked on the "`adminPath`" URL. The action to perform is named by the HTTP request parameter "`action`", with arguments for a specific action being provided as additional parameters.

All action names are uppercase. The actions names are:

- STATUS
- CREATE
- RELOAD
- RENAME
- ALIAS
- SWAP
- UNLOAD

These actions are described in detail in the sections below.

8.2.5.1 STATUS

The STATUS action returns the status of all running SolrCores, or status for only the named core.

```
http://localhost:8983/solr/admin/cores?action=STATUS
http://localhost:8983/solr/admin/cores?action=STATUS&core=core0
```

The STATUS action accepts one optional parameter

Parameter	Description
core	(Optional) The name of a core, as listed in the "name" attribute of a <core> element in solr.xml.

8.2.5.2 CREATE

The CREATE action creates a new core and registers it. If persistence is enabled (persistent="true" on the <solr> element), the updated configuration for this new core will be saved in solr.xml. If a SolrCore with the given name already exists, it will continue to handle requests while the new core is initializing. When the new core is ready, it will take new requests and the old core will be unloaded.

```
http://localhost:8983/solr/admin/cores?action=CREATE
    &name=coreX&instanceDir=path/to/dir
    &config=config_file_name.xml&schema=schem_file_name.xml&dataDir=data
```

The CREATE accepts the two mandatory parameters, as well as three optional parameters.

Parameter	Description
name	The name of the new core. Same as "name" on the <core> element.
instanceDir	The directory where files for this SolrCore should be stored. Same as "instanceDir" on the <core> element.
config	(Optional) Name of the config file (solrconfig.xml) relative to "instanceDir".
schema	(Optional) Name of the schema file (schema.xml) relative to "instanceDir".
datadir	(Optional) Name of the data directory relative to "instanceDir".

8.2.5.3 RELOAD

The RELOAD action loads a new core from the configuration of an existing, registered SolrCore. While the new core is initializing, the existing one will continue to handle requests. When the new SolrCore is ready, it takes over and the old core is unloaded.

This is useful when you've made changes to a SolrCore's configuration on disk, such as adding new field definitions. Calling the RELOAD action lets you apply the new configuration without having to restart the Web container.

```
http://localhost:8983/solr/admin/cores?action=RELOAD&core=core0
```

The RELOAD action accepts a single parameter

Parameter	Description
`core`	The name of the core to be reloaded.

8.2.5.4 RENAME

The RELOAD action hanges the name of a SolrCore.

```
http://localhost:8983/solr/admin/cores?action=RENAME
    &core=core0&other=core5
```

The RELOAD action requires the following two parameter:

Parameter	Description
`core`	The name of the SolrCore to be renamed.
`other`	The new name for the SolrCore. If the `persistent` attribute of <solr> is "`true`", the new name will be written to `solr.xml` as the "`name`" attribute of the `<core>` attribute.

8.2.5.5 ALIAS

The ALIAS action establishes an additional name by which a SolrCore may be referenced. Subsequent actions may use the SolrCore's original name or any of its aliases.

NOTE: This action is still considered experimental.

```
http://localhost:8983/solr/admin/cores?action=ALIAS&core=coreX&other=coreY
```

The ALIAS action requires two parameters:

Parameter	Description
core	The name or alias of an existing core.
other	The additional name by which this core should be known.

8.2.5.6 SWAP

SWAP atomically swaps the names used to access two existing SolrCores. This can be used to swap new content into production. The prior core remains available and can be swapped back, if necessary. Each core will be known by the name of the other, after the swap.

```
http://localhost:8983/solr/admin/cores?action=SWAP&core=core1&other=core0
```

The SWAP action requires two parameters, which are described in the table below.

Parameter	Description
core	The name of one of the cores to be swapped.
other	The name of one of the cores to be swapped.

8.2.5.7 UNLOAD

The UNLOAD action removes a core from LucidWorks for Solr. Active requests will continue to be processed, but no new requests will be sent to the named core. If a core is registered under more than one name, only the given name is removed.

```
http://localhost:8983/solr/admin/cores?action=UNLOAD&core=core0
```

The UNLOAD action requires a parameter identifying the core to be removed.

Parameter	Description
`core`	The name of the core to be to be removed. If the `persistent` attribute of `<solr>` is set to "`true`", the `<core>` element with this "`name`" attribute will be removed from `solr.xml`.

8.3 Solr Plugins

Solr allows you to load custom code to perform a variety of tasks within Solr—from custom Request Handlers to process your searches, to custom Analyzers and Token Filters for your text field. You can even load custom Field Types. These pieces of custom code are called plugins.

8.3.1 Loading Plugins

To load custom code into Solr, do one of the following:

- Install JAR files containing your classes in a `lib/` directory in your Solr Home directory before you start the servlet container. This `lib/` directory does not exist in the distribution, so you will need to create it before trying to install the JARs. In the example application, JARs may be installed in `example/solr/lib`.
- Install JARs for the plugin in a directory, and then specify that directory using a `<lib>` directive in `solrconfig.xml`. The `<lib>` directive specifies the path to a directory containing JARs to load; the path must be specified relative to `instanceDir`. The `<lib>` directive can include `regex` expressions, as shown in some of the examples below.

The directive below selects all the JARs found in the specified directory.

```
<lib dir="../../contrib/extraction/lib" />
```

The directive below selects only the JARs that match the regulation expression specified:

```
<lib dir="../../dist/" regex="apache-solr-cell-\d.*\.jar" />
```

NOTE: You can specify an exact filename for a file to be loaded, but if the file cannot be found or loaded, this will produce a serious error.

This feature for loading plugins uses a custom class loader. It has been tested with a variety of servlet containers, including Jetty and Tomcat.

8.3.2 Initializing Plugins

Plugins are initialized with either:

```
init( Map<String,String> args )
```

or with:

```
init (NamedList args )
```

8.3.2.1 ResourceLoaderAware

Classes that need to know about the ResourceLoader can implement ResourceLoaderAware. Valid ResourceLoaderAware classes include:

- CommonGramsFilterFactory
- CommonGramsQueryFilterFactory
- DelimitedPayloadTokenFilterFactory
- DictionaryCompoundWordTokenFilterFactory
- ElisionFilterFactory

- EnglishPorterFilterFactory
- KeepWordFilterFactory
- MappingCharFilterFactory
- SnowballPorterFilterFactory
- StopFilterFactory
- SynonymFilterFactory
- WordDelimiterFilterFactory

8.3.2.2 SolrCoreAware

Classes that need to know about the SolrCore can implement SolrCoreAware. Valid SolrCoreAware classes include:

- SolrRequestHandler
- QueryResponseWriter
- SearchComponent

8.3.2.3 Plugin Initialization Lifecycle

The initialization lifecycle for a plugin is:

1. Constructor
2. `init( Map / NamedList )`
3. ResourceLoaderAware classes call: `inform( ResourceLoader );`
4. Before the first request is made and after all plugins have been created and registered, SolrCoreAware plugins call: `inform( SolrCore );`

8.3.3 Classes That are Pluggable

The following is a complete list of every API that can be treated as a plugin in Solr, with information on how to use that configure your Solr instance to use an instance of that class.

8.3.3.1 Classes for Request Processing

SolrRequestHandler

Instances of SolrRequestHandler define the logic that is executed for any request. Multiple handlers (including multiple instances of the same SolrRequestHandler class with different configurations) can be specified in `solrconfig.xml`, as shown in the example below.

```
  <requestHandler name="foo" class="my.package.CustomRequestHandler" />
  <requestHandler name="bar"
class="my.package.AnotherCustomRequestHandler" />
  <requestHandler name="baz"
class="my.package.AnotherCustomRequestHandler">
    <!-- initialization args may optionally be defined here -->
     <lst name="defaults">
       <int name="rows">10</int>
       <str name="fl">*</str>
       <str name="version">2.1</str>
     <lst>
     <int name="someConfigValue">42</int>
  </requestHandler>
```

For more information about Request Handlers, see Chapter 7.

SearchComponent

Instances of SearchComponent define discrete units of logic that can be combined together and reused by Request Handlers (in particular, SearchHandler) that know about them. Search Components, too, can accept plugins.

QParserPlugin

QParserPlugin can be used to define customized user query processing instances of QParser .

First, implement a subclass of QParserPlugin and register it in `solrconfig.xml` like this:

```
<queryParser name="myqueryparser" class="my.package.MyQueryParserPlugin" />
```

Having done this, you can choose to use your query parser on a one-time basis using the defType query parameter, like this:

```
http://mysolrmachine:8983/solr/select/?defType=myqueryparser&q=hi
```

You can also specify your query parser with the `q` parameter, like this:

```
http://mysolrmachine:8983/solr/select/?&q={!myqueryparser}hi
```

For more permanent use, you will likely want to configure your Request Handler defaults to set the `defType`, like this:

```
  <requestHandler name="dismax" class="solr.SearchHandler" >
   <lst name="defaults">
    <str name="defType">myqueryparser</str>
    ...
```

ValueSourceParser

Use this to plugin your own custom functions see FunctionQuery. Register the plugin in `solrconfig.xml` directly under the `<config>` tag. For example:

```
<valueSourceParser name="myfunc" class="com.mycompany.MyValueSourceParser"
/>
```

The class must implement `org.apache.solr.search.ValueSourceParser`.

QueryResponseWriter

Instances of QueryResponseWriter define the formatting used to output the results of a request. Multiple writers (including multiple instances of the same QueryResponseWriter class with different configurations) can be specified in your solrconfig.xml...

```
  <queryResponseWriter name="wow"
class="my.package.CustomResponseWriter" />
```

```
  <queryResponseWriter name="woz"
class="my.package.AnotherCustomResponseWriter" />
  <queryResponseWriter name="woz"
class="my.package.AnotherCustomResponseWriter" >
    <!-- initialization args may optionally be defined here -->
    <int name="someConfigValue">42</int>
  </queryResponseWriter>
```

Similarity

Similarity is a native Lucene class that determines how much of the score calculations for the various types of queries are executed. For more information using the methods in the Similarity class, consult the Lucene scoring documentation, which is available here:

```
http://lucene.apache.org/java/2_9_1/scoring.html
```

If you wish to override the DefaultSimilarity provided by Lucene, you can specify your own subclass in your `schema.xml` file, like so

```
  <similarity class="my.package.CustomSimilarity"/>
```

CacheRegenerator

The CacheRegenerator API allows people who are writing custom SolrRequestHandlers which utilize custom User Caches to specify how those caches should be populated during auto-warming. A regenerator class can be specified when the cache is declared in `solrconfig.xml`.

```
    <cache name="myCustomCacheInstance"
      class="solr.LRUCache"
      size="4096"
      initialSize="1024"
      autowarmCount="1024"
      regenerator="my.package.CustomCacheRegenerator"
      />
```

8.3.3.2 Other Pluggable Interfaces

You can also create plugins for:

- Highlighting
- SolrFragmenter
- SolrFormatter
- UpdateRequestProcessorFactory

8.3.4 Plugins and Fields

8.3.4.1 The Analyzer Class

The Analyzer class is a native Lucene concept that determines how tokens are produced from a piece of text. Solr allows Analyzers to be specified for each `fieldtype` in your `schema.xml` that uses the TextField class. Solr also allows you to specify different Analyzers for indexing text as documents are added and for parsing text specified in a query.

```
    <fieldtype name="text_foo" class="solr.TextField">
      <analyzer class="my.package.CustomAnalyzer"/>
    </fieldType>
    <fieldtype name="text_bar" class="solr.TextField">
      <analyzer type="index"
class="my.package.CustomAnalyzerForIndexing"/>
      <analyzer type="query" class="my.package.CustomAnalyzerForQuering"/>
    </fieldType>
```

Solr also provides a SolrAnalyzer base class which you can use if you want to write your own Analyzer and configure the "positionIncrementGap" in your `schema.xml` file.

```
    <fieldtype name="text_baz" class="solr.TextField"
positionIncrementGap="100">
      <analyzer class="my.package.CustomSolrAnalyzer" />
    </fieldType>
```

Specifying an Analyzer class in your `schema.xml` makes a lot of sense if you already have an existing Analyzer you wish to use as is, but if you are planning to write Analysis code from scratch that you would like to use in Solr, you should keep reading the following sections.

8.3.5 Tokenizer and TokenFilter

In addition to specifying Analyzer classes, Solr can construct Analyzers on the fly for each field type using a Tokenizer and any number of TokenFilters. To take advantage of this functionality with any Tokenizers or TokenFilters you may have or you want to implement, you'll need to provide a TokenizerFactory and TokenFilterFactory which take care of any initialization and configuration, and specify these Factories in your `schema.xml` file, like so:

```
    <fieldtype name="text_zop" class="solr.TextField"
positionIncrementGap="100">
      <analyzer>
          <tokenizer class="my.package.CustomTokenizerFactory"/>
          <!-- this TokenFilterFactory has custom options -->
          <filter class="my.package.CustomTokenFilter" optA="yes"
optB="maybe" optC="42.5"/>
          <!-- Solr has many existing FilterFactories that you can reuse
-->
          <filter class="solr.StopFilterFactory" ignoreCase="true"/>
      </analyzer>
    </fieldtype>
```

8.3.5.1 The FieldType Class

If you have special needs for data types, you can specify your own FieldType class for each `<fieldtype>` you declare in `schema.xml` in order to control how the values for those fields are encoded in your index. For example:

```
    <fieldtype name="wacko" class="my.package.CustomFieldType" />
```

8.3.6 Internals

8.3.6.1 The SolrCache API

The SolrCache API allows you to specify custom cache implementations for any of various caches you might declare in your `solrconfig.xml`. For example:

```
<filterCache       class="my.package.CustomCache"          size="512" />
<queryResultsCache class="my.package.CustomCache"          size="512"  />
<documentCache     class="my.package.AlternateCustomCache" size="512"  />
```

8.3.6.2 SolrEventListener

You can configure instances of the SolrEventListener in `solrconfig.xml`, so that a listener executes when specific Solr events occur.

NOTE: Currently the only events that can be "listened" for are `firstSearcher` and `newSearcher.)`

```
    <listener event="newSearcher" class="my.package.CustomEventListener">
      <-- init args for the EventListener instance can be specified here
-->
      <lst name="arg1">
        <str name="q">solr</str> <str name="start">0</str> <str
name="rows">10</str>
      </lst>
      <int name="otherArg">42</int>
    </listener>
```

8.3.6.3 The UpdateHandler API

The UpdateHandler API allows you to specify a custom algorithm for determining how Solr processes sequences of adds and deletes. You can configure an UpdateHandler in `solrconfig.xml file`, but implementing a new UpdateHandler is considered **extremely** advanced and is not recommended.

```
  <updateHandler class="my.package.CustomUpdateHandler">
```

8.4

8.5 JVM Settings

Configuring your JVM can be a complex topic. A full discussion is beyond the scope of this document. Luckily, most modern JVMs are quite good at making the best use of available resources with default settings. The following sections contain a few tips that may be helpful when the defaults are not optimal for your situation.

8.5.1 Choosing Memory Heap Settings

The most important JVM configuration settings are those that determine the amount of memory it is allowed to allocate. There are two primary command-line options that set memory limits for the JVM. These are "`-Xms`", which sets the initial size of the JVM's memory heap, and "`-Xmx`", which sets the maximum size to which the heap is allowed to grow.

If your Solr application requires more heap space than you specify with the "`-Xms`" option, the heap will grow automatically. It's quite reasonable to not specify an initial size and let the heap grow as needed. The only downside is a somewhat slower startup time since the application will take longer to initialize. Setting the initial heap size higher than the default may avoid a series of heap expansions, which often results in objects being shuffled around within the heap, as the application spins up.

The maximum heap size, set with "`-Xmx`", is more critical. If the memory heap grows to this size, object creation may begin to fail and throw `OutOfMemoryException`. Setting this limit too low can cause spurious errors in your application, but setting it too high can be detrimental as well.

It doesn't always cause an error when the heap reaches the maximum size. Before an error is raised, the JVM will first try to reclaim any available space that already exists in the heap. Only if all garbage collection attempts fail will your application see an exception. As long as the maximum is big enough, your app will run without error, but it may run more slowly if forced garbage collection kicks in frequently.

The larger the heap the longer it takes to do garbage collection. This can mean minor, random pauses or, in extreme cases, "freeze the world" pauses of a minute or more. As a practical matter, this can become a serious problem for heap sizes that exceed about two gigabytes, even if far more physical memory is

available. On beefy hardware, you may get better results running multiple JVMs, rather than just one with a huge memory heap. Some specialized JVM implementations may have customized garbage collection algorithms which do better with large heaps. Also, Java 7 is anticipated to have a redesigned GC that should handle very large heaps efficiently. Consult your JVM vendor's documentation.

When setting the maximum heap size, be careful not to let the JVM consume all available physical memory. If the JVM process space grows too large, the operating system will start swapping it, which will severely impact performance. In addition, the operating system uses memory space not allocated to processes for file system cache and other purposes. This is especially important for I/O-intensive applications, like Lucene/Solr. The larger your indices, the more you will benefit from filesystem caching by the OS. It may require some experimentation to determine the optimal tradeoff between heap space for the JVM and memory space for the OS to use.

On systems with many CPUs/cores, it can also be beneficial to tune the layout of the heap and/or the behavior of the garbage collector. Adjusting the relative sizes of the generational pools in the heap can affect how often GC sweeps occur and whether they run concurrently. Configuring the various settings of how the garbage collector should behave can greatly reduce the overall performance impact when it does run. There is a lot of good information on this topic available on Sun's website. A good place to start is here: http://java.sun.com/javase/technologies/hotspot/gc/.

8.5.2 Use the Server HotSpot VM

If you're using Sun's JVM, be sure to add the "`-server`" command-line option when you start LucidWorks for Solr. This tells the JVM that it should optimize for a long running, server process. If the Java runtime on your system is a JRE, rather than a full JDK distribution (including javac and other development tools), then it's possible that it may not support the "`-server`" JVM option. Test this by running "`java -help`" and look for `-server` as an available option in the displayed usage message.

8.5.3 Checking JVM Settings

A great way to see what JVM settings your server is using, along with other useful information, is to use the admin RequestHandler, `solr/admin/system`. This request handler will display a wealth of server statistics and settings.

You can also use any of the tools that are compatible with the Java Management Extensions (JMX). See the section *Using JMX with Solr* in Chapter 9 for more information.

9 Managing Solr

9.1 Introduction

This chapter describes how to run Solr and how to look at Solr when it's running. Solr is a Java Web application and can run in a wide variety of Java application containers. The first part of this chapter contains details about how to run Solr on Tomcat, one of the most popular Web application containers, and the one which LucidWorks includes as the platform for running Solr.

Next up is a description of logging and how it is configured.

After that, you will learn about LucidGaze for Solr, which provides real-time, graphic analysis of Solr's performance.

This chapter concludes with a brief description of making backups and how Solr can be exposed via JMX.

9.2 Running LucidWorks for Solr on Tomcat

LucidWorks comes with an example schema and scripts for running Solr on either Tomcat or Jetty. See Chapter 2 for details on the LucidWorks installation options and how to start and stop Solr using the provided scripts. The next section describes some of the details of how things work "under the hood." The

following sections cover running multiple Solr instances and deploying Solr using the Tomcat application manager.

9.2.1 How Solr Works with Tomcat

The two basic steps for running Solr in any Web application container are as follows:

Make the Solr classes available to the container. In many cases, the Solr Web application archive (WAR) file can be placed into a special directory of the application container. In the case of Tomcat, you need to place the Solr WAR file in Tomcat's `webapps` directory. If you installed Tomcat with SolrWorks, take a look in `tomcat/webapps`—you'll see the `solr.war` file is already there.

Point Solr to the `solr` home directory that contains `conf/solrconfig.xml` and `conf/schema.xml`. There are a few ways to get this done. One of the best is to define the `solr.solr.home` Java system property. With Tomcat, the best way to do this is via a shell environment variable, `JAVA_OPTS`. Tomcat puts the value of this variable on the command line upon startup. Here is an example:

```
export JAVA_OPTS="-Dsolr.solr.home=/Users/jonathan/Desktop/solr"
```

Fortunately, LucidWorks comes with everything packaged up neatly. See Chapter 2 for details of starting Solr with the included script.

When you installed SolrWorks, the installer configured port 8983 to be the port upon which the server listens. If you are using Tomcat and wish to change this port, edit the file `tomcat/conf/server.xml` in the LucidWorks distribution. You'll find the port in this part of the file:

```
<Connector port="8983" protocol="HTTP/1.1"
connectionTimeout="20000"
redirectPort="8443" />
```

Modify the port number as desired and restart Tomcat if it is already running.

9.2.2 Running Multiple Solr Instances

Running multiple instances of Solr on a single Tomcat instance is also possible using Tomcat context fragments. In Tomcat's `conf/Catalina/localhost` directory (you might need to create it), store one context fragment per instance of Solr.

Each context fragment specifies where to find the Solr WAR and the path to the `solr` home directory. The name of the context fragment file determines the URL used to access that instance of Solr. For example, a context fragment named `harvey.xml` would deploy Solr to be accessed at http://localhost:8983/harvey.

Using Tomcat context fragments, you could run multiple instances of Solr on the same server, each with its own schema and configuration. For full details and examples of context fragments, take a look at the Solr Wiki: http://wiki.apache.org/solr/SolrTomcat

Here are examples of context fragments which would set up two Solr instances, each with its own `solr.home`:

```
<Context docBase="/some/path/solr.war" debug="0" crossContext="true" >
  <Environment name="solr/home" type="java.lang.String"
value="/some/path/solr1home" override="true" />
</Context>

<Context docBase="/some/path/solr.war" debug="0" crossContext="true" >
  <Environment name="solr/home" type="java.lang.String"
value="/some/path/solr2home" override="true" />
</Context>
```

An alternative way to deploy multiple Solr index instances in a single Web application is to use the multicore API described in Chapter 8.

9.2.3 Deploying Solr with the Tomcat Manager

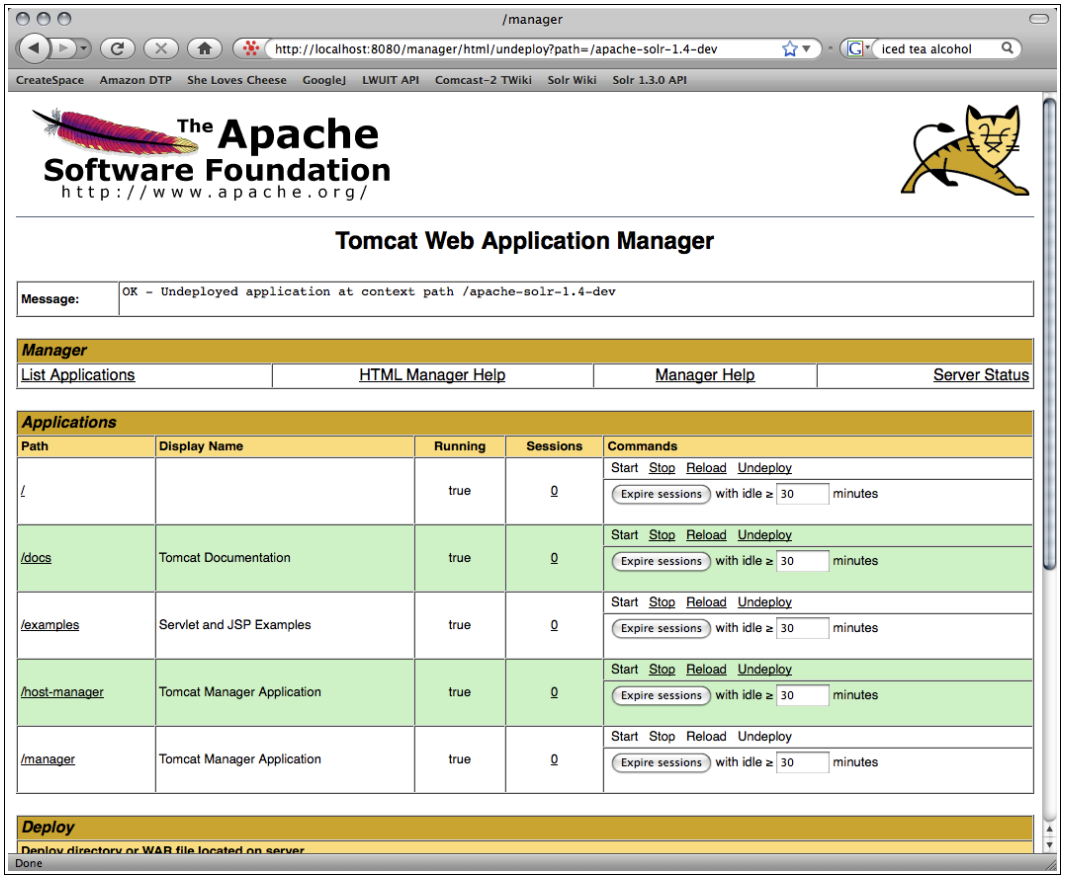
If your instance of Tomcat is running the Tomcat Web Application Manager, you can use its interface from your browser to deploy Solr.

Just as before, you have to tell Solr where to find the `solr` home directory. You can do this by setting `JAVA_OPTS` before starting Tomcat.

Once Tomcat is running, navigate to the Web application manager, probably available at a URL like this:

http://localhost:8983/manager/html

You'll see the main screen of the manager.



To add Solr, scroll down to the **Deploy** section, specifically **WAR file to deploy**. Click on **Browse...** and find the Solr WAR file, usually something like `dist/apache-solr-1.3.0.war` within your LucidWorks installation. Click on **Deploy**. Tomcat will load the WAR file and start running it. Click on the link in the application path column of the manager to see Solr. You won't see much, just a welcome screen, but it contains a link for the Admin Console.

Tomcat's manager screen, in its application list, has links so you can stop, start, reload, or undeploy the Solr application.

9.3 Running LucidWorks for Solr on Jetty

9.3.1 Changing the Port Solr Listens On

The LucidWorks for Solr installer configures Solr to listen on port 8983, which is the default port for Solr. If you are using Jetty and wish to change the, edit the file `jetty/etc/jetty.xml` in the LucidWorks distribution. You'll find the port in this part of the file:

```
    <New class="org.mortbay.jetty.bio.SocketConnector">
      <Set name="port"><SystemProperty name="jetty.port"
default="8983"/></Set>
      <Set name="maxIdleTime">50000</Set>
      <Set name="lowResourceMaxIdleTime">1500</Set>
    </New>
```

Modify the port number as desired, and restart Jetty if it is already running.

For both Jetty and Tomcat, modifying the port number will leave some of the samples and help file links pointing to the default port. It is out of the scope of this reference guide to provide full details of how to change all of the examples and other resources to the new port.

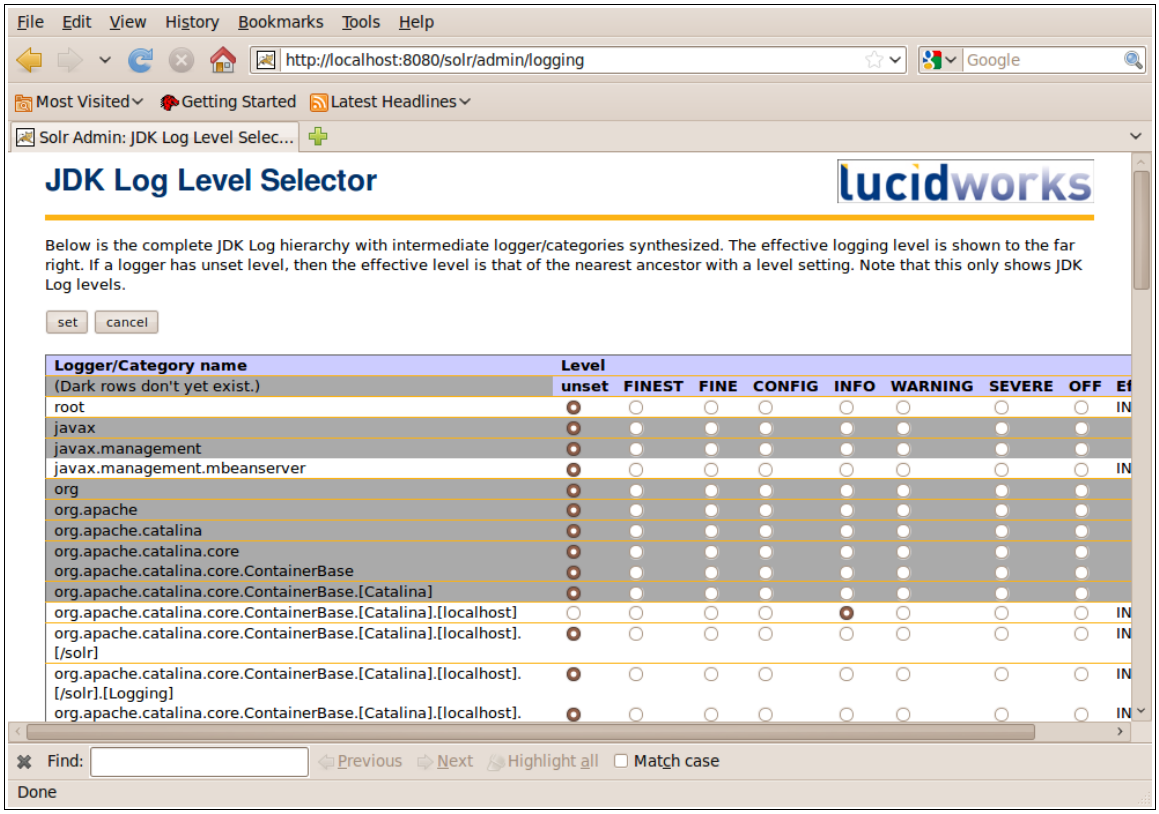
9.4 Configuring Logging

Logging is the practice of writing informative messages somewhere. System administrators or developers can read logs to learn information about a system. If an application dies unexpectedly, the key to its demise might be written in a log somewhere. A canny developer can examine a log to understand what went wrong, much like a detective can examine the scene of a crime to find out what happened.

Version 1.4 of LucidWorks for Solr uses the SLF4J Logging API (`http://www.slf4j.org`). If you want to see the log output in Tomcat, look in `solrworks/tomcat/logs`. You'll find a file named something like `catalina.2009-06-08.log`, except with the current date.

9.4.1 Temporary Logging Settings

You can control the amount of logging output in Solr by using the Admin Web interface. Select the **LOGGING** link. Note that this page only lets you change settings in the running system and is not saved for the next run. (For more information about the Admin Web interface, see Chapter 3.)



The JDK Log Level Selector screen.

This part of the Admin Web interface allows you to set the logging level for many different log categories. Fortunately, any categories that are **unset** will have the logging level of its parent. This makes it possible to change many categories at once by adjusting the logging level of their parent.

9.4.2 Permanent Logging Settings

Making permanent changes to the JDK Logging API configuration is a matter of creating or editing a properties file.

9.4.2.1 Tomcat Logging Settings

Tomcat offers a choice between settings for all applications or settings specifically for the Solr application.

To change logging settings for Solr only, edit `tomcat/webapps/solr/WEB-INF/classes/logging.properties`. You will need to create the `classes` directory and the `logging.properties` file. You can set levels from FINEST to SEVERE for a class or an entire package. Here are a couple of examples:

```
org.apache.commons.digester.Digester.level = FINEST
org.apache.solr.level = WARNING
```

Alternately, if you wish to change Tomcat's JDK Logging API settings for every application in this instance of Tomcat, edit `tomcat/conf/logging.properties`.

See the documentation for the SLF4J Logging API for more information:

http://slf4j.org/docs.html

9.4.2.2 Jetty Logging Settings

To change settings for the SLF4J Logging API in Jetty, you need to create a settings file and tell Jetty where to find it.

Begin by creating a file `jetty/logging.properties`. Use the example lines above as a guide.

To tell Jetty how to find the file, edit `start.sh`. Find the line which launches Jetty, which looks something like this, except it will have an absolute path to `start.jar`:

```
java -DSTOP.PORT=8079 -DSTOP.KEY=secret -jar start.jar
```

Add the location of the logging properties file like this:

```
java -Djava.util.logging.config.file=logging.properties
     -DSTOP.PORT=8079 -DSTOP.KEY=secret -jar start.jar
```

The next time you launch Jetty, it will use the settings in the file.

9.5 LucidGaze for Solr

LucidGaze is a tool that displays statistics about your Solr instance. It collects information about how Solr is used and how fast it responds. You choose which request handlers to examine and can view the collected data in a few different ways.

SolrWorks includes a fully functioning basic version of LucidGaze. The free version can monitor one running Solr instance. A payware version, LucidGaze Professional for Solr, is capable of monitoring a distributed Solr deployment. For more information, consult the Lucid Imagination web site: http://www.lucidimagination.com/

9.5.1 Running LucidGaze

Assuming you already have SolrWorks installed, running LucidGaze is easy. Just open up `solrworks/solr/conf/solrconfig.xml`. Find the line that defines the `solrgaze` request handler. Remove the comment marks `<!--` and `-->` from this line. When you're done, it should look like this (except all on one line):

```
<requestHandler
  name="/solrgaze" class="com.lucidimagination.gaze.plugin.StatMonitor" />
```

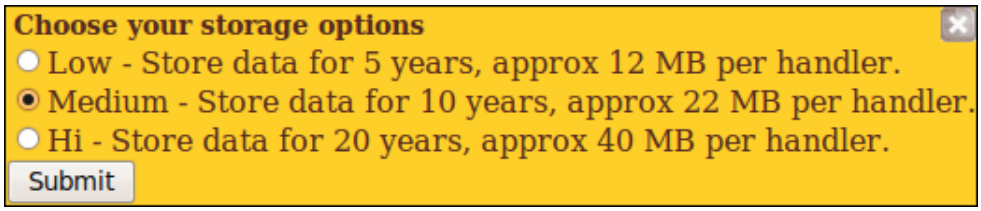
Save `solrconfig.xml` and start up Solr. LucidGaze is running!

In a browser, navigate to the LucidGaze home page:

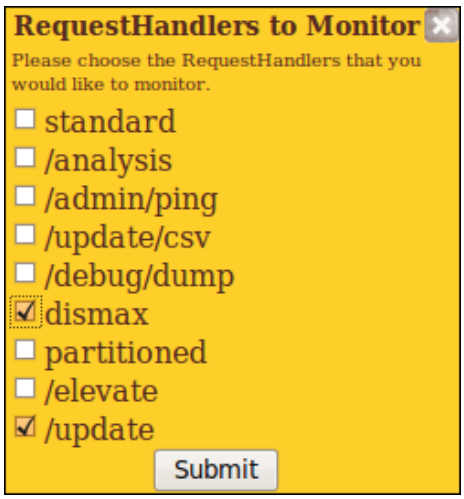
http://localhost:8983/gaze/

To get running, you have to answer a few questions. First, LucidGaze asks you for the URL of your Solr server. In most cases, this will already be filled in correctly for you.

After that, choose how long you want to keep your statistics. The data stored varies in resolution. A data point may reflect few seconds or a few weeks of statistics, depending on how far in the past the data reaches.



Finally, check off the request handlers you want to examine.



After clicking **Submit**, you'll see the main LucidGaze screen.

9.5.2 Monitoring Solr with LucidGaze

The main LucidGaze screen shows two graphs per request handler. One graph displays the number of requests per second, while the second displays the number of milliseconds per request.

On the detailed graph window, you can view the available data in three ways:

- **Day** shows the data for a single day.
- **Rng** allows you to choose a range of days.

- **Live** shows the specified interval up until the present. For example, you can look at the last thirty minutes or the last two hours.

9.6 Backing Up

If you're worried about data loss, and of course you *should* be, you need a way to back up your Solr indexes so that you can recover quickly in case of catastrophic failure.

9.6.1 Making Backups with the Solr Replication Handler

The easiest way to make back-ups in Solr 1.4 is to take advantage of the Replication Handler, which is described in detail in Chapter 10. The Replication Handler's primary purpose is to replicate an index on slave servers for load-balancing, but the Replication Handler can be used to make a back-up copy of a server's index, even if no slave servers are in operation.

Once you've configured the Replication Handler in `solrconfig.xml`, you can trigger a back-up with an HTTP command like this:

```
http://master_host/solr/replication?command=backup
```

For details on configuring the Replication Handler, see Chapter 10.

9.6.2 Backup Scripts from Earlier Solr Releases

Solr 1.4 also provides shell scripts in the `bin` directory that make copies of the indexes. However, these scripts only work with a Linux-style shell, and not everybody in the world runs Linux.

The scripts themselves are relatively simple. Look in the `bin` directory of your Solr home directory, for example `example/solr/bin`. In particular, `backup.sh` makes a copy of Solr's index directory and gives it a name based on the current date.

This scripts include the following:

Script Name	Description
`abc`	Atomic Backup post-Commit tells the Solr server to perform a commit. A snapshot of the index directory is made after the commit if the Solr server is configured to do so (by enabling the **postCommit** event listener in `solr/conf/solrconfig.xml`). A backup of the most recent snapshot directory is then made if the commit is successful. Backup directories are named `backup.`*`yyyymmddHHMMSS`* where *`yyyymmddHHMMSS`* is the timestamp of when the snaphot was taken.
`abo`	Atomic Backup post-Optimize tells the Solr server to perform an optimize. A snapshot of the index directory is made after the optimize if the Solr server is configured to do so (by enabling the **postCommit** or **postOptimize** event listener in `solr/conf/solrconfig.xml`). A backup of the most recent snapshot directory is then made if the optimize is successful. Backup directories are named `backup.`*`yyyymmddHHMMSS`* where *`yyyymmddHHMMSS`* is the timestamp of when the snaphot was taken.
`backup`	Backs up the index directory using hard links. Backup directories are named `backup.`*`yyyymmddHHMMSS`* where *`yyyymmddHHMMSS`* is the timestamp of when the backup was taken.
`backupcleaner`	Runs as a cron job to remove backups more than a configurable number of days old or all backups except for the most recent n number of backups. Also can be run manually.

For a details about backup scripts, see the Wiki page:

http://wiki.apache.org/solr/SolrOperationsTools

9.7 Using JMX with Solr

Java Management Extensions (JMX) is a technology that makes it possible for complex systems to be controlled by tools without the systems and tools having any previous knowledge of each other. In essence, it is a standard interface by which complex systems can be viewed and manipulated.

Solr, like any other good citizen of the Java universe, can be controlled via a JMX interface. You can enable JMX support by adding lines to `solrconfig.xml`. You can use a JMX client, like `jconsole`, to connect with Solr. Check out the Wiki page http://wiki.apache.org/solr/SolrJmx for more information.

You may also find the following overview of JMX to be useful:
http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html

You also need to enable JMX support when you start Tomcat. To do this, modify the `JAVA_OPTS` line in the file *lucidworks/lucidworks.sh* to read like this:

```
JAVA_OPTS="$SERVER $HEAPSIZE  $JMX -Dsolr.solr.home=$SOLR_HOME
-Dcom.sun.management.jmxremote "
```

9.8 Summary

In this chapter, you learned how to manage Solr through its Java Web container (either Tomcat or Jetty), how to collect statistics with Lucid Imagination's LucidGaze monitoring tool, how to run scripts to back up your Solr implementation, and how to use Java Management Extensions to manage your Solr implementation.

10 Scaling and Distribution

10.1 Introduction

Both Lucene and Solr were designed to scale to support large implementations with minimal custom coding.

This chapter covers:

- distributing an index across multiple servers
- replicating an index on multiple servers
- merging indexes

10.1.1 What Problem Does Distribution Solve?

If searches are taking too long or the index is approaching the physical limitations of its machine, you should consider distributing the index across two or more Solr servers.

To distribute an index, you divide the index into partitions called shards, each of which runs on a separate machine. Solr then partitions searches into sub-searches, which run on the individual shards, reporting results collectively. The architectural details underlying index sharding are invisible to end users, who simply experience faster performance on queries against very large indexes.

10.1.2 What Problem Does Replication Solve?

Replicating an index is useful when:

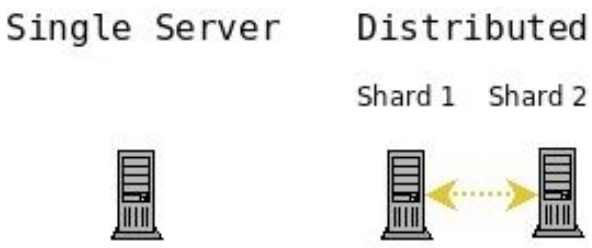
- You have a large search volume which one machine can't handle, so you need to distribute searches across multiple read-only copies of the index.
- There is a high volume/high rate of indexing which consumes machine resources and reduces search performance on the indexing machine, so you need to separate indexing and searching.
- You want to make a backup of the index (see Chapter 9).

10.2 Distributed Search with Index Sharding

10.2.1 Overview

When an index becomes too large to fit on a single system, or when a query takes too long to execute, an index can be split into multiple shards, and Solr can query and merge results across those shards. A single shard receives the query, distributes the query to other shards, and integrates the results.

The figure below compares a single server to a distributed configuration with two shards.



NOTE: If single queries are currently fast enough and one simply wishes to expand the capacity (queries/sec) of the search system, then standard index replication (replicating the entire index on multiple servers) should be used instead of index sharding.

10.2.2 Distributing Documents across Shards

It's up to you to get all your documents indexed on each shard of your server farm. Solr does not include out-of-the-box support for distributed indexing, but your method can be as simple as a round robin technique. Just index each document to the next server in the circle. (For more information about indexing, see Chapter 6.)

A simple hashing system would also work. The following should serve as an adequate hashing function.

```
uniqueId.hashCode() % numServers
```

One advantage of this approach is that it's easy to know where a document is if you need to update it or delete. In contrast, if you're moving documents around in a round-robin fashion, you may not know where a document actually is..

Solr does not calculate universal term/doc frequencies. For most large-scale implementations, it's not likely to matter that Solr calculates TD/IDF at the shard level. However, if your collection is heavily skewed in its distribution across servers, you may find misleading relevancy results in your searches. In general, it's probably best to randomly distribute documents to your shards.

10.2.3 Executing Distributed Searches with the shards Parameter

If a query request includes the `shards` parameter, the Solr server distributes the request across all the shards listed as arguments to the parameter. The `shards` parameter uses this syntax:

```
host:port/base_url[,host:port/base_url]*
```

For example, the `shards` parameter below causes the search to be distributed across two Solr servers: `solr1` and `solr2`, both of which are running on port 8983:

```
http://solr1:8983/solr/select?
shards=solr1:8983/solr,solr2:8983/solr&indent=true&q=ipod+solr
```

Rather than require users to include the `shards` parameter explicitly, it's usually preferred to configure this parameter as a default in the RequestHandler section of `solrconfig.xml`.

NOTE: Do not add the `shards` parameter to the standard requestHandler; otherwise, search queries may enter an infinite loop. Instead, define a new requestHandler that uses the `shards` parameter, and pass distributed search requests to that handler.

Currently, only query requests are distributed. This includes requests to the standard request handler (and subclasses such as the DisMax RequestHandler), and any other handler (`org.apache.solr.handler.component.searchHandler`) using standard components that support distributed search.

The following components support distributed search:

- The Query component, which returns documents matching a query
- The Facet component, which processes `facet.query` and `facet.field` requests where facets are sorted by count (the default).
- The Highlighting component, which enables Solr to include "highlighted" matches in field values.
- The Stats component, which returns simple statistics for numeric fields within the DocSet.
- The Debug component, which helps with debugging.

10.2.4 Limitations to Distributed Search

Distributed searching in Solr has the following following limitations:

- Each document indexed must have a unique key.
- If Solr discovers duplicate document IDs, Solr selects the first document and discards subsequent ones.
- Inverse-document frequency (IDF) calculations[17] cannot be distributed.
- Distributed searching does not support the QueryElevationComponent, which configures the top results for a given query regardless of Lucene's scoring.[18]

17 "The **tf–idf** weight (term frequency–inverse document frequency) is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf–idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query." "tf-idf," *Wikipedia,* retrieved on June 8, 2009

18 For more information, please see: http://wiki.apache.org/solr/QueryElevationComponent

- The index for distributed searching may become out of date; e.g., a document that once matched a query and was subsequently changed may no longer match the query but will still be retrieved.
- Distributed searching supports only sorted-field faceting, not date faceting
- The number of shards is limited by number of characters allowed for GET method's URI; most Web servers generally support at least 4000 characters, but many servers limit URI length to reduce their vulnerability to Denial of Service (DoS) attacks.
- TF/IDF computations are per shard. This may not matter if content is well (randomly) distributed.

10.2.5 Avoiding Distributed Deadlock

Each shard may also serve top-level query requests and then make sub-requests to all of the other shards. In this configuration, care should be taken to ensure that the max number of threads serving HTTP requests in the servlet container is greater than the possible number of requests from both top-level clients and other shards. If this is not the case, the configuration may result in a distributed deadlock.

Here's how a deadlock might occur. Consider the simplest case of two shards, each with just a single thread to service HTTP requests. Both threads could receive a top-level request concurrently, and make sub-requests to each other. Because there are no more remaining threads to service requests, the servlet containers will block the incoming requests until the other pending requests are finished, but they won't finish since they are waiting for the sub-requests. By ensuring that the servlets are configured to handle a sufficient number of threads, you can avoid deadlock situations like this.

10.2.6 Testing Index Sharding on Two Local Servers

For simple functionality testing, it's easiest to just set up two local Solr servers on different ports. (In a production environment, of course, these servers would be deployed on separate machines.)

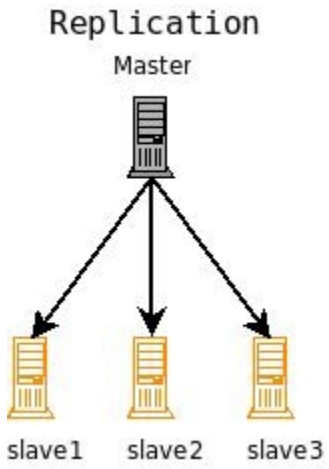
```
#make a copy of the solr example directory
cd solr
cp -r example example7574

#change the port number
perl -pi -e s/8983/7574/g example7574/etc/jetty.xml
example7574/exampledocs/post.sh

#in window 1, start up the server on port 8983
cd example
java -server -jar start.jar

#in window 2, start up the server on port 7574
cd example7574
java -server -jar start.jar

#in window 3, index some example documents to each server
cd example/exampledocs
./post.sh [a-m]*.xml
cd ../../example7574/exampledocs
./post.sh [n-z]*.xml

#now do a distributed search across both servers with your browser or curl
curl 'http://localhost:8983/solr/select?
shards=localhost:8983/solr,localhost:7574/solr&indent=true&q=ipod+solr'
```

10.3 Index Replication

10.3.1 Overview of Index Replication

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

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



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

10.3.2 Index Replication in Solr 1.4

Solr 1.4 introduces a new Java implementation of index replication that works over HTTP .

> **NOTE:** For information on the `ssh/rsync` based replication available since Solr 1.1, please see page 331.

The new Java-based implementation of index replication offers these benefits:

- Replication without requiring external scripts
- The configuration affecting replication is controlled by a single file, `solrconfig.xml`

- Supports the replication of configuration files as well as index files
- Works across platforms with same configuration
- No reliance on OS-dependent hard links
- Tightly integrated with Solr; an admin page offers fine-grained control of each aspect of replication

The new Java-based replication feature is implemented as a RequestHandler. Configuring replication is therefore similar to any normal RequestHandler.

10.3.3 Configuring the Replication RequestHandler on a Master Server

The example below shows how to configure the Replication RequestHandler on a master server.

```
<requestHandler name="/replication" class="solr.ReplicationHandler" >
    <lst name="master">
        <!--Replicate on 'optimize'. Other values can be 'commit',
'startup'. It is possible to have multiple entries of this config string--
>
        <str name="replicateAfter">optimize</str>

        <!--Create a backup after 'optimize'. Other values can be
'commit', 'startup'. It is possible to have multiple entries of this
config string.  Note that this is just for backup, replication does not
require this. -->
        <!-- <str name="backupAfter">optimize</str> -->

        <!--If configuration files need to be replicated give the names
here, separated by comma -->
        <str name="confFiles">schema.xml,stopwords.txt,elevate.xml</str>
       <!--The default value of reservation is 10 secs.See the
documentation below . Normally , you should not need to specify this -->
        <str name="commitReserveDuration">00:00:10</str>
    </lst>
</requestHandler>
```

Note:

- If your commits are very frequent and network is particularly slow, you can tweak an extra attribute `<str name="commitReserveDuration">00:00:10</str>`. This is roughly the time taken to download 5MB from master to slave. Default is 10 secs.

- If you are using **startup** option for *replicateAfter*, it is necessary to have a **commit**/**optimize** entry also, if you want to trigger replication on future commits/optimizes. If only the **startup** option is given, replication will not be triggered on subsequent commits/optimizes after it is done for the first time at the start.

10.3.3.1 Replicating solrconfig.xml

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

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

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

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

10.3.3.2 Configuring the Replication RequestHandler on a Slave Server

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

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

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

        <!--Interval in which the slave should poll master .Format is
HH:mm:ss . If this is absent slave does not poll automatically.
         But a fetchindex can be triggered from the admin or the http API
-->
        <str name="pollInterval">00:00:20</str>
        <!-- THE FOLLOWING PARAMETERS ARE USUALLY NOT REQUIRED-->
        <!--to use compression while transferring the index files. The
possible values are internal|external
         if the value is 'external' make sure that your master Solr has
the settings to honor the accept-encoding header.
```

```
         see here for details
http://wiki.apache.org/solr/SolrHttpCompression
         If it is 'internal' everything will be taken care of
automatically.
         USE THIS ONLY IF YOUR BANDWIDTH IS LOW . THIS CAN ACTUALLY
SLOWDOWN REPLICATION IN A LAN-->
        <str name="compression">internal</str>
        <!--The following values are used when the slave connects to the
master to download the index files.
         Default values implicitly set as 5000ms and 10000ms respectively.
The user DOES NOT need to specify
         these unless the bandwidth is extremely low or if there is an
extremely high latency-->
        <str name="httpConnTimeout">5000</str>
        <str name="httpReadTimeout">10000</str>

        <!-- If HTTP Basic authentication is enabled on the master, then
the slave can be configured with the following -->
        <str name="httpBasicAuthUser">username</str>
        <str name="httpBasicAuthPassword">password</str>

     </lst>
</requestHandler>
```

NOTE: If you are not using cores, then you simply omit the `corename` parameter above in the `masterUrl`. To ensure that the URL is correct, just hit the URL with a browser. You must get a status OK response.

10.3.3.3 Setting Up a Repeater with the ReplicationHandler

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

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

- Optionally, one can configure the repeater to fetch compressed files from the master through the compression parameter (see the sample configuration code on page 324for details) to reduce the index download time.

Here's an example of a ReplicationHandler configuration for a repeater:

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

10.3.3.4 Commit and Optimize Operations

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

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

The replicateAfter parameter can accept multiple arguments. For example:

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

10.3.3.5 Slave Replication

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

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

10.3.3.6 Replicating Configuration Files

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

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

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

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

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

10.3.3.7 Resolving Corruption Issues on Slave Servers

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

10.3.3.8 HTTP API Commands for the ReplicationHandler

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

Command	Description
`http://master_host:port/solr/replication?command=enablereplication`	Enables replication on the master for all its slaves.
`http://master_host:port/solr/replication?command=disablereplication`	Disables replication on the master for all its slaves.
`http://host:port/solr/replication?command=indexversion`	Returns the version of the latest replicatable index on the specified master or slave

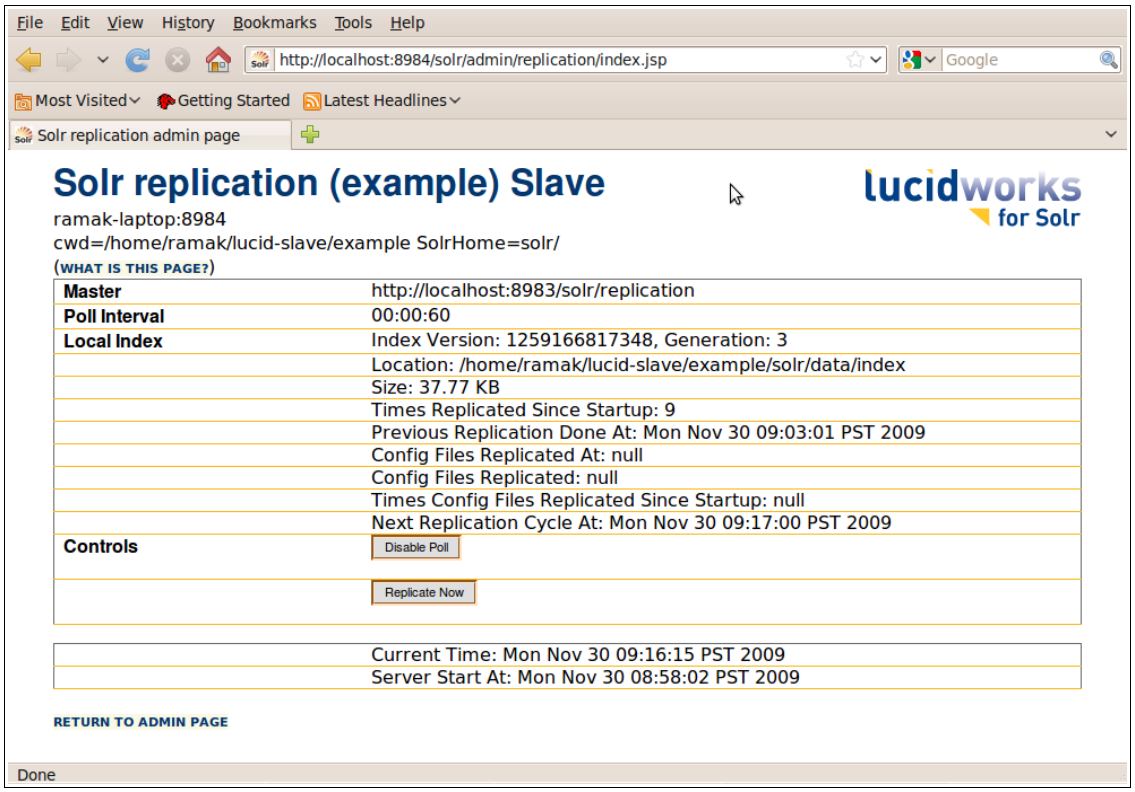
Command	Description
`http://slave_host:port/solr/replication?command=fetchindex`	Forces the specified slave to fetch a copy of the index from its master. If you like, you can pass an extra attribute such as `masterUrl` or `compression` (or any other parameter which is specified in the `<lst name="slave">` tag) to do a one time replication from a master. This obviates the need for hard-coding the master in the slave.
`http://slave_host:port/solr/replication?command=abortfetch`	Aborts copying an index from a master to the specified slave.
`http://slave_host:port/solr/replication?command=enablepoll`	Enables the specified slave to poll for changes on the master.
`http://slave_host:port/solr/replication?command=disablepoll`	Disables the specified slave from polling for changes on the master.
`http://slave_host:port/solr/replication?command=details`	Retrieves configuration details and current status.
`http://host:port/solr/replication?command=filelist&indexversion=<index-version-number>`	Retrieves a list of Lucene files present in the specified host's index. You can discover the version number of the index by running the `indexversion` command.
`http://master_host:port/solr/replication?command=backup`	Creates a backup on master if there are committed index data in the server; otherwise, does nothing. This command is useful for making periodic backups.

10.3.3.9 Using the Replication Dashboard

The Solr Replication Dashboard, which is accessible through the Distribution link on the Admin Web interfaces, shows the following information related to replication managed through the Replication Handler:

- status of current replication
- percentage/size downloaded/to be downloaded
- the name of the current file being downloaded
- the time taken compared to the time remaining

The figure below shows the Replication Dashboard for a slave server.



The Replication Dashboard reports details of the master-slave configuration and offers controls for managing the replication.

You can perform the following actions from the Replication Dashboard:

- Enable/Disable polling
- Force-start replication (sometimes useful for making a backup copy of an index)
- Abort an ongoing replication process

10.3.4 Index Replication using ssh and rsync

Since Solr 1.1, Solr has supported `ssh/rsync`-based replication. *This mechanism only works on systems that support removing open hard links.*

Solr distribution is similar in concept to database replication. All collection changes come to one master Solr server. All production queries are done against query slaves. Query slaves receive all their collection changes indirectly — as new versions of a collection which they pull from the master. These collection downloads are polled for on a cron'd basis.

A collection is a directory of many files. Collections are distributed to the slaves as snapshots of these files. Each snapshot is made up of hard links to the files so copying of the actual files is not necessary when snapshots are created. Lucene only *significantly* rewrites files following an optimization command. Generally, once a file is written, it will change very little, if at all. This makes the underlying transport of `rsync` very useful. Files that have already been transferred and have not changed do not need to be re-transferred with the new edition of a collection.

10.3.4.1 Replication Terminology

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

Term	Definition
Collection	A Lucene collection is a directory of files. These files make up the indexed and returnable data of a Solr search repository.
Distribution	The copying of a collection from the master server to all slaves. The distribution process takes advantage of Lucene's index file structure.

Term	Definition
Inserts and Deletes	As inserts and deletes occur in the collection, the directory remains unchanged. Documents are always inserted into newly created files. Documents that are deleted are not removed from the files. They are flagged in the file, deletable, and are not removed from the files until the collection is optimized.
Master and Slave	The Solr distribution model uses the master/slave model. The master is the service which receives all updates initially and keeps everything organized. Solr uses a single update master server coupled with multiple query slave servers. All changes (such as inserts, updates, deletes, etc.) are made against the single master server. Changes made on the master are distributed to all the slave servers which service all query requests from the clients.
Update	An update is a single change request against a single Solr instance. It may be a request to delete a document, add a new document, change a document, delete all documents matching a query, etc. Updates are handled synchronously within an individual Solr instance.
Optimization	A process that compacts the index and merges segments in order to improve query performance. New secondary segment(s) are created to contain documents inserted into the collection after it has been optimized. A Lucene collection must be optimized periodically to maintain satisfactory query performance. Optimization is run on the master server only. An optimized index will give you a performance gain at query time of *at least* 10%. This gain may be more on an index that has become fragmented over a period of time with many updates and no optimizations. Optimizations require a **much** longer time than does the distribution of an optimized collection to all slaves.
Segments	The number of files in a collection.
`mergeFactor`	A parameter that controls the number of files (segments) in a collection. For example, when `mergeFactor` is set to 3, Solr will fill one segment with documents until the limit `maxBufferedDocs` is met, then it will start a new segment. When the number of segments specified by `mergeFactor` is reached—in this example, 3—then Solr will merge all the segments into a single index file, then begin writing new documents to a new segment.
Snapshot	A directory containing hard links to the data files. Snapshots are distributed from the master server when the slaves pull them, "smartcopying" the snapshot directory that contains the hard links to the most recent collection data files.

10.3.4.2 The Snapshot and Distribution Process

Here are the steps that Solr follows when replicating an index:

1. The **snapshooter** command takes snapshots of the collection on the master. It runs when invoked by Solr after it has done a commit or an optimize.

2. The **snappuller** command runs on the query slaves to pull the newest snapshot from the master. This is done via rsync in daemon mode running on the master for better performance and lower CPU utilization over rsync using a remote shell program as the transport.

3. The **snapinstaller** runs on the slave after a snapshot has been pulled from the master. This signals the local Solr server to open a new index reader, then auto-warming of the cache(s) begins (in the new reader), while other requests continue to be served by the original index reader. Once auto-warming is complete, Solr retires the old reader and directs all new queries to the newly cache-warmed reader.

4. All distribution activity is logged and written back to the master to be viewable on the distribution page of its GUI.

5. Old versions of the index are removed from the master and slave servers by a cron'd **snapcleaner**.

If you are building an index from scratch, distribution is the final step of the process.

Manual copying of index files is not recommended; however, running distribution commands manually (i.e., not relying on `crond` to run them) is perfectly fine.

10.3.4.3 Snapshot Directories

Snapshots are stored in directories whose names follow this format: `snapshot.`*`yyyymmddHHMMSS`*

All the files in the index directory are hard links to the latest snapshot. This design offers these advantages:

- The Solr implementation can keep multiple snapshots on each host without needing to keep multiple copies of index files that have not changed.
- File copying from master to slave is very fast.
- Taking a snapshot is very fast as well.

10.3.4.4 Solr Distribution Scripts

For the Solr distribution scripts, the name of the index directory is defined either by the environment variable `data_dir` in the configuration file `solr/conf/scripts.conf` or the command line argument `-d`. It should match the value used by the Solr server which is defined in `solr/conf/solrconfig.xml`.

All Solr collection distribution scripts are bundled in a Solr release and reside in the directory `solr/src/scripts`. Lucid Imagination recommends that you install the scripts in a `solr/bin/` directory.

Collection distribution scripts create and prepare for distribution a snapshot of a search collection after each commit and optimize request if the *postCommit* and *postOptimize* event listener is configured in `solrconfig.xml` to execute **snapshooter**.

The **snapshooter** script creates a directory `snapshot.<ts>`, where `<ts>` is a timestamp in the format, *yyyymmddHHMMSS*. It contains hard links to the data files.

Snapshots are distributed from the master server when the slaves pull them, "smartcopying" the snapshot directory that contains the hard links to the most recent collection data files.

Name	Description
`snapshooter`	Creates a snapshot of a collection. Snapshooter is normally configured to run on the master Solr server when a commit or optimize happens. `snapshooter` can also be run manually, but one must make sure that the index is in a consistent state, which can only be done by pausing indexing and issuing a commit.

Name	Description
`snappuller`	A shell script that runs as a cron job on a slave Solr server. The script looks for new snapshots on the master Solr server and pulls them.
`snappuller-enable`	Creates the file, `solr/logs/snappuller-enabled`, whose presence enables `snappuller`.
`snapinstaller`	Installs the latest snapshot (determined by the timestamp) into the place, using hard links (similar to the process of taking a snapshot). Then `solr/logs/snapshot.current` is written and scp'd (secure copied) back to the master Solr server. `snapinstaller` then triggers the Solr server to open a new Searcher.
`snapcleaner`	Runs as a cron job to remove snapshots more than a configurable number of days old or all snapshots except for the most recent *n* number of snapshots. Also can be run manually.
`rsyncd-start`	Starts the `rsyncd` daemon on the master Solr server which handles collection distribution requests from the slaves.
`rsyncd daemon`	Efficiently synchronizes a collection—between master and slaves—by copying only the files that actually changed. In addition, `rsync` can optionally compress data before transmitting it.
`rsyncd-stop`	Stops the `rsyncd` daemon on the master Solr server. The stop script then makes sure that the daemon has in fact exited by trying to connect to it for up to 300 seconds. The stop script exits with *error code 2* if it fails to stop the `rsyncd` daemon.
`rsyncd-enable`	Creates the file, `solr/logs/rsyncd-enabled`, whose presence allows the `rsyncd` daemon to run, allowing replication to occur.
`rsyncd-disable`	Removes the file, `solr/logs/rsyncd-enabled`, whose absence prevents the `rsyncd` daemon from running, preventing replication.

For more information about usage arguments and syntax see the SolrCollectionDistributionScripts page on the Wiki.

10.3.4.5 Solr Distribution-related Cron Jobs

The distribution process is automated through the use of `cron` jobs. The `cron` jobs should run under the user ID that the Solr server is running under.

Cron Job	Description
`snapcleaner`	The `snapcleaner` job should be run out of `cron` at the regular basis to clean up old snapshots. This should be done on both the master and slave Solr servers. For example, the following `cron` job runs everyday at midnight and cleans up snapshots 8 days and older: `0 0 * * * <solr.solr.home>/solr/bin/snapcleaner -D 7` Additional cleanup can always be performed on-demand by running `snapcleaner` manually.
`snappuller` `snapinstaller`	On the slave Solr servers, `snappuller` should be run out of `cron` regularily to get the latest index from the master Solr server. It is a good idea to also run `snapinstaller` with `snappuller` back-to-back in the same `crontab` entry to install the latest index once it has been copied over to the slave Solr server.

For example, the following `cron` job runs every 5 minutes to keep the slave Solr server in sync with the master Solr server:

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * *
<solr.solr.home>/solr/bin/snappuller;<solr.solr.home>/solr/bin/snapinstaller
```

NOTE: Modern `cron` allows this to be shortened to `*/5 * * * * ...`.

10.3.4.6 Commit and Optimization

On a very large index, adding even a few documents then running an optimize operation causes the complete index to be rewritten. This consumes a lot of disk I/O and impacts query performance. Optimizing a very large index may even involve copying the index twice and calling optimize at the beginning *and* at the end. If some documents have been deleted, the first optimize call will rewrite the index even before the second index is merged.

Optimization is an I/O intensive process, as the entire index is read and re-written in optimized form. Anecdotal data shows that optimizations on modest server hardware can take around 5 minutes per GB, although this obviously varies considerably with index fragmentation and hardware bottlenecks. We do not know what happens to query performance on a collection that has not been optimized for a long time.

We *do* know that it will get worse as the collection becomes more fragmented, but how much worse is very dependent on the manner of updates and commits to the collection. The setting of the mergeFactor attribute affects performance as well. Dividing a large index with millions of documents into even as few as five segments may degrade search performance by as much as 15-20%.

We are presuming optimizations should be run once following large *batch-like* updates to the collection and/or once a day.

10.3.4.7 Distribution and Optimization

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

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

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

Optimizing on the master allows for a straight-forward optimization operation. No query slaves need to be taken out of service. The optimized collection can be distributed in the background as queries are being

normally serviced. The optimization can occur at any time convenient to the application providing collection updates.

10.3.5 Performance Tuning for Script-based Replication

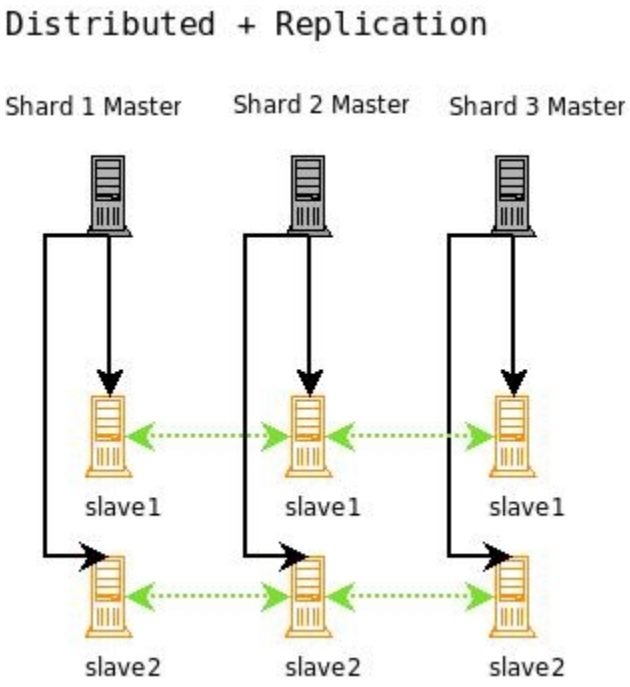
Because fetching a master index uses the `rsync` utility, which transfers only the segments that have changed, replication is normally very fast. However, if the master server has been optimized, then `rsync` may take a long time, because many segments will have been changed in the process of optimization.

- If replicating to multiple slaves consumes too much network bandwidth, consider the use of a repeater.
- Make sure that slaves do not pull from the master so frequently that a previous replication is still running when a new one is started. In general, it's best to allow at least a minute for the replication process to complete. But in configurations with low network bandwidth or a very large index, even more time may be required.

10.4 Combining Distribution and Replication

When your index is too large for a single machine and you have a query volume that single shards cannot keep up with, it's time to replicate each shard in your distributed search setup.

The idea is to combine distributed search with replication. As shown in the figure below, a combined distributed-replication configuration features a master server for each shard and then 1-*n* slaves that are replicated from the master. As in a standard replicated configuration, the master server handles updates and optimizations without adversely affecting query handling performance. Query requests should be load balanced across each of the shard slaves. This gives you both increased query handling capacity and fail-over backup if a server goes down.



A Solr configuration combining both replication and master-slave distribution.

None of the master shards in this configuration know about each other. You index to each master, the index is replicated to each slave, and then searches are distributed across the slaves, using one slave from each master/slave shard.

For high availability you can use a load balancer to set up a virtual IP for each shard's set of slaves. If you are new to load balancing, HAProxy (http://haproxy.1wt.eu/) is a good open source software load-balancer. If a slave server goes down, a good load-balancer will detect the failure using some technique (generally a heartbeat system), and forward all requests to the remaining live slaves that served with the failed slave. A single virtual IP should then be set up so that requests can hit a single IP, and get load balanced to each of the virtual IPs for the search slaves.

With this configuration you will have a fully load balanced, search-side fault-tolerant system (Solr does not yet support fault-tolerant indexing). Incoming searches will be handed off to one of the functioning slaves, then the slave will distribute the search request across a slave for each of the shards in your configuration. The slave will issue a request to each of the virtual IPs for each shard, and the load balancer will choose one of the available slaves. Finally, the results will be combined into a single results set and returned. If any of the slaves go down, they will be taken out of rotation and the remaining slaves will be used. If a shard master goes down, searches can still be served from the slaves until you have corrected the problem and put the master back into production.

10.5 Merging Indexes

If you need to combine indexes from two different projects or from multiple servers previously used in a distributed configuration, you can use the IndexMergeTool included in `lucene-misc`.

To merge indexes, they must meet these requirements:

- The two indexes must be compatible: their schemas should include the same fields and they should analyze fields the same way.
- The indexes must not include duplicate data.

Optimally, the two indexes should be built using the same schema.

To merge the indexes, do the following:

Find the `lucene` JAR file that your version of Solr is using. You can do this by copying your `solr.war` file somewhere and unpacking it (`jar xvf solr.war`). Your `lucene` JAR file

should be in `WEB-INF/lib`. It is probably called something like `lucene-core-2007-05-20_00-04-53.jar`. Copy it somewhere easy to find.

Download a copy of Lucene from `www.lucidimagination.com/Downloads` and unpack it. The file you're interested in is `contrib/misc/lucene-misc-VERSION.jar`

Make sure that both indexes you want to merge are closed.

Issue this command:

```
java -cp /path/to/lucene-core-VERSION.jar:/path/to/lucene-misc-VERSION.jar org/apac
he/lucene/misc/IndexMergeTool /path/to/newindex /path/to/index1 /path/to/index2
```

This will create a new index at `/path/to/newindex` that contains both `index1` and `index2`. Copy this new directory to the location of your application's solr index (move the old one aside first, of course) and start Solr. For example:

```
java -cp /tmp/lucene-core-2007-05-20_00-04-53.jar:./lucene-
2.2.0/contrib/misc/lucene-misc-2.2.0.jarorg/apache/lucene/misc/IndexMergeTool
./newindex ./app1/solr/data/index ./app2/solr/data/index
```

10.6 Summary

This chapter described different ways you can scale a Solr installation to accelerate performance or to manage large indexes or data volumes.

- **Index distribution** splits a large index into multiple shards, each of which runs on a separate server. A single shard accepts a query, distributes it among all the shards, and integrates the results. Index distribution enables Solr to quickly and efficiently process queries against a large index. If necessary, index shards can later be merged into a single intact index.

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

- Distribution and replication can be combined so that Solr can process queries against both large indexes and large data volumes.

11 Client APIs

11.1 Introduction

At its heart, Solr is a Web application, but because it is built on open protocols, any type of client application can use Solr.

HTTP is the fundamental protocol used between client applications and Solr. The client makes a request and Solr does some work and provides a response. Clients use requests to ask Solr to do things like perform queries or index documents.

Client applications can reach Solr by creating HTTP requests and parsing the HTTP responses. Client APIs encapsulate much of the work of sending requests and parsing responses, which makes it much easier to write client applications.

Clients use Solr's five fundamental operations to work with Solr. The operations are query, index, delete, commit, and optimize.

Queries are executed by creating a URL that contains all the query parameters. Solr examines the request URL, performs the query, and returns the results. The other operations are similar, although in certain cases the HTTP request is a POST operation and contains information beyond whatever is included in the request URL. An Index operation, for example, may contain a document in the body of the request.

Solr 1.4 also features an EmbeddedSolrServer that offers a Java API without requiring an HTTP connection. For details, see page 352.

11.2 Choosing an Output Format

Many programming environments are able to send HTTP requests and retrieve responses. Parsing the responses is a slightly more thorny problem. Fortunately, Solr makes it easy to choose an output format that will be easy to handle on the client side.

Specify a response format using the `wt` parameter in a query. The available response formats are documented in Chapter 7.

Most client APIs hide this detail for you, so for many types of client applications, you won't ever have to specify a `wt` parameter. In JavaScript, however, the interface to Solr is a little closer to the metal, so you will need to add this parameter yourself.

11.3 JavaScript is Really Easy

Using Solr from JavaScript clients is so straightforward that it deserves a special mention. In fact, it is so straightforward that there is no client API. You don't need to install any packages or configure anything.

HTTP requests can be sent to Solr using the standard `XMLHttpRequest` mechanism.

Out of the box, Solr can send JavaScript Object Notation (JSON) responses, which are easily interpreted in JavaScript. Just add `wt=json` to the request URL to have responses sent as JSON.

For more information and an excellent example, read the SolJSON page on the Solr Wiki:

http://wiki.apache.org/solr/SolJSON

11.4 Python is Pretty Darn Easy, Too

Solr includes an output format specifically for Python, but JSON output is a little more robust.

11.4.1 Plain Vanilla Python

Making a query is a simple matter. First, tell Python you will need to make HTTP connections.

```
from urllib2 import *
```

Now open a connection to the server and get a response. The `wt` query parameter tells Solr to return results in a format that Python can understand.

```
connection = urlopen(
                'http://localhost:8983/solr/select?q=cheese&wt=python')
response = eval(connection.read())
```

Now interpreting the response is just a matter of pulling out the information that you need.

```
print response['response']['numFound'], "documents found."

# Print the name of each document.
for document in response['response']['docs']:
  print "  Name =", document['name']
```

11.4.2 Kick it Up a Notch with JSON

JSON is a more robust response format, but you will need to add a Python package in order to use it. At a command line, install the `simplejson` package like this:

```
$ sudo easy_install simplejson
```

Once that is done, making a query is nearly the same as before. However, notice that the `wt` query parameter is now `json`, and the response is now digested by `simplejson.load()`.

```
from urllib2 import *
import simplejson

connection = urlopen('http://localhost:8983/solr/select?q=cheese&wt=json')
response = simplejson.load(connection)

print response['response']['numFound'], "documents found."

# Print the name of each document.
for document in response['response']['docs']:
  print "  Name =", document['name']
```

11.5 Client API Lineup

The Solr Wiki contains a list of client APIs:

http://wiki.apache.org/solr/IntegratingSolr

Here is the list of client APIs, current at this writing (June 2009):

Name	Environment	URL
SolRuby	Ruby	http://wiki.apache.org/solr/SolRuby
DelSolr	Ruby	http://delsolr.rubyforge.org/
acts_as_solr	Rails	http://acts-as-solr.rubyforge.org/ http://rubyforge.org/projects/background-solr/
Flare	Rails	http://wiki.apache.org/solr/Flare
SolPHP	PHP	http://wiki.apache.org/solr/SolPHP
SolrJ	Java	http://wiki.apache.org/solr/SolJava
Python API	Python	http://wiki.apache.org/solr/SolPython
PySolr	Python	http://code.google.com/p/pysolr/
SolPerl	Perl	http://wiki.apache.org/solr/SolPerl
Solr.pm	Perl	http://search.cpan.org/~garafola/Solr-0.03/lib/Solr.pm
SolrForrest	Forrest/Cocoon	http://wiki.apache.org/solr/SolrForrest
SolrSharp	C#	http://www.codeplex.com/solrsharp
SolColdfusion	ColdFusion	http://solcoldfusion.riaforge.org/

11.6 Using SolrJ

SolrJ (also sometimes known as SolJava) is an API that makes it easy for Java applications to talk to Solr. SolrJ hides a lot of the details of connecting to Solr and allows your application to interact with Solr with simple high-level methods.

The center of SolrJ is the `org.apache.solr.client.solrj` package, which contains just five main classes. Begin by creating a `SolrServer`, which represents the Solr instance you want to use. Send `SolrRequest`s or `SolrQuery`s and get back `SolrResponse`s.

`SolrServer` is abstract, so to connect to a remote Solr instance, you'll actually create an instance of `org.apache.solr.client.solrj.impl.CommonsHttpSolrServer`, which knows how to use HTTP to talk to Solr.

```
String urlString = "http://localhost:8983/solr";
SolrServer solr = new CommonsHttpSolrServer(urlString);
```

Creating a `SolrServer` doesn't make a network connection—that'll happen later when you perform a query or some other operation—but it will throw `MalformedURLException` if you give it a bad URL string.

Once you've got a `SolrServer`, you can make it dance by calling methods like `query()`, `add()`, and `commit()`.

11.6.1 Building and Running SolrJ Applications

The SolrJ API is included with Solr, so you don't have to download or install anything else. However, in order to build and run applications that use SolrJ, you have to add some libraries to the classpath.

At build time, the examples presented with this chapter require the following libraries in the classpath (all paths are relative to the root of the Solr installation).

```
apache-solr-common-1.4.0.jar
apache-solr-solrj-1.4.0.jar
```

At run time, the examples in this chapter require the following libraries.

```
apache-solr-common-1.4.0.jar
apache-solr-solrj-1.4.0.jar
solrj-lib/commons-httpclient-3.1.jar
solrj-lib/commons-logging-1.0.4.jar
solrj-lib/commons-codec-1.4.jar
```

The Ant script bundled with this chapter's examples includes the libraries as appropriate when building and running.

You can sidestep a lot of the messing around with the JAR files by using Maven instead of Ant. All you will need to do to include SolrJ in your application is to put the following dependency in the project's `pom.xml`:

```
<dependency>
   <groupId>org.apache.solr</groupId>
   <artifactId>solr-solrj</artifactId>
   <version>1.3.0</version>
</dependency>
```

If you are worried about the SolrJ libraries blowing up the size of your client application, you can use a code obfuscator like ProGuard to remove APIs that you are not using. ProGuard is available here:

http://proguard.sourceforge.net/

11.6.2 Setting XMLResponseParser

As of Solr 1.4, SolrJ uses a binary format, rather than XML, as its default format. Users of earlier Solr releases who wish to continue working with XML must explicitly set the parser to the XMLResponseParser, like so:

```
server.setParser(new XMLResponseParser());
```

11.6.3 Performing Queries

Use `query()` to have Solr search for results. You have to pass a `SolrQuery` object that describes the query, and you'll get back a `QueryResponse` (from the `org.apache.solr.client.solrj.response` package).

`SolrQuery` has methods that make it easy to add parameters to choose a request handler and send parameters to it. Here is a very simple example that uses the default request handler and sets the `q` parameter:

```
SolrQuery parameters = new SolrQuery();
parameters.set("q", mQueryString);
```

To choose a different request handler, for example, just set the `qt` parameter like this:

```
parameters.set("qt", "/spellCheckCompRH");
```

Once you have your `SolrQuery` set up just the way you like it, shoot it out with `query()`:

```
QueryResponse response = solr.query(parameters);
```

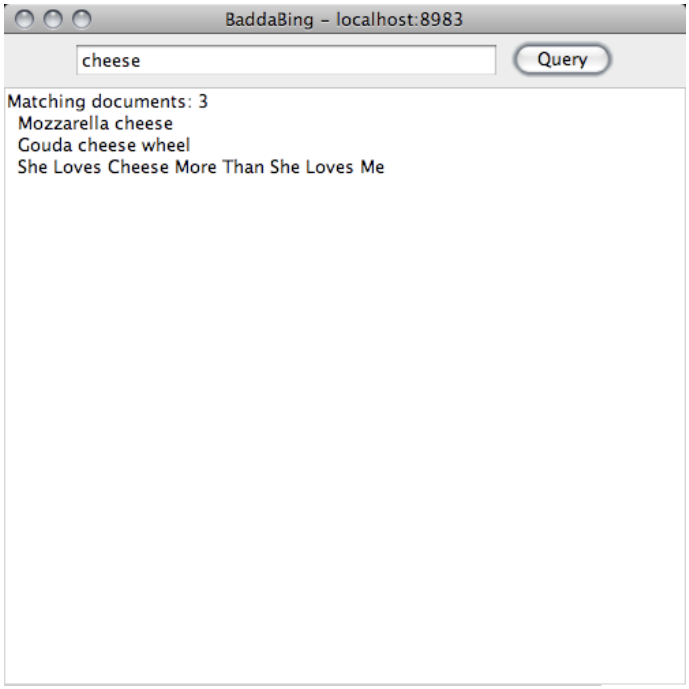
This is when the magic really happens: the client make a network connection, the query is sent, Solr chugs away, and the response is sent and parsed into a `QueryResponse`.

In essence, `QueryResponse` is a collection of documents that satisfy the query parameters. You can retrieve the documents directly with `getResults()` and you can call other informative methods to find out information about highlighting or facets.

```
SolrDocumentList list = response.getResults();
```

The example source code for this guide includes `BaddaBing`, a simple Swing interface for making queries. `BaddaBing` contains the user interface code, but all the Solr stuff is encapsulated in `BaddaBingWorker`.

If you haven't added any documents to your instance of Solr, of course, you won't be able to find any documents. In that case, keep reading to find out how to add documents.



11.6.4 Indexing Documents

Other operations are just as simple. To index (add) a document, all you need to do is create a `SolrInputDocument` and pass it along to the `SolrServer`'s `add()` method.

```
String urlString = "http://localhost:8983/solr";
SolrServer solr = new CommonsHttpSolrServer(urlString);

SolrInputDocument document = new SolrInputDocument();
document.addField("id", "552199");
document.addField("name", "Gouda cheese wheel");
document.addField("price", "49.99");

UpdateResponse response = solr.add(document);
```

Don't forget to commit your changes!

```
solr.commit();
```

11.6.5 Uploading Content in XML or Binary Formats

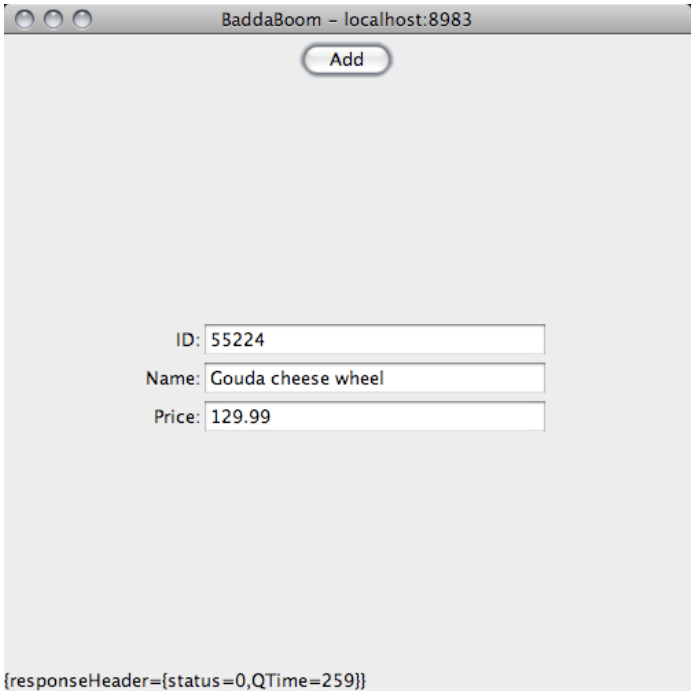
SolrJ lets you upload content in XML and binary formats instead of the default XML format. Use the following to upload using Binary format. this is the same format which SolrJ uses to fetch results.

```
server.setRequestWriter(new BinaryRequestWriter());
```

11.6.6 Trying out SolrJ with BaddaBoom and BaddaBing

The example code includes a simple application, `BaddaBoom`, for adding documents. You can add documents with `BaddaBoom` and search for them with `BaddaBing`.

`BaddaBoom` contains mainly user interface code, and `BaddaBoomWorker` is where all the Solr work happens.



11.6.7 EmbeddedSolrServer

The EmbeddedSolrServer provides the Java interface described above without requiring an HTTP connection. This is the recommended approach if you need to use Solr in an embedded application. This approach enables you to work with the same Java interface whether or not you have access to HTTP.

> **NOTE:** EmbeddedSolrServer works only with handlers registered in `solrconfig.xml`. RequestHandler must be mapped to `/update` for a request to `/update` to function. For information about configuring handlers in `solrconfig.xml`, see Chapter 8.

```
  // Note that the following property could be set through JVM level
arguments too
  System.setProperty("solr.solr.home",
"/home/shalinsmangar/work/oss/branch-1.3/example/solr");
  CoreContainer.Initializer initializer = new CoreContainer.Initializer();
  CoreContainer coreContainer = initializer.initialize();
  EmbeddedSolrServer server = new EmbeddedSolrServer(coreContainer, "");
```

If you want to use MultiCore features (which are described in Chapter 8), then you should use this:

```
    File home = new File( "/path/to/solr/home" );
    File f = new File( home, "solr.xml" );
    CoreContainer container = new CoreContainer();
    container.load( "/path/to/solr/home", f );

    EmbeddedSolrServer server = new EmbeddedSolrServer( container, "core
name as defined in solr.xml" );
    ...
```

11.6.8 Using the StreamingUpdateSolrServer

If you're working with Java, you can take advantage of the StreamingUpdateSolrServer to perform bulk updates at high speed. StreamingHttpSolrServer buffers all added documents and writes them into open HTTP connections. This class is thread safe. Although any SolrServer request can be made with this implementation, it is only recommended to use the StreamingUpdateSolrServer for `/update` requests.

You can learn more about the StreamingUpdateSolrServer here:

lucene.apache.org/solr/api/org/apache/solr/client/solrj/impl/StreamingUpdateSolrServer.html

11.6.9 More Information

As you begin developing with SolrJ, you will find the API documentation indispensable. It is available online at the Apache Lucene site:

http://lucene.apache.org/solr/api/solrj/index.html

For more information about using SolrJ, read the page at the Solr Wiki:

http://wiki.apache.org/solr/Solrj

The Solr Wiki also contains another example which demonstrates setting `qt`:

http://wiki.apache.org/solr/SolJava

11.7 Using Solr From Ruby

For Ruby applications, the `solr-ruby` gem encapsulates the fundamental Solr operations.

At a command line, install `solr-ruby` as follows:

```
$ gem install solr-ruby
Bulk updating Gem source index for: http://gems.rubyforge.org
Successfully installed solr-ruby-0.0.7
1 gem installed
Installing ri documentation for solr-ruby-0.0.7...
Installing RDoc documentation for solr-ruby-0.0.7...
```

This gives you a `Solr::Connection` class that makes it easy to add documents, perform queries, and do other Solr stuff.

`solr-ruby` takes advantage of Solr's Ruby response writer, which is a subclass of the JSON response writer. This response writer sends information from Solr to Ruby in a form that Ruby can understand and use directly.

11.7.1 Performing Queries

To perform queries, you just need to get a `Solr::Connection` and call its query method. Here is a script that looks for cheese. The return value from `query()` is an array of documents, which are dictionaries, so the script iterates through each document and prints out a few fields.

```
require 'rubygems'
require 'solr'

solr = Solr::Connection.new('http://localhost:8983/solr')

response = solr.query('cheese')

response.each do |hit|
    puts hit['id'] + ' ' + hit['name'] + ' ' + hit['price'].to_s
end
```

An example run looks like this:

```
$ ruby query.rb
551299 Gouda cheese wheel 49.99
123 Fresh mozzarella cheese
```

11.7.2 Indexing Documents

Indexing is just as simple. You have to get the `Solr::Connection` just as before. Then call the `add()` and `commit()` methods. You're done!

```
require 'rubygems'
require 'solr'

solr = Solr::Connection.new('http://localhost:8983/solr')

solr.add(:id => 123, :name => 'Fresh mozzarella cheese')
solr.commit()
```

11.7.3 More Information

For more information on solr-ruby, read the page at the Solr Wiki:

http://wiki.apache.org/solr/solr-ruby

11.8 Summary

Although Solr is most readily available as a Web application, any type of client application can take advantage of Solr's power. This chapter describes the available client APIs and provides examples for accessing Solr from Java applications and Ruby applications. The example code that accompanies this document includes working examples that add documents and perform queries.

This page is intentionally left blank.

Alphabetical Index

lucid
IMAGINATION

1875 South Grant Street, 10th Floor
San Mateo, California 94402 USA
tel 1 650 353 4057

www.lucidimagination.com